FMBC 2020: Pre-Proceedings

Bruno Bernardo   Diego Marmsoler

July 20-21, 2020

Preface

The 2nd Workshop on Formal Methods for Blockchains (FMBC) took place virtually on July 20/21 2020 as part of CAV 2020, the 32nd International Conference on Computer-Aided Verification. Its purpose was to be a forum to identify theoretical and practical approaches applying formal methods to blockchain technology.

This second edition of FMBC attracted 18 submissions (10 long papers, 4 short papers, and 4 extended abstracts) on topics such as verification of smart contracts or analysis of consensus protocols. Each paper was reviewed by at least three program committee members or appointed external reviewers. This led to a selection of 10 papers (7 long and 3 short) that will be presented at the workshop as regular talks, as well as 1 long paper and 4 extended abstracts that will be presented as lightning talks. Additionally, we were very pleased to have an invited keynote by Grigore Rosu (University of Illinois at Urbana-Champaign).

This volume contains the papers selected for regular talks, the extended abstracts and paper selected for lightning talks as well as the abstract of the invited talk.

We thank all the authors that submitted a paper, as well as the program committee members and external reviewers for their immense work. We are grateful to Shuvendu Lahiri and Chao Wang, Program Chairs of CAV 2020, and to Zvonimir Rakamaric, Workshop Chair of CAV 2020, for their support and guidance. Finally, we would like to express our gratitude to our sponsor Nomadic Labs for its generous support.

July 2020

Bruno Bernardo
Diego Marmsoler

Program Committee

Wolfgang Ahrendt	Chalmers University of Technology, Sweden
Lacramioara Astefanoei	Nomadic Labs, France
Massimo Bartoletti	University of Cagliari, Italy
Bernhard Beckert	Karlsruhe Institute of Technology, Germany
Bruno Bernardo	Nomadic Labs, France
Achim Brucker	University of Exeter, UK
Silvia Crafa	Universita di Padova, Italy
Zaynah Dargaye	Nomadic Labs, France
Jérémie Decouchant	University of Luxembourg, Luxembourg
Ansgar Fehnker	University of Twente, Netherlands
Georges Gonthier	Inria, France
Maurice Herlihy	Brown University, USA
Florian Kammueller	Middlesex University London, UK
Igor Konnov	Informal, Austria
Andreas Lochbihler	Digital Asset, Switzerland
Diego Marmsoler	University of Exeter, UK
Anastasia Mavridou	NASA Ames, USA
Simão Melo de Sousa	Universidade da Beira Interior, Portugal
Andrew Miller	University of Illinois at Urbana-Champaign, USA
Karl Palmskog	KTH, Sweden
Vincent Rahli	University of Birmingham, UK
Andreas Rossberg	Dfinity Foundation, Germany
Claudio Russo	Dfinity Foundation, USA
César Sanchez	Imdea, Spain
Clara Schneidewind	TU Wien, Austria
Ilya Sergey	Yale-NUS College/NUS, Singapore
Bas Spitters	Aarhus University/Concordium, Denmark
Mark Staples	CSIRO Data61, Australia
Meng Sun	Peking University, China
Simon Thompson	University of Kent, UK
Philip Wadler	University of Edinburgh / IOHK, UK

Supporting Reviewers

Luis Arrojado da Horta
Yi Li
João Santos Reis
Ralf Sasse
Søren Eller Thomsen

Papers

Keynote: Formal Design, Implementation and Verification of Blockchain Languages using K

Abstract

The usual post-mortem approach to formal language semantics and verification, where the language is firstly implemented and used in production for many years before a need for formal semantics and verification tools naturally arises, simply does not work anymore. New blockchain languages or virtual machines are proposed at an alarming rate, followed by new versions of them every few weeks, together with programs (or smart contracts) in these languages that are responsible for financial transactions of potentially significant value. Formal analysis and verification tools are therefore needed immediately for such languages and virtual machines. We will present recent academic and commercial results in developing blockchain languages and virtual machines that come directly equipped with formal analysis and verification tools. The main idea is to generate all these automatically, correct-by-construction from a formal language specification.

Bio

Grigore Rosu is a professor in the Department of Computer Science at the University of Illinois at Urbana-Champaign (UIUC), where he leads the Formal Systems Laboratory (FSL), and the founder of Runtime Verification, Inc (RV). His research interests encompass both theoretical foundations and system development in the areas of formal methods, software engineering and programming languages. Before joining UIUC in 2002, he was a research scientist at NASA Ames. He obtained his Ph.D. at the University of California at San Diego in 2000. He was offered the CAREER award by the NSF, the Dean's award for excellence in research by the College of Engineering at UIUC in 2014, and the outstanding junior award by the Computer Science Department at UIUC in 2005. He won the ASE IEEE/ACM most influential paper award in 2016 (for an ASE 2001 paper) and the RV test of time award (for an RV 2001 paper) for papers that helped shape the runtime verification field, the ACM SIGSOFT distinguished paper awards at ASE 2008, ASE 2016, and OOPSLA 2016, and the best software science paper award at ETAPS 2002.

Part I.

Smart contracts and payments

A Blockchain Model in Tamarin and Formal Analysis of Hash Time Lock Contract

Colin Boyd, Kristian Gjøsteen, and Shuang Wu

NTNU - Norwegian University of Science and Technology, Trondheim, Norway
{`colin.boyd, kristian.gjosteen, shuang.wu` }@ntnu.no

Abstract

Formal analysis and verification methods can aid the design and validation of security properties in blockchain based protocols. However, to generate a reasonable and correct verification, a proper model for the blockchain is needed. In this paper, we give a blockchain model in Tamarin. Based on our model we analyze and give a formal verification for the hash time lock contract, an atomic cross chain trading protocol. The result shows that our model is able to identify an underlying assumption for the hash time lock contract and that the model is useful for analyzing blockchain based protocols.

1 Introduction

In a blockchain based protocol, the blockchain serves as a reliable public ledger to deliver ordered outcomes to all its agents. Protocols can be executed by using smart contracts and the execution states are recorded on the blockchain. The blockchain essentially performs as a distributed trusted party to reduce the direct trust between the entities in the system.

In order to formally verify the security properties of protocols built on top of blockchains, a proper model for blockchains is needed. The model must capture the interesting properties of blockchains, without becoming too complicated. A blockchain is more than a public ledger. The dynamics of the growing chain provide a time reference: the relatively stable growth of the blockchain height offers a 'global time'. With respect to this global time, a blockchain enables a time lock function used as a restriction specifying that a transaction cannot be added to blockchain before a set time (actually a given chain height). Thus in order to capture properties of time lock contracts a blockchain model should include the following features.

Model time. The blockchain model should contain a global time reference in the system. Different blockchains contain different global time references. The model should be able to capture time-relevant risks, such as race conditions.

Model the time lock restriction. The time out event of a time lock should be triggered by the time reference. It should be possible to model the risk introduced by a time lock that times out earlier or later than is expected.

Clarify the underlying assumptions. If a certain property of the blockchain fails, a protocol built on top of it will not be safe either.

Related work. In 2014, Andrychowicz et al. [2] modeled a multiparty computation contract in Bitcoin by using timed automata. Back then the time lock functionality in Bitcoin was limited and consequently the structure of the contract is different today. Bursuc and Kremer [4] used Tamarin to model the blockchain as a public ledger, and analysed the ZKCP [7] protocol built on top of it. But in their model, the executions are not time-relevant. Turuani et al. [10] give a formal model in ASLan++ of the two-factor authentication protocol used by the Electrum

Bitcoin wallet. Bentov et al. [3] propose a real-time cryptocurrency exchange service, and they give an informal cryptographic proof for the security of a hash time lock protocol, with a probabilistic modeling of forking. Sun and Yu [9] give a formal verification model for five kinds of security issues in the Ethereum blockchain using Coq.

Our contributions: To address the above challenges, we build a blockchain model in Tamarin [8]. The model defines a public ledger and a global time reference for the system, with time lock functionality built on top. We also define the security properties of an atomic cross chain trading protocol and give a formal proof for the security of the hash time lock contract (HTLC). To our knowledge, this is the first HTLC analysis by formal verification tools. The proof clarifies a 'hidden' security assumption: the growth speed of the two blockchains need to be stable, otherwise security will fail. We further use our model to analyze an older version of the hash time lock contract, and Tamarin is able to find a flaw. Even if the assumption looks trivial and the flaw is somewhat obscure, this demonstrates that our model is able to address the above challenges and can be used for formal verification of blockchain based protocols.

2 Background

2.1 Hash time lock contract

The goal of the hash time lock contract (HTLC) is to exchange different cryptocurrencies between two players in a decentralized way. Consider Alice who wants to exchange Bitcoin for Altcoin, and Bob who wants to exchange Altcoin for Bitcoin. They could do the following:

1. Alice creates a transaction that is locked by a hash value $\mathsf{h} := \mathsf{H}(\mathsf{sk})$ to send Bob 1 Bitcoin. Bob can take the funds only if he can provide the hash pre-image. This is Alice's *commitment transaction.*
2. After Alice's commitment transaction has been confirmed on the Bitcoin blockchain, Bob creates a transaction (contract) to send 1 Altcoin to Alice, locked using the same hash value $\mathsf{h} := \mathsf{H}(\mathsf{sk})$. This is Bob's commitment transaction.
3. Alice takes Bob's Altcoin by providing her signature and the pre-image of the hash lock. Bob learns the hash pre-image and unlocks the Bitcoin that Alice sent to him.

In order to avoid an interrupted protocol leaving players' funds locked forever, the commitment transactions are also locked by time locks. After the time lock times out, the transaction can be redeemed by the sender. The time lock of Alice's commitment transaction should be longer than Bob's commitment transaction, since in the case that Alice takes Bob's Altcoin at the last moment before Bob's commitment transaction timed out, her commitment is still locked by the time lock and Bob still has time to take Alice's Bitcoin. A successful execution of a hash time lock contract can be seen in figure 1. Notice that in the figure we use the same structure (script) to describe Altcoin and Bitcoin, but in fact we just consider two Bitcoin-like blockchains. As long as the blockchain supports both timelock and hash lock functionalities, the hash time lock contract protocol can be used.

The above description is the latest version of the hash time lock contract [6], where a time lock restricts when a transaction can be spent by its following transaction. Thus the two potential outputs of a commitment transaction are specified inside the commitment transaction. The previous version [5, 1] utilizes a time lock that only restricts when a transaction can be added to blockchain. The time lock is then not specified in the commitment transaction, but

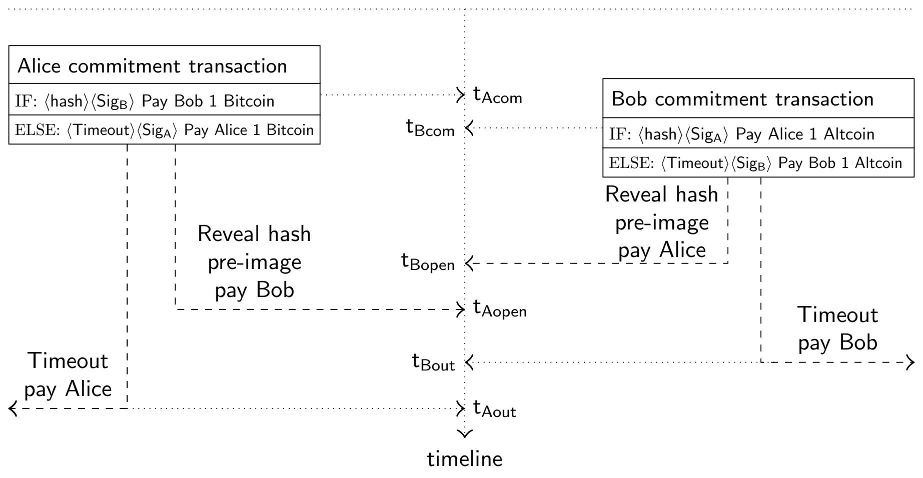

Alice commitment transaction
IF: ⟨hash⟩⟨Sig_B⟩ Pay Bob 1 Bitcoin
ELSE: ⟨Timeout⟩⟨Sig_A⟩ Pay Alice 1 Bitcoin
Bob commitment transaction
IF: ⟨hash⟩⟨Sig_A⟩ Pay Alice 1 Altcoin
ELSE: ⟨Timeout⟩⟨Sig_B⟩ Pay Bob 1 Altcoin
t_Acom
t_Bcom
Reveal hash pre-image pay Alice
t_Bopen
Reveal hash pre-image pay Bob
t_Aopen
Timeout pay Bob
t_Bout
Timeout pay Alice
t_Aout
timeline

Figure 1: Hash time lock contract execution.

in the redeem transaction. In this case the redeem transaction must be signed by multiple signatures, thus the procedure involves two players exchanging signatures on transactions.

2.2 About Tamarin

Tamarin [8] is an automatic symbolic protocol verification tool. Given a protocol, the user specifies the roles running the protocol and their behaviors, the adversary model and the security properties by using the Tamarin programming language. Tamarin applies malicious adversarial behavior to the roles and uses a backward search method to generate counterexamples to the security claims. Tamarin ends up with either a proof that demonstrates that the given protocol satisfies the security properties, or Tamarin would give an attack for a failed security claim.

In Tamarin, the communication messages, fresh randomness and the states of the protocols are represented by symbolic terms called facts. There are two special facts to model the interaction with the untrusted environment: $\mathsf{In}(*)$, $\mathsf{Out}(*)$,representing the protocol's input and output from and to the environment. All the messages forwarded by $\mathsf{In}(*)$ and $\mathsf{Out}(*)$ can be learned by the adversary. The fact $\mathsf{K}(\mathsf{x})$ denotes the adversary learning x. Some facts are linear, which means that they can be used only once. The protocols and the specifications of the adversaries are modeled by using multiset rewriting rules. These rules and facts define a labeled transition system. Security properties are either defined in terms of traces of the transition system or the observational equivalence of two transition systems.

A role in the protocol is specified by Tamarin multiset rewriting rules. A rule consists of three elements: $(\mathsf{L}, \mathsf{A}, \mathsf{R})$:$[\mathsf{L}] - [\mathsf{A}] \rightarrow [\mathsf{R}]$, the left side facts L (states, messages of the protocol) are the premises of the rule, the right side facts R are rule conclusions, and the actions in the middle square brackets A are to label the traces. A rule can be executed as long as its premises exist in the current system states. Then the facts in the premises will be removed from the current system states, while the facts in conclusion will be added. Users can also add restrictions to enforce that only traces satisfying the restrictions are considered by Tamarin's backward search.

We illustrate Tamarin syntax by introducing a toy Diffie-Hellman key exchange protocol:

```
rule Server_1: [ Fr( ~a ) ]-->[ S_1( ~a, 'g' ^~a), Out( 'g' ^~a) ]
rule Client: [ Fr( ~b ), In( X ) ]--[ Key( X ^~b ) ]->[ Out( 'g' ^~b ) ]
rule Server_2: [ S_1( ~ a, 'g' ^~a ), In( Y ) ]--[ Key( Y ^~a ) ]->[ ]
```

In the first step, the server generates fresh randomness ~a (the symbol ~ denotes a fresh nonce, the function Fr(∗) means generating a fresh nonce), sends g^a to client by the fact Out('g' ^ ~a), and it records the inner state by the fact S_1(~ a, 'g' ^~a) . This state will be used in next step of the server with the name Server_2.

The client receives the message from server by fact In(X), it then generates the session key according to the Diffie-Hellman key exchange protocol. This trace and its parameters are recorded by the action Key(X^~b), this will later be used to claim the security property of the protocol. The server's next step generates a similar action.

The security properties to be evaluated are defined by lemmas. In the above example we want to claim there is no adversary that can learn the secret key.

```
Lemma Key_secrecy " All key #i . Key( key ) @i ==> not Ex #j . K( key ) @j "
```

The lemma Key_secrecy specifies that in all the traces that have an action Key(key), no adversary could learn the input of the action, namely, the value key, expressed by statement that there is no fact K(key) in the trace.

3 Tamarin Blockchain model

3.1 Simplification

A complete blockchain model would be too complex for Tamarin to work with, if it could even be expressed. We have simplified the structures of the transactions and blocks to make our blockchain model simpler, while still expressive enough to capture the essential elements for describing attacks on the protocols, and thus making verification possible. We let the blocks only include zero or one transactions, and forks are not allowed, thus we only consider the blocks that are already stable. The consensus protocol and cost are not modeled in our work. A transaction contains six elements: the id of the transaction that is being spent, the sender's address (we simply denote the addresses as public keys), the input signature (or script), output address (or script), the block sequence and the id of this transaction.

We set the relative growing speeds of the two blockchains to be the same. This simplification will not change the primary mechanism of the protocol because if the speed of Alice's blockchain is two time faster than Bob's blockchain, the time lock of Alice will be twice as long to ensure that it is longer than Bob's time lock in real time.

3.2 Tamarin blockchain model rules

We describe the rules of our blockchain model in two parts: the ledger rules and the global time rules. The ledger rules add a transaction to a block. The global time rules generate the time state called 'Tick' to specify the time point of a block being added to the blockchain.

In the global time rules, each time Tick has a unique parameter time. (It also has another parameter to tie a Tick to a specific blockchain, so that block chains can grow at different speeds. For simplicity we leave it out of Figure 2.) When generating a new Tick, an older Tick that has the largest time will be consumed and the time will be increased by one. Thus the Ticks form a time state transition chain that we call a Tickchain. Given the uniqueness of each Tick and since 'time' is always increasing, each Tick can be considered as an empty block and the Tickchain can serve as base for a blockchain. We refer the blockchain in our Tamarin

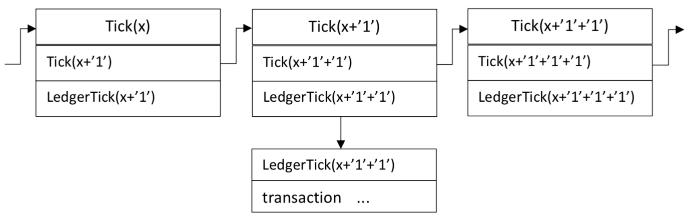


Figure 2: Tick chain.

model as Tickchain and its blocks are called Tickblocks. In order to model adding a transaction to a certain Tickblock, a LedgerTick with a parameter Height equal to time is generated along with a new Tick. The ledger rules consume a LedgerTick to create a new transaction. In this way we bind a transaction to a Tickblock. The parameter Height also implies a sequence of transactions. After the executions of a protocol, there may be some LedgerTicks left without being consumed, which means that no transaction was added to the corresponding Tickblock.

Global time rules: There are two rules: Tick_start and Tick to create a blockchain. (We also use Tick to name the rule that generates a Tick state.) The rule Tick_start initiates the clock and the rule Tick updates the clock, i.e. increase the clock by adding '1'. There are three facts involved in the global time rules: Chain(BC), Tick(BC, time) and LedgerTick(BC, height). Chain(BC) specifies which blockchain. Tick(BC, x) and LedgerTick(BC, x) denote a certain block with the block height x. Tick(BC, x) will be consumed by the Tick rules to updates the clock by iteration. LedgerTick(BC, x) will be consumed by the ledger rules to link a transaction to a block. All these facts are linear facts that can be only consumed one time.

Global time rules

Tick_Start	Tick
Input: Chain(BC)	Input: Tick(BC, time)
Output: Tick('1'),LedgerTick('1')	Output: Tick(BC, time +' 1'), LedgerTick(BC, time +' 1')

Ledger rules: The ledger rules model the nodes in blockchain network: the nodes get transaction information from the network, check its validity and then record the transaction to the blockchain.

There are two types of transaction in our model: SimpleTx(BC, InTx, InSig, OutPk, tx, height) to model the transactions without the hash and time lock, and CommitTx(BC, InTx, InSig, OutScript, tx, height) to model the transactions locked by a hash and a time lock. In these two transactions, BC denotes which blockchain the transaction belongs to; InTx is a nonce that identifies a previous unspent transaction owned by the sender; InSig is the sender's signature. tx is a nonce that identifies this transaction; height specifies in which block this transaction has been recorded. In the simple transaction, the OutPk is the receiver's address, while the OutScript in a commitment transaction is a hash time lock contract script, specifying the hash value, time lock value and receiver's address.

There are five rules to model the blockchain behaviors, Mine_Coin, Simple_Tx, Commit_Tx, Commit_open and Commit_timeout. The purpose of these rules are to generate blocks that contain different types of transactions and append the block to the blockchain. Mine_Coin creates the original coins of the blockchain. Simple_Tx spends a simple transaction and creates a new unspent simple transaction. Commit_Tx creates a transaction that is locked by a hash and a time lock. Commit_open models the transaction that is spent by revealing the hash pre-image. The Commit_timeout model the commit transaction that is spent by sender redeeming the transaction in the case of timeout.

Ledger rules

<u>Mine_Coin</u>

Input: $\mathsf{Fr}(\sim n),\ \mathsf{PK}(A, pkA),\ \mathsf{ledgerTick}(BC, t)$
Output: $\mathsf{SimpleTx}(BC, '0', '0', pkA,\ n, t)$

<u>Simple_Tx</u>

Input: $\mathsf{SimpleTx}(BC, InTx, InSig, Pk, n, height)$,
$\mathsf{In}(\langle tx, Sig, pkB\rangle),\ \mathsf{ledgerTik}(BC, t)$
Output: $\mathsf{SimpleTx}(BC, n, Sig, pkB, tx, t)$

<u>Commit_Tx</u>

Input: $\mathsf{SimpleTx}(BC, InTx, InSig, Pk, n, height)$,
$\mathsf{In}(\langle Sig, \langle pkA, timelock, hash, pkB\rangle\rangle)$,
$\mathsf{LedgerTick}(BC, t)$
Output: $\mathsf{Commit\_Tx}(BC, n, Sig, \langle pkA, timelock, hash, pkB\rangle, tx, t)$

<u>Commit_open</u>

Input: $\mathsf{Commit\_Tx}(BC, InTx, InSig, \langle pkA, timelock, hash, pkB\rangle, n, height)$,
$\mathsf{In}(\langle\langle Script1, Script2\rangle, PKaddress\rangle)$,
$\mathsf{LedgerTick}(BC, t)$,
Output: $\mathsf{SimpleTx}(BC, n, Sig, pkB, tx, t)$

<u>Commit_timeout</u>

Input: $\mathsf{Commit\_Tx}(BC, InTx, InSig, \langle pkA, timelock, hash, pkB\rangle, n, height)$
$\mathsf{In}(\langle Script1, PKaddress\rangle)$
$\mathsf{LedgerTik}(BC, t)$
Output: $\mathsf{SimpleTx}(BC, n, Script1, Pk, tx, t)$

Restrictions The no double spending property of blockchain is guaranteed by restriction rule. When a transaction has been spent, an action Spend(BC, tx, M, t) will be recorded in the Tamarin system to identify the event. On a single blockchain, for a transaction x, there can only exist one Spend(BC, x, M, t).

```
restriction DoubleSpending:
"All BC x n m t1 t2 #i #j .Spend(BC,x,n,t1)@i &Spend(BC,x,m,t2)@j==>#i=#j"
```

To help Tamarin reason more efficiently, we add one more restriction HappenBefore. This restriction simply tells Tamarin that a transaction that has a larger block number should happen later than a transaction that has a smaller block number.

```
restriction HappenBefore:
"All BC t1 t2 #i .HappenBefore(BC,t1,t2)@i==>Ex x .t2=t1+x"
```

4 Model HTLC in Tamarin

We model the two roles Alice and Bob in the hash time lock contract. Alice is the contract initiator, Bob is the responder. Alice is not allowed to set up a hash time lock contract with herself. The roles send data to blockchain network by using fact Out(∗), i.e. they send data to the environment directly. It models the blockchain network as public, where the adversary learns anything sent to and received from the network.

4.1 HTLC rules

Alices' rules: Alice is defined by two rules: Alice_send and Alice_receive. The rule Alice_send broadcasts Alice's commitment transaction and redeem transaction to the blockchain network.

The rule Alice_receive broadcasts the transaction to open Bob's commitment transaction. Note that even though Alice broadcasts her commitment transaction and its redeem transaction at the same time, the redeem transaction cannot be added to the blockchain until the time lock of Alice's commitment transaction expires.

Alice_send: The rule takes a simple transaction tx, Alice's secret keys, Bob's address and a fresh nonce as input. It outputs Alice's commitment transaction, Alice's redeem transaction, and a state Alice_1_record to record the hash pre-image. It spends the simple transaction tx with signature SigA and outputs a commitment transaction that has $\langle \mathsf{pk}(\mathsf{ltkA1}), \mathsf{timelock\_A}, \mathsf{hash}, \mathsf{pkB3}\rangle$ as output. The two potential ways to spend this commitment transaction are: 1) Redeem by Alice: when the time lock timelock_A timed out, Alice could redeem the commitment transaction by providing the signature of the public key pk(ltkA1). 2) Opened by Bob: Bob can take the funding by providing the pre-image of the hash lock and the signature of pkB3. The rule outputs the redeem transaction at the same time, since Alice desires to broadcast the redeem transaction earlier so that it can be added on the blockchain as soon as the time lock expires. The Tamarin code is listed below.

```
rule Alice_send:
   let
     timelock_A='1'+'1'
     hash=HTLChash(~hsk)
     SigA=sign(<'BC1',tx,pk(ltkA),<pk(ltkA1),timelock_A,hash,pkB3>>,ltkA)
     CommitTxAlice=TXhash(<tx,SigA,<pk(ltkA1),timelock_A,hash,pkB3>>)
     SigA1=sign(<'BC1',CommitTxAlice,<pk(ltkA1),timelock_A,hash,pkB3>,pkA2>,ltkA1)
   in
    [ !SimpleTx('BC1','0','0',pk(ltkA),tx,t) ,!PK(A,pk(ltkA1)),!PK(A,pkA2),!PK(B,pkB3),
    Fr(~hsk)]
  --[ InEq(A,B) ]->
    [ Out(<tx,SigA,<pk(ltkA1),timelock_A,hash,pkB3>>),Out(<CommitTxAlice,SigA1,pkA2>)
      ,Alice_1_record(hash,~hsk)]
```

Alice_receive: The rule takes a commitment transaction that has Alice as receiver, a state record Alice_1_record and Alice's address as inputs. It outputs a transaction spending the commitment transaction and a fact Reveal(hsk). The rule opens the pre-image of hash and provide the signature SigA3 for pkA3. This spending transaction will be added to the blockchain by the ledger rule Commit_open, it transfers the funding to the target address.

```
rule Alice_receive:
  let
    SigA3=sign(<'BC2',CommitTxBob,<pkB1,timelock_B,hash,pk(ltkA3)>,pkA4>,ltkA3)
  in
    [!CommitTx('BC2',tx0,SigB0,<pkB1,timelock_B,hash,pk(ltkA3)>,CommitTxBob,t)
    ,Alice_1_record(hash,hsk),!PK(A,pkA4)]
  --[ Alice_receive(CommitTxBob) ]->
    [ Out(<CommitTxBob,<hsk,SigA3>,pkA4,hsk>)]
```

Bob's rules: Bob is specified by the rules Bob_send, Bob_receive and a restriction Not_Spend. The rule Bob_send generates the commitment transaction and its redeem transaction, and broadcasts them to the blockchain network. The rule Bob_receive is to open Alice's commitment transaction and the restriction Not_Spend checks that Alice's commitment transaction has not been spent.

Bob_send: The rule takes Alice's commitment transaction, a simple transaction, Alice's receiving address, Bob's redeem address and Bob's secret key as input. The restriction Not_Spend checks if Alice's commitment transaction is just added to the blockchain. If it is, the rule will output Bob's commitment transaction, its redeem transaction and Bob_1_record to record the hash lock and the transaction id of Alice's commitment transaction. The output script in Bob's commitment transaction is $\langle \mathsf{pk}(\mathsf{ltkB}), \mathsf{timelock\_B}, \mathsf{hash}, \mathsf{pkA3}\rangle$. The hash is the same with the hash lock in Alice's commitment transaction.

```
rule Bob_send:
  let
    timelock_B='1'
    SigB=sign(<'BC2',tx,pk(ltkB),<pk(ltkB1),timelock_B,hash,pkA3>>,ltkB)
    CommitTxBob=TXhash(<tx,SigB,<pk(ltkB1),timelock_B,hash,pkA3>>)
    SigB1=sign(<'BC2',CommitTxBob,<pk(ltkB1),timelock_B,hash,pkA3>,pkB2>,ltkB1)
  in
    [!SimpleTx('BC2','0','0',pk(ltkB),tx,t1),!PK(B,pk(ltkB1)) ,!PK(B,pkB2),!PK(A,pkA3)
    ,!CommitTx('BC1',tx_0,SigA_0,<pkA,timelock_A,hash,pkB>,CommitTxAlice,t)]
  --[Not_Spend(CommitTxAlice)]->
    [ Bob_1_record(hash,CommitTxAlice)
     ,Out(<tx,SigB,<pk(ltkB1),timelock_B,hash,pkA3>>)
     ,Out(<CommitTxBob,SigB1,pkB2>)]
```

Bob_receive: The rule takes a state record Bob_1_record, a fact Real(hsk), Bob's address and Alice's commitment transaction as inputs. It outputs the transaction to open Alice's commitment transaction. By providing the signature of the public key pk(ltkB3) and the pre-image of the hash lock, Bob transfers the funding to his address pkB4.

```
rule Bob_receive:
  let
    SigB3=sign(<'BC1',CommitTxAlice,<pkA1,timelock_A,hash,pk(ltkB3)>,pkB4>,ltkB3)
  in
    [ Bob_1_record(hash,CommitTxAlice),Reveal(hsk),!PK(B,pkB4)
    ,!CommitTx('BC1',tx0,SigA0,<pkA1,timelock_A,hash,pk(ltkB3)>,CommitTxAlice,t)]
  --[ Bob_receive(CommitTxAlice) ]->
    [ Out(<CommitTxAlice,<hsk,SigB3>,pkB4,hsk>) ]
```

5 Tamarin Security analysis

5.1 Preliminaries

We describe a transaction as a tuple of six elements: $\mathsf{TX}\{\mathsf{BC}, \mathsf{InTx}, \mathsf{InSig}, \mathsf{Output}, \mathsf{n}, \mathsf{height}\}$, where BC is the blockchain which this transaction belongs to, InTx is the ID of the input transaction, InSig is the input signature, n is the id of this transaction, and height specifies which block contains this transaction. For a simple transaction, the parameter Output is simply a public key, while for a commitment transaction, the Output will be a tuple $\langle \mathsf{pk}_1, \mathsf{timelock}, \mathsf{hash}, \mathsf{pk}_2\rangle$ that contains two public keys pk_1 and pk_2, a time lock and a hash lock. The commitment transaction can be spent by revealing the hash pre-image and the signature of pk_2 or providing the signature of pk_1 if the time lock timed out. The parameter height is ignored if a transaction is not recorded on the blockchain yet.

For a specific time lock, we denotes its value as Δ. It restricts a commitment transaction can be spent only if there is at least Δ blocks appended after the block that contains this commitment transaction. We specify the corresponding real time duration of generating these Δ blocks as δ. The relationship is typically simple, for instance, in the Bitcoin blockchain the approximate time to generate 20 blocks is 200 minutes, so with timelock $\Delta = 20$ we get real time $\delta = 200$. The reason why the real time also involved in the formula is because we are dealing with two blockchains. Each of the two blockchains can be seen as a time reference, but these two time references might get out of sync, thus we need a single global clock.

When a commitment transaction is added on the blockchain, we denote the event as $\{\Gamma^{\mathsf{h_{sk}},\Delta_\mathsf{A}}_{\mathsf{Acom}}, \mathsf{t}, \mathsf{Tick}\}$, which means Alice's commitment transaction is recorded on blockchain at time point t in block sequence Tick, locked by Δ_A and a hash with pre-image $\mathsf{h_{sk}}$. The open and timeout of the commitment transaction are specified as $\{\Gamma^{\mathsf{h_{sk}},\Delta}_{\mathsf{Aopen}}, \mathsf{t}, \mathsf{Tick}\}$ and $\{\Gamma^{\mathsf{h_{sk}},\Delta}_{\mathsf{Ared}}, \mathsf{t}, \mathsf{Tick}\}$, respectively.

5.2 Security claim

For Alice, the hash time lock contract should satisfy the first two properties. For Bob, the protocol should guarantee the last two security properties:
Property 1 Bob cannot open Alice's commitment transaction and take her funding unless Bob has created a commitment transaction to Alice.

$$\forall\{\Gamma^{h_{sk},\Delta}_{Aopen}, t_{Aopen}, Tick_{Aopen}\} ==>\exists\{\Gamma^{h_{sk},\Delta}_{Bcom}, t_{Bcom}, Tick_{Bcom}\}$$

The equation claims that for all the events that Alice's commitment transactions has been opened, there must exist an event that Bob made a commitment transaction before. The commitment transaction made by Bob should use the same hash lock generated from h_{sk} and it sends funding to Alice's address.

```
lemma Security_1_Alice:
" All A tx1 SigA pkA1 timelock_A hash pkB3 CommitTxAlice
      TickAcom TickAopen #tAcom #tAopen #Apk1 .


    !PK(A,pkA1)@Apk1
    &!CommitTx('BC1',tx1,SigA,<pkA1,timelock_A,hash,pkB3>,CommitTxAlice,TickAcom)@tAcom
    &Spend('BC1',CommitTxAlice,'CommitOpen',TickAopen)@tAopen



  ==>Ex tx2 SigB pkB1 timelock_B pkA3 CommitTxBob
        TickBcom #tBcom #Apk2 .

      !PK(A,pkA3)@Apk2
      &!CommitTx('BC2',tx2,SigB,<pkB1,timelock_B,hash,pkA3>,CommitTxBob,TickBcom)@tBcom
      &#tBcom<#tAopen
"
```

Tamarin verifies the first security claim is true.
Property 2 Bob can redeem his funding only if the time lock of his commitment transaction timed out.

$$\forall(\{\Gamma^{h,\Delta_B}_{Bcom}, t_{Bcom}, Tick_{Bcom}\} \wedge \{\Gamma^{h,\Delta_A}_{Bred}, t_{Bred}, Tick_{Bred}\})==> t_{Bred} > t_{Bcom} + \delta_B$$

Since in a single blockchain, a transaction recorded early has smaller height than those recorded later. We reduce this security property to:

$$\forall(\{\Gamma^{h,\Delta_B}_{Bcom}, t_{Bcom}, Tick_{Bcom}\} \wedge \{\Gamma^{h,\Delta_A}_{Bred}, t_{Bred}, Tick_{Bred}\})==> Tick_{Bred} > Tick_{Bcom} + \Delta_B$$

The equation claims that the duration between the time point Bob's commitment transaction is added to the blockchain and the time point Bob's redeem transaction is added to the blockchain is always larger than the duration of its time lock. Bob cannot redeem his commitment transaction before it timed out.This property guarantees that there is no race condition between Bob's redeem transaction and the transaction of Alice to open Bob's commitment transaction.

```
lemma Security_2_Alice:
" All tx2 SigB pkB1 timelock_B hash pkA3 CommitTxBob TickBcom #tBcom
      TickBTout #tBTout .

    !CommitTx('BC2',tx2,SigB,<pkB1,timelock_B,hash,pkA3>,CommitTxBob,TickBcom)@tBcom
    &Spend('BC2',CommitTxBob,'CommitTout',TickBTout)@tBTout
 ==>Ex x. TickBTout=TickBcom+timelock_B+x
"
```

Tamarin verifies the above security claim is true.

Property 3 After Alice takes Bob's funding, Bob has time to take Alice's funding before Alice's commitment transaction time out.

$$\forall(\{\Gamma^{h,\Delta_A}_{Acom}, t_{Acom}, Tick_{Acom}\} \wedge \{\Gamma^{h,\Delta_B}_{Bopen}, t_{Bopen}, Tick_{Bopen}\} ==> t_{Acom} + \delta_A > t_{Bopen}$$

The equation claims that if Alice takes Bob's funding at the last moment before it timed out, Bob should always have some time left before Alice's commitment transaction is timed out. This property avoids the risk of the race condition between Bob opening Alice's commitment transaction and Alice redeem her commitment transaction.

```
lemma Security_3_Bob:
"
All CommitTxAlice hash timelock_A pkA1 tx1 SigA pkB3 TickAcom #tAcom
     CommitTxBob timelock_B pkB1 tx2 SigB pkA3 TickBcom #tBcom
       #tBopen1 #tATout1 #tBopen #tATout .

   !CommitTx('BC1',tx1,SigA,<pkA1,timelock_A,hash,pkB3>,CommitTxAlice,TickAcom)@tAcom
   &!CommitTx('BC2',tx2,SigB,<pkB1,timelock_B,hash,pkA3>,CommitTxBob,TickBcom)@tBcom


   &Spend('BC2',CommitTxBob,'CommitOpen',TickBcom+timelock_B)@tBopen1
   &LedgerTick('BC2',TickBcom+timelock_B)@tBopen

   &LedgerTick('BC1',TickAcom+timelock_A+'1')@tATout
   &Spend('BC1',CommitTxAlice,'CommitTout',TickAcom+timelock_A+'1')@tATout1
==> #tBopen<#tATout
"
```

Tamarin gives a counterexample to this security claim, because the growth speed of the blockchains may differ. We explain in detail in the next subsection.

Property 4 Alice could redeem her funding only if her commitment transaction timed out.

$$\forall(\{\Gamma^{h,\Delta_A}_{Acom}, t_{Acom}, Tick_{Acom}\} \wedge \{\Gamma^{h,\Delta_A}_{Ared}, t_{Ared}, Tick_{Ared}\}) ==> t_{Ared} > t_{Acom} + \delta_A$$

The equation removes the same race condition risk that Alice has as described in security property 2.

```
lemma Security_4_Bob:
"
All tx1 SigA pkA1 timelock_A hash pkB3 CommitTxAlice TickAcom #tAcom #tATout
      TickATout .

   !CommitTx('BC1',tx1,SigA,<pkA1,timelock_A,hash,pkB3>,CommitTxAlice,TickAcom)@tAcom
   &Spend('BC1',CommitTxAlice,'CommitTout',TickATout)@tATout
==>Ex x. TickATout=TickAcom+timelock_A+x
"
```

Tamarin verifies this security claim is true.

5.3 Discussion on property 3

The failure of property 3 claims that after Alice taking Bob's funding, there exists a case that Bob has no time to open Alice commitment transaction before it expires. Tamarin shows that this attack happens in the case that the blockchain on which Alice made a commit transaction grows faster than is expected. Thus Alice's commitment transaction expires earlier even before Bob's commitment transaction expires. Therefore Alice has the chance to redeem her funding and also take Bob's funding.

Therefore, we need to have a blockchain that not only has liveness and consistency, but also keeps a stable growth speed for the block height. Based on the Tamarin result, we add an extra restriction to restrict the growing speed of the blockchain 'BC2' to be at least as fast as 'BC1', and then evaluate the security property again.

```
restriction stable_growing_blockchain: "All height #i .Tick('BC1',height)@i==>Ex #j.Tick('BC2',height)@j"
```

Tamarin now proves that property 3 holds. Notice that in real scenario we expected both two blockchains should have stable growing speed, but this condition is not necessary for HTLC. The result shows as long as 'BC2' grows relatively no slower than 'BC1', HTLC is secure. The reason is Alice holds the pre-image of the hash, she only need to observe the height of 'BC2' to take Bob's funding before its timelock expires, she doesn't need to worry Bob will take her funding since he doesn't know the hash pre-image. While for Bob, if he publishes his commitment transaction, he needs to make sure Alice cannot withdraw her funding earlier than Alice taking his funding.

6 Analysis on the old version of HTLC

The old version of the hash time lock contract was used when the time lock functionality could only constrain the time point that a certain transaction is allowed to be added to blockchain. In this case the time lock is specified in the redeem transaction rather than the commitment transaction. To make an agreement for the time lock duration of the redeem transaction, the two players need to exchange their signatures on the redeem transaction. The multi-signatures are checked by the nodes before they add the transaction into a block. The signature exchanging procedure is done before the players publish their commitment transaction, otherwise, they might be unable to redeem their commitment transactions.

We claim the same four security properties from section 5 for the old version hash time lock contract. Tamarin verifies that the protocol satisfies the security claims given that Alice is allowed to only use a fixed duration of timelock in the contract. However, in reality Alice might use the timelock with different durations. In this case, there is an attack that allows Alice to redeem her funding earlier than the time period that Bob has signed.

The attack is as follows: Alice will initiate two hash time lock contracts with Bob, these two hash time lock contracts are the same except the second one has longer time lock than the first one. She aborts the first one when she gets Bob's signature on her redeem transaction. Bob will also abort the contract since Alice doesn't publish her commitment transaction. Alice initiates the second contract with Bob, using the same hash lock, but longer time lock. (Bob could in principle notice that he has signed a same hash before, but this requires Bob to keep track of earlier contracts, which is impractical.) In this scenario, after both players publish their commitment transactions to blockchains, Alice can use the redeem transaction of the fist hash time lock contract to unlock her commitment transaction in the second hash time lock contract. Because the second redeem transaction has a shorter time lock, she can redeem the commitment transaction earlier than Bob's expectation.

When we enable different timelocks in our Tamarin model, Tamarin finds the attack and shows that security property 3 fails even with the synchronization between the two blockchain growth speeds.

7 Conclusion

In this paper, we give a formal model for blockchain in Tamarin. Using this model we give a formal verification for the security of hash time lock contract. The verification result from Tamarin shows that the security of HTLC is based on the security assumptions of underlying blockchain, but also requires that the responder blockchain (the blockchain that Bob operates on) needs to grow at least as fast as the initiator's blockchain. This result demonstrates that our Tamarin blockchain model can be used to find security issues in blockchain-based protocols. We note that the verification process of Tamarin needs human guidance to some extent, which could be improved in future work. Also, the model can be improved to allow forks to be more comprehensive.

References

[1] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Łukasz Mazurek. Fair two-party computations via bitcoin deposits. In Rainer Böhme, Michael Brenner, Tyler Moore, and Matthew Smith, editors, *Financial Cryptography and Data Security*, pages 105–121, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[2] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Łukasz Mazurek. Modeling bitcoin contracts by timed automata. In Axel Legay and Marius Bozga, editors, *Formal Modeling and Analysis of Timed Systems*, pages 7–22, Cham, 2014. Springer International Publishing.

[3] Iddo Bentov, Yan Ji, Fan Zhang, Lorenz Breidenbach, Philip Daian, and Ari Juels. Tesseract: Real-time cryptocurrency exchange using trusted hardware. In Lorenzo Cavallaro et al., editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019*, pages 1521–1538. ACM, 2019.

[4] Sergiu Bursuc and Steve Kremer. Contingent payments on a public ledger: models and reductions for automated verification. In *ESORICS 2019 - The 24th European Symposium on Research in Computer Security*, Luxembourg, Luxembourg, September 2019.

[5] Christian Decker and Roger Wattenhofer. A fast and scalable payment network with bitcoin duplex micropayment channels. In Andrzej Pelc and Alexander A. Schwarzmann, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 3–18, Cham, 2015. Springer International Publishing.

[6] Thaddeus Dryja Joseph Poon. The Bitcoin Lightning Network: Scalable off-chain instant payments, 2016. https://lightning.network/lightning-network-paper.pdf.

[7] Gregory Maxwell. Zero knowledge contingent payments, 2011. https://en.bitcoin.it/wiki/ZeroKnowledgeContingentPayment.

[8] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The tamarin prover for the symbolic analysis of security protocols. In *Proceedings of the 25th International Conference on Computer Aided Verification - Volume 8044*, CAV 2013, page 696–701, Berlin, Heidelberg, 2013. Springer-Verlag.

[9] Tianyu Sun and Wensheng Yu. A formal verification framework for security issues of blockchain smart contracts. *Electronics*, 9:255, 02 2020.

[10] Mathieu Turuani, Thomas Voegtlin, and Michael Rusinowitch. Automated verification of electrum wallet. In Jeremy Clark, Sarah Meiklejohn, Peter Y.A. Ryan, Dan Wallach, Michael Brenner, and Kurt Rohloff, editors, *Financial Cryptography and Data Security*, pages 27–42, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

TEZLA, an intermediate representation for static analysis of MICHELSON smart contracts*

João Santos Reis[1], Paul Crocker[2], and Simão Melo de Sousa[1,2]

[1] Nova-Lincs, University of Beira Interior, Portugal
`joao.reis@ubi.pt`
[2] Release Lab, C4 and University of Beira Interior, Portugal
`{desousa|crocker}@di.ubi.pt`

Abstract

This paper introduces TEZLA, an intermediate representation of MICHELSON smart contracts that eases the design of static smart contract analysers. This intermediate representation uses a store and preserves the semantics, flow and resource usage of the original smart contract. This enables properties like gas consumption to be statically verified. We provide an automated decompiler of MICHELSON smart contracts to TEZLA. In order to support our claim about the adequacy of TEZLA, we develop a static analyser that takes advantage of the TEZLA representation of MICHELSON smart contracts to prove simple but non-trivial properties.

1 Introduction

The term "smart contract" was proposed by Nick Szabo as a way to formalize and secure relationships over public networks [22]. In a blockchain, a smart contract is an application written in some specific language that is embedded in a transaction (hence the program code is immutable once it is out in the network). Some examples of smart contracts applications are the management of agreements between parties without resorting to a third party (escrow) and to function as a multi-signature account spending requirement. Smart contracts have the ability to transfer/receive funds to/from users or from other smart contracts and can interact with other smart contracts.

There has been recent reports of bugs and consequently attacks in smart contracts that have led to losses of millions of dollars worth of assets. One of the most famous and most costly of these attacks was on the Distributed Autonomous Organization (DAO), on the Ethereum blockchain. The attacker managed to withdraw approximately 3.6 million ether from the contract.

Given the fact that a smart contract in a blockchain can't be updated or patched, there is an increasing interest in providing tools and mechanisms that guarantee or potentiate the correctness of smart contracts and to verify certain properties.

However, current tools and algorithms for program verification, based for example on deductive verification and static analysis, are usually designed for classical store-based languages in contrast with MICHELSON, the smart contract language for the Tezos Blockchain [12, 2], which is stack based.

To facilitate the usage of such tools to verify MICHELSON smart contracts, we present TEZLA, a store based intermediate representation language for MICHELSON, and its respective tooling. We provide an automated decompiler of MICHELSON smart contracts to TEZLA. The

*This work was supported and funded by the Tezos Foundation by the project FRESCO (Formal Verification of Tezos Smart Contracts).

decompiler preserves the semantics, flow and resource usage of the original smart contract, so that properties like gas consumption can be faithfully verified at the TEZLA representation level. To support our work, we present a case-study of a demo platform for the static analysis of Tezos smart contracts using the TEZLA intermediate representation alongside with an example analysis.

The paper is structured as follows. In section 2 we introduce the syntax and semantics of TEZLA. The decompiler mechanism is described in section 3. Section 4 addresses the static analysis platform case-study that targets TEZLA-represented smart contracts. Finally section 5 concludes with a general overview of this contribution and future lines of work.

2 Tezla

TEZLA aims to facilitate the adoption of existing static analysis tools and algorithms. As such, TEZLA is an intermediate representation of MICHELSON code that uses a store instead of a stack, enforces the Static Assignment Form (SSA) and preserves information about gas consumption. We will see in the next section how such characteristics ease the translation of TEZLA program into their Control Flow Graph (CFG) forms and the construction of data-flow equations.

Compiled languages (like Albert, LIGO, SmartPy, Lorentz, etc.) also provide a higher-level abstraction over Michelson. However, as it happens with most compiled languages, the produced code may not be as concise or compact as expected which, in the case of smart contracts, may result in undesired costs. TEZLA was designed to have a tight integration with the Michelson code to be executed, not as a language that compiles to it nor a higher level language that ease the writing of MICHELSON smart contracts.

In the TEZLA representation, push-like instructions are translated into variable assignments, whereas instructions that consume stack values are transformed to expressions that use as arguments the variables that match the values from the stack. Furthermore, lists, sets and maps deconstruct and lifting of `option` and `or` types that happen implicitly are represented through explicit expressions added to TEZLA.

Since the operational effect of stack manipulation is transposed into variable assignments, we also expose in a TEZLA represented contract the stack manipulation as instructions that act as no-op instructions in the case of a semantics that do not take resource consumption into account[1]. In the case of a resource aware semantics, these instructions will semantically encode this consumption.

The following section describes in detail the process of transforming a MICHELSON smart contract to a TEZLA representation.

2.1 Push-like instructions and stack values consumption

Instructions that push N values to the stack are translated to N variable assignments of those values. The translation process maintains a MICHELSON program stack that associates each stack position to the variable to which that position value was assigned to. When a stack element is consumed, the corresponding variable is used to represent the value. A very simple example is provided in fig. 1.

The block on figure 1a is translated to the TEZLA representation shown in figure 1b.

From the previous example, we can also observe that MICHELSON instructions that consume N stack variables are translated to an expression that consumes those N values. Concretely,

[1]This is the case of the semantics presented in this paper.

```
PUSH nat 5;
PUSH nat 6;
ADD;
```

(a) MICHELSON code.

```
v1 := PUSH nat 5;
v2 := PUSH nat 6;
v3 := ADD v1 v2;
```

(b) TEZLA code.

Figure 1: Stack manipulation example.

the instruction `ADD` that consumes two values (say, `a` and `b`), from the stack is translated to `ADD a b`.

2.2 Branching

MICHELSON provides developers with branching structures that act on different conditions. As TEZLA aims at being used as an intermediate representation for static analysis, there are some properties we would like to maintain. One such property is static single assignment form (SSA-form) [18]. This is guaranteed as TEZLA-represented smart contracts are, by construction, in SSA-form, since each assignment uses new variables.

In order to deal with branching, the TEZLA representation makes use of ϕ-functions (see [18]) that select between two values depending on the branch. As an illustration consider the MICHELSON example in figure 2a.

```
parameter int ;
storage (list int) ;
code { UNPAIR ;
       SWAP ;
       IF_CONS
         { DUP ; DIP { CONS ; SWAP } ;
           ADD ; CONS }
         { NIL int ; SWAP ; CONS } ;
       NIL operation ;
       PAIR }
```

(a) MICHELSON code.

```
v0 := CAR parameter_storage;
v1 := CDR parameter_storage;
SWAP;
IF_CONS v1
{
  v2 := hd v1;
  v3 := tl v1;
  v4 := DUP v2;
  v5 := CONS v2 v3;
  SWAP;
  v6 := ADD v4 v0;
  v7 := CONS v6 v5
}
{
  v8 := NIL int;
  SWAP;
  v9 := CONS v0 v8
};
v10 := φ(v7, v9);
v11 := NIL operation;
v12 := PAIR v11 v10;
```

(b) TEZLA code.

Figure 2: Branching example.

This contract takes an `int` as parameter and a list of `int`s as storage and inserts the sum of the parameter with the head of the list at the lists's head. If the list is empty, it inserts the parameter into the empty list. Here, each branch of the `IF_CONS` instruction will result in a stack with a list of integers, whose values depends on which branch was executed.

This translates to the TEZLA representation presented figure 2b.

The variable `v10` will receive its value through a ϕ-function that returns the value of `v7` if the true branch is executed, or the value of `v9` otherwise.

The `IF_CONS` instruction deconstructs a list in the true branch, putting the head and the tail of the list on top of the stack. From this example, it is possible to observe that the deconstruction of a list is explicit through two variable assignments. This is also the behaviour of `IF_NONE` and `IF_LEFT` instructions, where the unlifting of `option` and `or` types happens explicitly through an assignment.

2.3 Loops, maps and iterations

MICHELSON also provides language constructs for looping and iteration over the elements of lists, sets and maps. These are treated using the same ϕ-functions mechanism in order to preserve SSA-form. We can observe this on the example fig. 3.

```
PUSH nat 0 ;
LEFT nat ;
LOOP_LEFT
 { DUP ;
   PUSH nat 100 ;
   COMPARE ;
   GE ;
   IF
    { PUSH nat 1 ;
      ADD ; LEFT nat }
    { RIGHT nat } } ;
INT ;
```

(a) MICHELSON code.

```
v0 := PUSH nat 0;
v1 := LEFT nat v0;
LOOP_LEFT v2 := ϕ(v1, v12)
{
  v3 := unlift_or v2;
  v4 := DUP v3;
  v5 := PUSH nat 100;
  v6 := COMPARE v5 v4;
  v7 := GE v6;
  IF v7
  {
    v8 := PUSH nat 1;
    v9 := ADD v8 v3;
    v10 := LEFT nat v9;
  }
  {
    v11 := RIGHT nat v3;
  }
  v12 := ϕ(v10, v11);
}
v13 := unlift_or v2;
v14 := INT v13;
```

(b) TEZLA code.

Figure 3: Loop example.

This example uses a `LOOP_LEFT` (loop with an accumulator) to sum 1 to a `nat` (starting with the value 0) until that value becomes greater than 100 and casts the result to an `int`. This example translates to the code presented in fig 3b.

Note that the `LOOP_LEFT` variable is assigned to the value of `v1` if it is the first time that the loop condition is checked, or `v12` if the program flow comes from the loop body. Also notice that the same explicit deconstruction of an `or` variable is applied here, where `v5` gets assigned the value of the unlifting of the loop variable in the beginning of the loop body and at the end of the loop. Similar behaviour applies to the other looping and iteration instructions.

2.4 Full example

We now present a full example of a complete Michelson smart contract (figure 4a).

```
parameter (list bool) ;
storage (pair bool (pair nat int)) ;
code { DUP ;
       CAR ;
       DIP { CDR } ;
       DIP { DUP ; CAR ; DIP { CDR } ;
             DIP { DUP ; CAR ;
                   DIP { CDR } } } ;
       ITER { AND ;
              DUP ;
              IF
               { DIP 2
                     { PUSH int 1 ;
                       ADD } }
               { DIP 2
                     { PUSH int -1 ;
                       ADD } } } ;
       DIP { PAIR } ;
       PAIR ;
       NIL operation ;
       PAIR }
```

(a) Michelson code.

```
v0 := DUP parameter_storage;
v1 := CAR v0;
v2 := CDR parameter_storage;
v3 := DUP v2;
v4 := CAR v3;
v5 := CDR v2;
v6 := DUP v5;
v7 := CAR v6;
v8 := CDR v5;
ITER v9 := ϕ(v1, v18)
{
    v10 := hd v9;
    v11 := AND v10 v4;
    v12 := DUP v11;
    IF v12
    {
        v13 := PUSH int 1;
        v14 := ADD v13 v8;
    }
    {
        v15 := PUSH int -1;
        v16 := ADD v15 v8;
    };
    v17 := ϕ(v14, v16);
    v18 := tl v9;
};
v19 := PAIR v7 v17;
v20 := PAIR v11 v19;
v21 := NIL operation;
v22 := PAIR v21 v20;
return v22;
```

(b) Tezla code.

Figure 4: Example contract.

The contract takes a list of `bool`s as parameter and iterates over that list, It performs a boolean `AND` between an element of the list and the previous `AND` (the initial value of this accumulator is the `bool` on the storage). Depending on the result it either adds 1 ot -1 to the `int` on the storage. The values to be stored are the last `AND` result, the `nat` that was previously on the storage (notice that this value isn't changed nor it is used anywhere else in the program) and the resulting `int` from the sums on the iteration. This contract translates to the Tezla code of fig. 4b.

In this complete example we can observe that a Michelson contract has a parameter and storage. The initial stack of any Michelson smart contract is a stack that contains a single pair whose first element is the input parameter and second element is the contract storage. As such, we introduce a variable called `parameter_storage` that contains the value of that pair.

The final stack of any Michelson smart contract is also a stack that contains a single pair whose first element is a list of internal operations that it wants to emit and whose second element is the resulting storage of the smart contract. We identify the variable containing this pair through the addition of a `return` instruction.

3 Building statics analyses for Tezla smart contracts

In this section, we present the experiments conducted in order to test and demonstrate the applicability of the Tezla intermediate representation to perform static analysis.

3.1 SoftCheck

We build and organise these static analyses upon a generic data-flow analysis platform called SoftCheck [16]. SoftCheck provides an internal and intermediate program representation, called SCIL, rich enough to express high-level as well as low-level imperative programming constructs and simple enough to be adequately translated into CFGs.

SoftCheck is organised upon a generic monotone framework [13] that is able to extract a set of data-flow equations from (1) a suitable representation of programs and; (2) a set of monotone functions; and then to solve them. SoftCheck is written in OCaml and makes use of functor interfaces to leverage its genericity (see fig 5).

By generic we mean that, given a translation from a programming language to SCIL. SoftCheck gives the ability to instantiate its underlying monotone framework by means of a functor interface. Then all defined static analyses are automatically available for the given programming language.

On the other hand, once written as a set of properties and monotone functions, a particular static analysis can be incorporated (again, through instantiating a functor) as an available static analysis for all interfaced programming languages.

SoftCheck offers several standard data-flow analysis such as very busy expressions, available expressions, tainted analysis etc.

We propose in the next sections to detail how we have interfaced Tezla with SCIL, how we have designed a simple but useful data-flow analysis within SoftCheck and how we have tested this analysis on the Michelson smart contracts running in the Tezos blockchain.

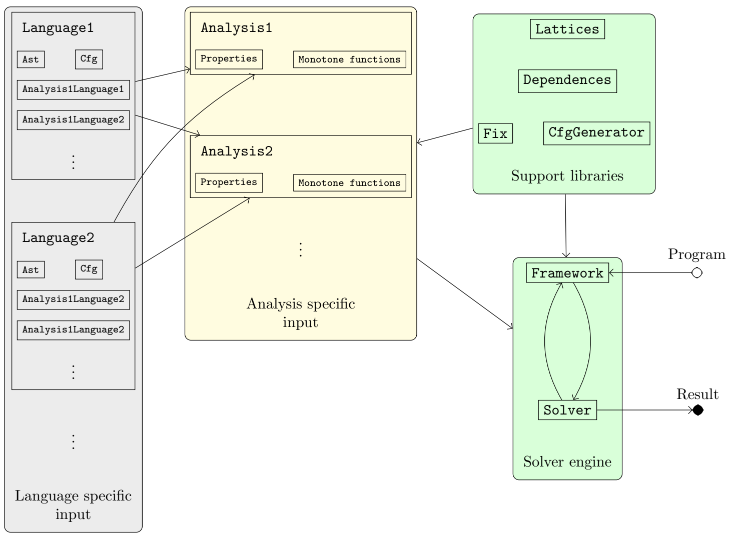


Figure 5: SoftCheck in a picture

3.2 Constructing a Tezla Representation of a Contract

To obtain the TEZLA representation of a smart contract, we first developed a parser to obtain an abstract syntax representation of a MICHELSON smart contract. This parser was implemented in OCaml and Menhir and respects the syntax described in the Tezos documentation [2]. It allows us to obtain a data type that fully abstracts the syntax (with the exception of annotations). To improve the integration between these two forms, TEZLA data types were built upon the data types of MICHELSON.

Control-flow graphs are a common representation among static analysis tools. We provide a library for automatic extraction of such representation from any TEZLA-represented smart contract. This library is based upon the control-flow generation template present on SOFTCHECK. As such, control-flow graphs generated with this library can be used with SOFTCHECK without further work. To instantiate the control-flow graph generation template, we simply provided the library with a module with functions that describe how control flows between each node.

3.3 Sign detection: an example analysis

Here we devise an example of a static analysis for sign detection. The abstract domain consists of the following abstract sign values:: 0 (zero), 1 (one), 0+ (zero or positive), 0- (zero or negative), $\top$ (don't know) and $\perp$ (not a number). These values are organised according to the lattice on figure 6.

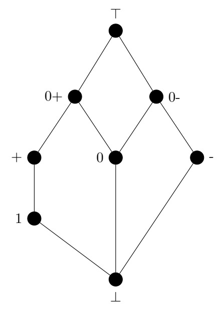


Figure 6: Sign lattice.

Using SOFTCHECK, we implemented a simple sign detection analysis of numerical values. By definition, `nats` have a lowest precision value of 0+, while `ints` can have any value. Every other data type has a sign value of $\perp$.

This implementation does not propagate information to non-simple types (`pair`, `or`, etc.), but it does perform some precision refinements on branching.

To implement such an analysis, we provided SOFTCHECK, in addition to the previously defined TEZLA control-flow graph library, a module that defines how each instruction impacts the sign value of a variable. Then, using the integrated solver mechanism based on the monotone framework, we are able to run this analysis on any TEZLA represented smart contract.

We now present an example. Figure 7 shows the code of a smart contract and its TEZLA representation. This contract multiplies its parameter by -5 if the parameter is equal to 0, or by -2 otherwise, and stores the result in the storage. Figure 8 shows the control-flow graph of representation of that contract.

Running this analysis on the previously mentioned contract produced the results available in Figure 9. In these results we can observe the known sign value of each variable at the exit of each

```
parameter nat ;
storage int ;
code { CAR ;
       DUP ;
       PUSH nat 0 ;
       COMPARE ;
       EQ ;
       IF { PUSH int -5 ; MUL }
          { PUSH int -2 ; MUL } ;
       NIL operation ;
       PAIR }
```

(a) MICHELSON code.

```
v0 := CAR parameter_storage;
v1 := DUP v0;
v2 := PUSH nat 0;
v3 := COMPARE v2 v1;
v4 := EQ v3;
IF v4
{
    v5 := PUSH int -5;
    v6 := MUL v5 v0
}
{
    v7 := PUSH int -2;
    v8 := MUL v7 v0
};
v9 := phi(v6, v8);
v10 := NIL operation;
```

(b) TEZLA code.

Figure 7: Example contract for sign analysis.

block of the control-flow graph in Figure 8. For brevity purposes, we omitted non-numerical variables from the result.

It it possible to observe from the results that the analysis takes into account several details. For instance, the sign of values of type `nat` are, by definition, always zero or positive. The analysis also refines the sign values on conditional branches according to the test. In this case, we can notice that in blocks 6 and 7 (true branch) the sign value of `v1` must be `0`, as the test corresponds to `0 == v1`. Complementary to this, in blocks 8 and 9 the value of `v1` assumes the sign value of `+`, given that being a `nat` value its value must be `0+` and we know that its values is not zero because the test `0 == v1` failed.

We can also conclude from the result of this analysis that the block 17 (true branch) will never be carried out, as the test of that conditional (`0 < v11`) will always be false because the sign of `v11` is `0-`, which means it will always be less than 0.

Due to the TEZLA nature, we were able to take advantage of existing of tooling, such as the SOFTCHECK platform, and effortlessly design the run a data-flow analysis. This enables and eases the development of static analysis that can be used to verify smart contracts but also to perform code optimisations, such as dead code elimination. Albeit simple, the sign analysis can be used to instrument such dead code elimination procedure.

3.4 Experimental Results and Benchmarking

TEZLA and all the tooling are implemented in OCaml and are available under [1]. TEZLA accepts Michelson contracts that are valid according to the Tezos protocol 006 Carthage. We conducted Experimental evaluations that consisted in transforming to TEZLA and running the developed analyses on a batch of smart contracts.

In order to so, we implemented a tool that allows the extraction of smart contracts available in the Tezos blockchain. With that tool, we extracted 142 unique smart contracts. We tested these unique contracts alongside 21 smart contracts we have implemented ourselves.

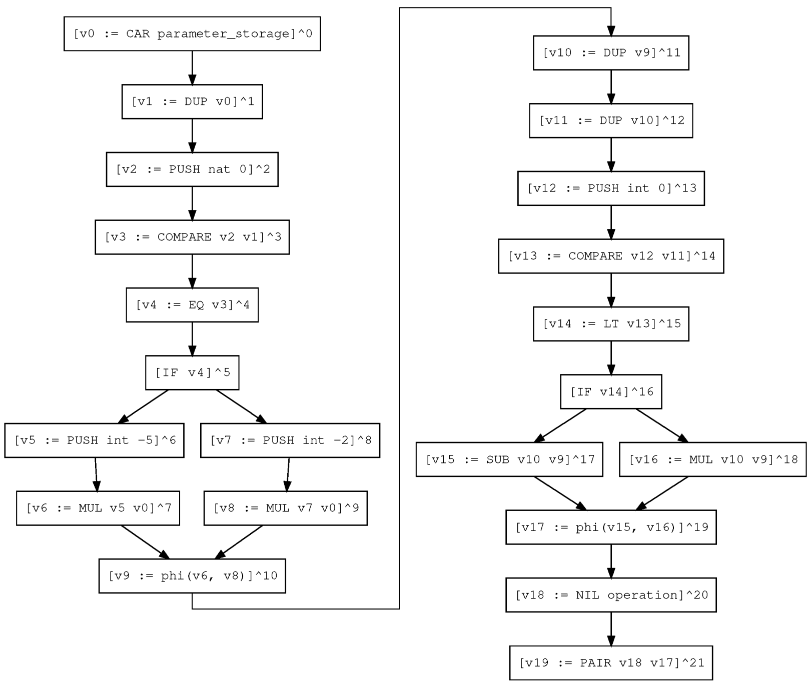


Figure 8: Generated CFG, by the SoftCheck tool

We successfully converted all smart contracts with a coverage result of all Michelson instructions except for 9 instruction that were not used in any of these 163 contracts. On those, we ran the available analyses and obtained the benchmarks presented on table 1. These experiments were performed on a machine with an Intel i7–8750H (2.2 GHz) with 6 cores and 32 GB of RAM.

In the absence of an optimisation tool that takes advantages of the information computed by the analysis, the reports produced by the analysis need to be manually inspected. These reports, the source code of contracts under evaluation, as well as the respective analysis result and other performed static analyses are available at [3, 17].

Average time	0.48 s	Worst-case (number of instructions)	2231 (6.08 s)
Worst-case (time)	9.87 s (926 instructions)	Average time per instrucion	0.0009

Table 1: Benchmark results.

```
0: {
     v0: 0+
}
1: {
     v0: 0+,
     v1: 0+
}
2: {
     v0: 0+,
     v1: 0+,
     v2: 0
}
3: {
     v0: 0+,
     v1: 0+,
     v2: 0,
     v3: 0−
}
4: {
     v0: 0+,
     v1: 0+,
     v2: 0,
     v3: 0−
}
5: {
     v0: 0+,
     v1: 0+,
     v2: 0,
     v3: 0−
}
6: {
     v0: 0,
     v1: 0,
     v2: 0,
     v3: 0−,
     v5: −
}
7: {
     v0: 0,
     v1: 0,
     v2: 0,
     v3: 0−,
     v5: −,
     v6: 0
}
8: {
     v0: +,
     v1: +,
     v2: 0,
     v3: 0−,
     v7: −
}
9: {
     v0: +,
     v1: +,
     v2: 0,
     v3: 0−,
     v7: −,
     v8: −
     }
10: {
     v0: 0+,
     v1: 0+,
     v2: 0,
     v3: 0−,
     v5: −,
     v6: 0,
     v7: −,
     v8: −,
     v9: 0−
}
11: {
     v0: 0+,
     v1: 0+,
     v2: 0,
     v3: 0−,
     v5: −,
     v6: 0,
     v7: −,
     v8: −,
     v9: 0−,
     v10: 0−
}
12: {
     v0: 0+,
     v1: 0+,
     v2: 0,
     v3: 0−,
     v5: −,
     v6: 0,
     v7: −,
     v8: −,
     v9: 0−,
     v10: 0−,
     v11: 0−
}
13: {
     v0: 0+,
     v1: 0+,
     v2: 0,
     v3: 0−,
     v5: −,
     v6: 0,
     v7: −,
     v8: −,
     v9: 0−,
     v10: 0−,
     v11: 0−,
     v12: 0
}
14: {
     v0: 0+,
     v1: 0+,
     v2: 0,
     v3: 0−,
     v5: −,
     v6: 0,
     v7: −,
     v8: −,
     v9: 0−,
     v10: 0−,
     v11: 0−,
     v12: 0,
     v13: 0+
     }
15: {
     v0: 0+,
     v1: 0+,
     v2: 0,
     v3: 0−,
     v5: −,
     v6: 0,
     v7: −,
     v8: −,
     v9: 0−,
     v10: 0−,
     v11: 0−,
     v12: 0,
     v13: 0+
}
16: {
     v0: 0+,
     v1: 0+,
     v2: 0,
     v3: 0−,
     v5: −,
     v6: 0,
     v7: −,
     v8: −,
     v9: 0−,
     v10: 0−,
     v11: 0−,
     v12: 0,
     v13: 0+
}
17: {
     v0: 0+,
     v1: 0+,
     v2: 0,
     v3: 0−,
     v5: −,
     v6: 0,
     v7: −,
     v8: −,
     v9: +,
     v10: +,
     v11: +,
     v12: 0,
     v13: 0+,
     v15: top
}
18: {
     v0: 0+,
     v1: 0+,
     v2: 0,
     v3: 0−,
     v5: −,
     v6: 0,
     v7: −,
     v8: −,
     v9: 0−,
     v10: 0−,
     v11: 0−,
     v12: 0,
     v13: 0+,
     v16: 0+
}
19: {
     v0: 0+,
     v1: 0+,
     v2: 0,
     v3: 0−,
     v5: −,
     v6: 0,
     v7: −,
     v8: −,
     v9: 0−,
     v10: 0−,
     v11: 0−,
     v12: 0,
     v13: 0+,
     v15: top,
     v16: 0+,
     v17: top
}
```

Figure 9: generated report for the sign analysis

4 Related Work

Albert [8] is an intermediate language for the development of Michelson smart contracts. This language provides an high-level abstraction of the stack and some of the language datatypes. This language can be compiled to Michelson through a compiler written in Coq that targets Mi-Cho-Coq [7], a Coq specification of the Michelson language.

Several high-level languages [4, 5, 14, 9, 21] that target Michelson have been developed. Each one presents a different mechanism that abstracts the low-level stack usage. However, a program analysis tool that would target one of these languages should not be easily reusable to programs written in the other languages.

Scilla [19, 20] is an intermediate language that aims to be a translation target of high-level languages for smart contract development. It introduces a communicating automata-based computational model that separates the communication and programming aspects of a contract. The purpose of this language is to serve as a basis representation for program analysis and verification of smart contracts.

Slither [11], presented in 2019, is a static analysis framework for Ethereum smart contract. It uses the Solidity smart contract compiler generated Abstract Syntax Tree to transform the contract into an intermediate representation called SlithIR. This representation also uses a SSA form and reduced instruction in order to facilitate the implementation of program analyses of smart contracts. However Slither has no formal semantics and also the representation is not able to accurately model some low level information like gas computations.

Solidifier [6] is a bounded model checker for Ethereum smart contracts that converts the original source code to Solid, a formalisation of Solidity that runs on its own execution environment. Solid is translated to Boogie, an intermediate verification language that is used by the bounded model checker Corral, which it then used to look for semantic-property violations.

Durieux et. al [10] presented a review on static analysis tools for Ethereum smart contracts. This work presents an extensive list of 35 tools, of which 9 respected their inclusion criteria and were used to test several vulnerabilities on a sample set of 47,587 smart contracts.

5 Conclusion

To the best of our knowledge, this is the first work towards a static analysis framework for Tezos smart contracts. TEZLA positions itself as an intermediate representation obtained from a Michelson smart contract, the low-level language of Tezos smart contracts. This representation abstracts the stack usage through the usage of a store, easing the adoption of mechanism and frameworks for program analysis that assume this characteristic, while maintaining the original semantics of the smart contract.

We have presented a case study on how this intermediate representation can be used to implement a static analysis by using TEZLA along side the SOFTCHECK platform. This has shown how effortlessly one can perform static analysis on Michelson code without forcing developers to use a different language or implement `ad-hoc` static analysis tooling for a stack based language.

Michelson smart contracts have a mechanism of contract level polymorphism called entrypoints, where a contract can be called with an entrypoint name and an argument. This mechanism takes the form of a parameter composed as nesting of `or` types with entrypoint name annotations. This parameter is then checked at the top of contract in a nesting of `IF_LEFT` instructions, running the desired entrypoint this way. This mechanism is optional and transparent to smart contracts without entrypoints. As such, they are also transparent to TEZLA. We therefore plan to extend TEZLA in order to deal with entrypoints and generate isolated components for each entrypoint of a smart contract, which allow us to obtain clearer control-flow graphs and analysis results.

Future plans include a formal account of the TEZLA resource analysis in order to formally verify that the semantics (including gas consumption) of a TEZLA-represented contract are maintained in respect to the original Michelson code. This will also make way to the development of a platform for principled static analysis of Michelson smart contracts. We plan to study which properties are of interest so that we can integrate existing tools and algorithms for code optimization, resource usage analysis and security and correctness verification.

Another direction to tackle is the interfacing of TEZLA with other static analysis platforms such as those provided by the MOPSA project [15] which, among other abilities, provides a means to integrate static analyses.

References

[1] FRESCO - formal verification and static analysis of tezos compliant smart contracts, gitlab. URL: https://gitlab.com/releaselab/fresco.

[2] Michelson: The language of Smart Contracts in Tezos. URL: http://tezos.gitlab.io/whitedoc/michelson.html.

[3] TezCheck - softcheck interface for tezos, gitlab. URL: https://gitlab.com/releaselab/fresco/tezcheck.

[4] G. Alfour. LIGO. URL: https://ligolang.org/.

[5] S. Andrews and R. Ayotte. Fi - Smart coding for smart contracts. URL: https://fi-code.com/.

[6] P. Antonino and A. W. Roscoe. Formalising and verifying smart contracts with Solidifier: A bounded model checker for Solidity. Feb. 2020. arXiv:2002.02710.

[7] B. Bernardo, R. Cauderlier, Z. Hu, B. Pesin, and J. Tesson. Mi-Cho-Coq, a framework for certifying Tezos Smart Contracts. In *1st Workshop on Formal Methods for Blockchains*, Sept. 2019. arXiv:1909.08671.

[8] B. Bernardo, R. Cauderlier, B. Pesin, and J. Tesson. Albert, an intermediate smart-contract language for the Tezos blockchain. In *4th Workshop on Trusted Smart Contracts*, Jan. 2020. arXiv:2001.02630.

[9] DaiLambda. SCaml. URL: https://gitlab.com/dailambda/scaml.

[10] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz. Empirical Review of Automated Analysis Tools on 47,587 Ethereum Smart Contracts. In *42nd International Conference on Software Engineering (ICSE '20)*, Feb. 2020. arXiv:1910.10601, doi:10.1145/3377811.3380364.

[11] J. Feist, G. Greico, and A. Groce. Slither: A static analysis framework for smart contracts. In *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*, WETSEB '19, pages 8–15, Montreal, Quebec, Canada, May 2019. IEEE Press. doi:10.1109/wetseb.2019.00008.

[12] L. M. Goodman. Tezos-a self-amending crypto-ledger White paper. Technical report, 2014.

[13] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–317, 1977. doi:10.1007/BF00290339.

[14] F. Maurel and S. C. Arena. SmartPy. URL: https://smartpy.io/.

[15] A. Miné, A. Ouadjaout, and M. Journault. Design of a modular platform for static analysis. In *Proc. of 9h Workshop on Tools for Automatic Program Analysis (TAPAS'18)*, Lecture Notes in Computer Science (LNCS), page 4, 28 Aug. 2018. http://www-apr.lip6.fr/~mine/publi/mine-al-tapas18.pdf.

[16] J. Reis. Softcheck, a generic and modular data-flow analyser. URL: https://gitlab.com/joaosreis/softcheck.

[17] J. Reis. TezCheck Analysis Results - GitLab. URL: https://gitlab.com/releaselab/fresco/tezcheck-results.

[18] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '88*, pages 12–27, San Diego, California, United States, 1988. ACM Press. doi:10.1145/73560.73562.

[19] I. Sergey, A. Kumar, and A. Hobor. Scilla: A Smart Contract Intermediate-Level LAnguage. 2018. arXiv:1801.00687.

[20] I. Sergey, V. Nagaraj, J. Johannsen, A. Kumar, A. Trunov, and K. C. G. Hao. Safer smart contract programming with Scilla. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–30, Oct. 2019. doi:10.1145/3360611.

[21] Serokell and T. Group. Lorentz. URL: https://gitlab.com/morley-framework/morley/-/tree/master/code/lorentz.

[22] N. Szabo. Formalizing and Securing Relationships on Public Networks. *First Monday*, 2(9), Sept. 1997. doi:10.5210/fm.v2i9.548.

Populating the Peephole Optimizer of a Smart Contract Compiler

Maria A Schett[1] and Julian Nagele[2]

[1] University College London, UK, mail@maria-a-schett.net
[2] London, UK, mail@jnagele.net

Abstract

Developing compiler optimizations, especially for new, rapidly evolving smart contract languages, can be onerous and error-prone, but is especially important for smart contracts, where deployment and execution directly translate to monetary cost and which cannot change once deployed. One common optimization technique is the use of *peephole optimizations*, replacement rules that are applied using pattern-matching. These rules are normally constructed using human expertise, which is both time-consuming and far from systematic in exploring opportunities for optimization. In this work we propose a pipeline to automatically populate the peephole optimizer of a smart contract compiler. We apply superoptimization to an existing code base to obtain sequences of instructions, which can be replaced by cheaper, observationally equivalent instructions. We then generate peephole optimization rules by extracting the underlying patterns of these optimizations. We provide a case study of our approach and a prototype implementation for bytecode of the Ethereum Virtual Machine, the tool pppltr, which combines the superoptimizer ebso and the rule generator sorg. Then we evaluate our approach by generating and applying nearly $1k$ peephole optimization rules extracted from $2k$ optimizations obtained from deployed bytecode.

1 Introduction

In this work we leverage formal methods to automatically populate the peephole optimizer of a smart contract compiler. A *peephole optimizer* uses pattern matching to optimize a small fragment of code, *i.e.*, a peephole, by applying *optimization rules*. But finding sound optimization rules is a bottleneck as witnessed by the peephole optimizer of the Solidity compiler solc.[1] Currently, solc features fewer than 20 rules compared to LLVM's 1000+ rules. Thus we propose a pipeline to automatically populate the peephole optimizer of a smart contract compiler by combining techniques from constraint solving and rewriting as illustrated in Figure 1.

Smart contract languages typically have a large and accessible code base to use as a basis for finding optimizations, *e.g.*, code deployed to public blockchains or test cases. This allows us to start from an existing code base, to (1) *find optimizations* by using automated tools to synthesize observationally equivalent but cheaper instruction sequences. This automatic synthesis is possible, because many smart contract languages come with formally defined operational semantics, *e.g.*, the Ethereum yellow paper [14]. Moreover, execution of a smart contract comes with a clear cost model—gas—giving rise to a precise notion of optimality. To give an example, the bytecode for the Ethereum virtual machine `PUSH 0 SUB PUSH 3 ADD SHA3` computes a hash of $3 + (0 - x)$ for some input x. As $3 + (0 - x) = 3 - x$ the bytecode corresponding to `PUSH 3 SUB SHA3`, computes the same result *and* cheaper. From such optimizations, we can (2) *generate rules*. Using concepts from rewriting we generalize "unnecessarily specific"

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

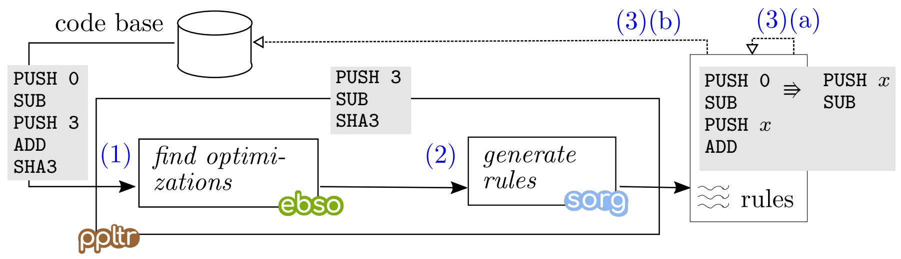


Figure 1: Pipeline to automatically generate peephole optimization rules from a code base.

arguments and strip away "unnecessary" context to obtain optimization rules. For the above example, we generate the rule `PUSH 0 SUB PUSH` x `ADD` $\Rrightarrow$ `PUSH` x `SUB`. Finally we can (3) feed back and apply the generated rules to (a) the rules themselves, or (b) the code base and again start the cycle to find new optimizations.

We demonstrate the applicability of our pipeline in a case study for bytecode of the Ethereum virtual machine (EVM). We implemented a prototype: ppltr, a peephole otimization rule generator. For phase (1), we use the tool ebso [12], a superoptimizer for EVM bytecode. For phase (2), we use sorg, a superoptimization based rule generator. All tools are available open-source under the Apache-2.0 license.[2] We evaluated our approach on bytecode of the 250 most called contracts of the Ethereum blockchain, where we found 2032 distinct optimizations from which we automatically generated 993 optimization rules.

Contributions. We propose *(i)* a pipeline for automatically populating a peephole optimizer, and *(ii)* a sound and complete procedure to generate optimization rules from optimizations. We perform *(iii)* a case study for EVM bytecode with *(iv)* a prototype implementation, together with *(v)* an evaluation.

2 Approach

We assume a machine model with a *state* over a set of *words* $\mathbb{W}$ with an observational equivalence relation $\equiv$ on states, which may take only parts of the state into account. States are modified based on *instructions* from a set $\mathcal{I}$, where an instruction $\iota \in \mathcal{I}$ deterministically transforms a state σ into some state σ' denoted by $\sigma \xrightarrow{\iota} \sigma'$. Some instructions act only on parts of the state, while others take immediate arguments from $\mathbb{W}$. We write $\iota(w_1, \ldots, w_k)$ for an instruction $\iota \in \mathcal{I}$ which takes k immediate arguments $w_1, \ldots, w_k \in \mathbb{W}$ and say that ι has arity k.

A *program* ρ is a sequence of instructions $\iota_0 \cdots \iota_n$. The length of ρ is its number of instructions, denoted by $|\rho|$. We write ε for the *empty* program and $\rho \cdot \tau$ for the concatenation of programs ρ and τ. A program $\rho = \iota_0 \cdots \iota_n$ transforms a family of states $\boldsymbol{\sigma} = (\sigma_j)_{j \in \mathbb{N}}$ by stepwise transformation, *i.e.*, $\sigma_0 \xrightarrow{\iota_0} \sigma_1 \xrightarrow{\iota_1} \ldots \xrightarrow{\iota_n} \sigma_{k+1}$, and we write $\sigma_0 \overset{\rho}{\twoheadrightarrow} \sigma_{n+1}$. Here σ_j is the state after executing j instructions, and σ_0 is the designated start state. We write $\mathsf{cost}(\iota, \sigma)$ for the cost incurred by executing instruction ι on state σ. The cost of executing a program is simply the sum of the cost of its instructions: $\mathsf{cost}(\iota_0 \cdots \iota_n, \boldsymbol{\sigma}) = \sum_{j=0}^{n} \mathsf{cost}(\iota_j, \sigma_j)$. Two programs

[2]Available at github.com/juliannagele/ebso, github.com/mariaschett/sorg, and github.com/mariaschett/ppltr.

ρ and τ are *equal*, denoted by $\rho = \tau$, if they are syntactically equal, and *equivalent*, $\rho \equiv \tau$, if they are observationally equivalent, *i.e.*, for states $\boldsymbol{\sigma}$ and $\boldsymbol{\sigma'}$ with $\sigma_0 \equiv \sigma'_0$, $\sigma_0 \overset{\rho}{\twoheadrightarrow} \sigma_{|\rho|+1}$, and $\sigma'_0 \overset{\tau}{\twoheadrightarrow} \sigma'_{|\tau|+1}$ we have $\sigma_{|\rho|+1} \equiv \sigma'_{|\tau|+1}$.

Definition 2.1. Let ρ and τ be programs with $\rho \equiv \tau$ and $\mathsf{cost}(\rho, \boldsymbol{\sigma}) > \mathsf{cost}(\tau, \boldsymbol{\sigma})$ for all states $\boldsymbol{\sigma}$. Then τ is an *optimization* of ρ, and we write $\rho \gtreqless \tau$.

In Section 2.1, we will show how we can obtain such optimizations—and in Section 2.2 we will use them to generate optimization rules. To do so, we need to define what constitutes a rule. Therefore we abstract over the immediate arguments of instructions by using a countably infinite set of variables $\mathcal{V}$. We extend $\mathcal{I}$ to $\mathcal{I}^{\mathcal{V}}$ by adding instructions $\iota(x_1, \ldots, x_k)$ for all $x_1, \ldots, x_k \in \mathcal{V}$ and all $\iota \in \mathcal{I}$ of arity $k > 0$.

A program over $\mathcal{I}^{\mathcal{V}}$ is called a *program schema*. To obtain a *maximal schema* of a program schema s every $\iota(w_1, \ldots, w_k)$ in s is replaced by $\iota(x_1, \ldots, x_k)$, where $x_1, \ldots, x_k$ are fresh variables from $\mathcal{V}$. All variables in a program schema s are collected in $\mathcal{V}\mathsf{ar}(s)$. A *substitution* $\gamma : \mathcal{V} \to \mathbb{W} \cup \mathcal{V}$ maps variables to variables and words. In a *ground* substitution $\overline{\gamma}$ the range is restricted to $\mathbb{W}$, *i.e.*, $\overline{\gamma} : \mathcal{V} \to \mathbb{W}$. We *apply* γ to a schema s by replacing all variables x in s by $\gamma(x)$ and write $s\gamma$ for the result. Note that $s\overline{\gamma}$ is a program. We call program schemas s and t observationally equivalent if $s\overline{\gamma} \equiv t\overline{\gamma}$ holds for all $\overline{\gamma}$ and write $\mathsf{cost}(s, \boldsymbol{\sigma}) > \mathsf{cost}(t, \boldsymbol{\sigma'})$ if $\mathsf{cost}(s\overline{\gamma}, \boldsymbol{\sigma}) > \mathsf{cost}(t\overline{\gamma}, \boldsymbol{\sigma'})$ for all $\overline{\gamma}$.

Definition 2.2. Let ℓ and r be program schemas with $\ell \equiv r$ and $\mathsf{cost}(\ell, \boldsymbol{\sigma}) > \mathsf{cost}(r, \boldsymbol{\sigma})$. Then $\ell \Rrightarrow r$ is an (optimization) *rule*.

By definition, every optimization $\rho \gtreqless \tau$ is an optimization rule $\rho \Rrightarrow \tau$. A rule $\ell \Rrightarrow r$ is a *generalization* of a rule $\ell' \Rrightarrow r'$ if there is a substitution γ such that $\ell\gamma = \ell'$ and $r\gamma = r'$. A *context* C is a pair of program schemas (s_1, s_2). We write $C[t]$ for the program schema $s_1 \cdot t \cdot s_2$ and call s_1 a *prefix* and s_2 a *postfix* of $C[t]$.

Definition 2.3. The *optimization rules* for an optimization $\rho \gtreqless \tau$ are defined as $\mathcal{R}(\rho \gtreqless \tau) = \{\ell \Rrightarrow r \mid \rho = C[\ell\gamma] \text{ and } \tau = C[r\gamma] \text{ for some } \gamma \text{ and } C\}$.

We ensure that applying peephole optimizations is sound by the following lemma.

Lemma 2.1. *If $\rho \equiv \tau$ then $C[\rho] \equiv C[\tau]$ for all contexts C.*

Proof. We show the statement by induction on C. By assumption, the statement holds for the base case $C = (\varepsilon, \varepsilon)$. For the step case $C = (\iota \cdot s_1, s_2)$ observe that every instruction ι is deterministic, *i.e.*, executing ι starting from a state σ leads to a deterministic state σ'. By induction hypothesis, executing $s_1 \rho s_2$ and $s_1 \tau s_2$ from a state σ' leads to the same state σ'', and therefore $\iota \cdot s_1 \cdot \rho \cdot s_2 \equiv \iota \cdot s_1 \cdot \tau \cdot s_2$ holds. We can reason analogously for $C = (s_1, s_2 \cdot \iota)$. □

2.1 Find Optimizations

As Definition 2.1 suggests finding an optimization for a program ρ necessitates finding *(i)* an observationally equivalent program τ, where *(ii)* the cost of τ is less than the cost of ρ. We leverage a constraint solver such as Z3 [6] to automatically find equivalent, but cheaper programs. To this end, we express the above as an SMT problem: given a source program ρ, $\exists$ a target program τ such that $\forall$ possible inputs, ρ implements τ and the cost of τ is less than the cost of ρ? Our encoding is based on the encoding from unbounded superoptimization [7].

To find an observationally equivalent program. To encode observational equivalence we first need a constraint that expresses equality on states: Let $\mathsf{enc\_eq\_state}(\sigma, \sigma')$ be an SMT constraint that evaluates to true, whenever state σ and state σ' are observationally equivalent. The concrete instantiation of this constraint depends on the machine that is modelled. For instance, the state may be modelled as several uninterpreted functions. An encoding for the EVM, modelling the state with a stack, storage, and exceptional halting can be found in Example 3.1, with the corresponding encoding of $\mathsf{enc\_eq\_state}$ in Example 3.3.

Based on the operational semantics for every $\iota \in \mathcal{I}$, we need to encode the effect of ι on a state σ_j, *i.e.*, $\sigma_j \xrightarrow{\iota} \sigma_{j+1}$.

Definition 2.4. Let $\mathsf{enc\_step}(\iota, \sigma, \sigma')$ be an SMT encoding of the effect of an instruction ι as constraints between state σ and state σ'. For a program $\rho = \iota_0 \cdots \iota_n$ and states $\boldsymbol{\sigma}$ we define $\mathsf{enc\_progr}(\rho, \boldsymbol{\sigma})$ as $\bigwedge_{0 \leqslant j \leqslant n} \mathsf{enc\_step}(\iota_j, \sigma_j, \sigma_{j+1})$.

Again, the concrete encoding of $\mathsf{enc\_step}$ depends on the machine that is modelled, see Example 3.2 for our instantiation for the EVM.

Most programs will consume input $\vec{x}$. To pass them to the program, we assume an encoding $\mathsf{enc\_init}(\vec{x}, \boldsymbol{\sigma})$ that sets constraints on the start state σ_0 appropriately, *e.g.*, putting the words in $\vec{x}$ in registers or on the stack according to the machine model. Based on the constraint $\mathsf{enc\_step}$, we can encode the search space of all possible target programs. To this end we represent the target program as a pair $\tau = \langle \mathsf{instr}, n \rangle$ of an uninterpreted function $\mathsf{instr}(j) : \mathbb{N} \to \mathcal{I}$ and its length $n \in \mathbb{N}$. The function instr acts as a template to be filled by the SMT solver returning the instruction to be used at position j of the target program. After a model has been found, the concrete target program can be reconstructed as $\mathsf{instr}(0) \cdot \mathsf{instr}(1) \cdots \mathsf{instr}(n-1)$.

Definition 2.5. Given a set of instructions $\mathcal{I}$ we define the SMT encoding for the enumeration of every program of length n as $\mathsf{enc\_search}(\tau, \boldsymbol{\sigma})$ as

$$\forall j.\, 0 \leqslant j < n \to \bigwedge_{\iota \in \mathcal{I}} \mathsf{instr}(j) = \iota \to \mathsf{enc\_step}(\iota, \sigma_j, \sigma_{j+1}) \wedge \bigvee_{\iota \in \mathcal{I}} \mathsf{instr}(j) = \iota \tag{1}$$

The first clause states that if we pick ι at position j, then the effect will be as determined by $\mathsf{enc\_step}(\iota, \sigma_j, \sigma_{j+1})$. The second clause, $\bigvee_{\iota \in \mathcal{I}} \mathsf{instr}(j) = \iota$, ascertains that for every position j some instruction is picked.

Definition 2.6. The encoding for finding an observationally equivalent program to a given program ρ is

$$\begin{aligned} &\exists n, \forall \vec{x}.\, \mathsf{enc\_init}(\vec{x}, \boldsymbol{\sigma}) \wedge \mathsf{enc\_init}(\vec{x}, \boldsymbol{\sigma'}) \wedge \\ &\qquad \mathsf{enc\_progr}(\rho, \boldsymbol{\sigma}) \wedge \mathsf{enc\_search}(\tau, \boldsymbol{\sigma'}) \wedge \mathsf{enc\_eq\_state}(\sigma_{|\rho|+1}, \sigma'_n) \end{aligned} \tag{2}$$

The first two constraints initialise states $\boldsymbol{\sigma}$ and $\boldsymbol{\sigma'}$ with the same inputs, the third and fourth constraint encode the effects of the existing program ρ and the sought after target program τ respectively, while the final constraint demands that they are observationally equivalent, *i.e.*, that they result in equivalent states. With this constraint we will find observational equivalent programs. Now we will need to add constraints on the cost.

To find a cheaper program. To achieve this we extend Constraint (2) from Definition 2.6 by a constraint stating that the cost of executing the target program τ is less than the cost of executing the source program ρ: *i.e.*, $\mathsf{cost}(\rho, \boldsymbol{\sigma}) > \mathsf{cost}(\tau, \boldsymbol{\sigma'})$. Here the cost of τ is again defined by summation, *i.e.*, for $\tau = \langle \mathsf{instr}, n \rangle$ we have $\mathsf{cost}(\tau, \boldsymbol{\sigma'}) = \sum_{j=0}^{n-1} \mathsf{cost}(\mathsf{instr}(j), \sigma_j)$.

2.2 Generate Rules

As Definition 2.3 indicates generating optimization rules necessitates *(i)* to find a substitution γ, and *(ii)* to find a context C.

To find a substitution. We generalize the immediate arguments of instructions in an optimization $\rho \gtreqless \tau$ by finding a substitution. Example 3.4 shows generalized rules for EVM bytecode.

Definition 2.7. The *generalized rules* of an optimization rule $\rho \Rrightarrow \tau$ are defined as $\mathcal{G}(\rho \Rrightarrow \tau) = \{\ell \Rrightarrow r \mid \ell\gamma = \rho \text{ and } r\gamma = \tau \text{ for some substitution } \gamma\}$.

$\mathsf{generalize}(\rho \Rrightarrow \tau)$**.** Let ℓ and r be maximal program schemas for ρ and τ with $\mathcal{V}\mathsf{ar}(\ell) \cap \mathcal{V}\mathsf{ar}(r) = \varnothing$. Let γ_0 be the substitution with $\rho = \ell\gamma_0$ and $\tau = r\gamma_0$. We collect all possible substitutions in $\Gamma = \{\gamma \mid \ell\gamma \Rrightarrow r\gamma \text{ and } \gamma(x) = \gamma_0(x) \text{ or } \gamma(x) = y \text{ for } \gamma_0(x) = \gamma_0(y) \text{ and } x, y \in \mathcal{V}\mathsf{ar}(\ell) \cup \mathcal{V}\mathsf{ar}(r)\}$, and say $\gamma_1 < \gamma_2$ if $\ell\gamma_1 \Rrightarrow r\gamma_1$ is more general than $\ell\gamma_2 \Rrightarrow r\gamma_2$. Then we define $\mathsf{generalize}(\rho \Rrightarrow \tau) = \{\ell\gamma \Rrightarrow r\gamma \mid \gamma \in \Gamma \text{ and } \gamma \text{ is minimal } \textit{wrt.} >\}$. The order $>$ is key for implementation, because it allows to prune the search space. Pruning only removes rules covered by others as the following Lemma shows.

Lemma 2.2. *For every $\ell \Rrightarrow r \in \mathcal{G}(\alpha)$ of a rule α there is a $\ell' \Rrightarrow r' \in \mathsf{generalize}(\alpha)$ and a substitution γ such that $\ell'\gamma = \ell$ and $r'\gamma = r$.*

Proof. We fix $\ell \Rrightarrow r \in \mathcal{G}(\alpha)$. By definition there is a substitution γ'' such that $\ell\gamma'' \Rrightarrow r\gamma'' = \alpha$. We compute the maximal schemas ℓ_0 and r_0 of α. By definition of maximal schema we can find a γ' such that $\ell_0\gamma' = \ell$ and $r_0\gamma' = r$. A renaming of γ' is in Γ and by definition $\mathsf{generalize}(\alpha)$ contains a rule $\ell_0\gamma \Rrightarrow r_0\gamma = \beta$ where $\gamma \leqslant \gamma'$ and $\gamma'\gamma = \gamma''$. Hence, β is the desired generalization of $\ell \Rrightarrow r$. ☐

To find a context. We strip the generalized rule of any unnecessary pre- and postfix. Example 3.5 shows rules stripped of their context in EVM bytecode.

Definition 2.8. The *stripped rules* of a rule $\ell \Rrightarrow r$ are defined as $\mathcal{C}(\ell \Rrightarrow r) = \{\ell' \Rrightarrow r' \mid \ell = C[\ell'] \text{ and } r = C[r']\}$.

$\mathsf{strip}(\rho \Rrightarrow \tau)$**.** Let s be the longest common prefix, and t be the longest common postfix of ℓ and r such that $s \cdot \ell \cdot t = \rho$ and $s \cdot r \cdot t = \tau$. We collect all possible contexts in $\Gamma = \{C \mid C[\ell] \Rrightarrow C[r] \text{ for } C = (s_1, t_2) \text{ for some } s_1, t_2 \text{ where } s_1 \cdot s_2 = s \text{ and } t_1 \cdot t_2 = t\}$ and say $C_1 = (s_1, t_1) > (s_2, t_2) = C_2$ if s_1 is longer than s_2 and t_1 is longer than t_2. Then, we define $\mathsf{strip}(\rho \Rrightarrow \tau) = \{C[\ell] \Rrightarrow C[r] \mid C \in \Gamma \text{ and } C \text{ is maximal } \textit{wrt.} >\}$. Again, the order $>$ allows to prune the search space without loss.

Lemma 2.3. *For every $\ell \Rrightarrow r \in \mathcal{C}(\alpha)$ of a rule α there is a $\ell' \Rrightarrow r' \in \mathsf{strip}(\alpha)$ and a context C such that $C[\ell'] = \ell$ and $C[r'] = r$.*

Proof. We fix a $\ell \Rrightarrow r \in \mathcal{C}(\alpha)$. By definition, there is a context $C'' = (s'', t'')$ such that $C''[\ell] \Rrightarrow C''[r] = \alpha$. By definition, $\mathsf{strip}(\alpha)$ contains a rule $\ell' \Rrightarrow r'$ such that $C'[\ell'] \Rrightarrow C'[r'] = \alpha$ for some $C' = (s', t')$ in Γ. By construction of strip, s'' is longer than s' and t'' is longer than t', and thus we can find s and t such that $s' \cdot s = s''$ and $t' \cdot t = t''$. Then $C = (s, t)$ is the desired context with $C[\ell'] = \ell$ and $C[r'] = r$. ☐

$\mathsf{sorg}(\rho \gtreqless \tau)$. For an optimization $\rho \gtreqless \tau$, we define $\mathsf{sorg}(\rho \gtreqless \tau) = \{\mathsf{strip}(\ell \Rrightarrow r) \mid \ell \Rrightarrow r \in \mathsf{generalize}(\rho \Rrightarrow \tau)\}$.

Soundness and completeness. The rules generated by $\mathsf{sorg}(\rho \gtreqless \tau)$ are *sound*: for every $\ell \Rrightarrow r \in \mathsf{sorg}(\rho \Rrightarrow \tau)$ there is a substitution γ and a context C such that $C[\ell\gamma] = \rho$ and $C[r\gamma] = \tau$. This directly follows from $\mathsf{generalize}(\rho \Rrightarrow \tau) \subseteq \mathcal{G}(\rho \Rrightarrow \tau)$ and $\mathsf{strip}(\rho \Rrightarrow \tau) \subseteq \mathcal{C}(\rho \Rrightarrow \tau)$. The rules generated by $\mathsf{sorg}(\rho \gtreqless \tau)$ are also *complete*: for every $\ell \Rrightarrow r \in \mathcal{R}(\rho \gtreqless \tau)$ there is a $\ell' \Rrightarrow r' \in \mathsf{sorg}(\rho \gtreqless \tau)$, a substitution γ and a context C such that $C[\ell'\gamma] = \ell$ and $C[r'\gamma] = r$. This directly follows by Lemma 2.2, and Lemma 2.3.

3 Case Study: EVM bytecode

We apply our pipeline in Figure 1 in the context of Ethereum for EVM bytecode. The EVM is a virtual machine formally defined in the Ethereum yellow paper [14]. It is based on a stack which holds bit vectors of size 256. The stack may over- or underflow; both lead the EVM to enter an exceptional halting state. The EVM also features a volatile memory, a word-addressed byte array, and a persistent key-value storage, a word-addressed word array, whose contents are stored on the Ethereum blockchain.

3.1 Find Optimizations with ebso

We find optimizations using our tool ebso [12], an EVM bytecode superoptimizer.[3] As an input ebso takes an ebso block—a basic block that additionally does not contain instructions whose semantics are not encoded, such as instructions that have an outside effect like `LOG`. Then, encoding the EVM execution state and unbounded superoptimization following Section 2.1, in the best case ebso produces a cheaper, observationally equivalent ebso block.

Example 3.1. We encode the EVM execution state $\boldsymbol{\sigma}$ using four uninterpreted functions $\langle \mathsf{sk}, \mathsf{c}, \mathsf{hlt}, \mathsf{str}\rangle$ to model the stack, stack pointer, exceptional halting and storage: *(i)* $\mathsf{sk}(j, \vec{x}, n)$ returns the word from position n in the stack after executing j instructions on $\vec{x}$, *(ii)* $\mathsf{c}(j)$ returns the number of words on the stack after executing j instructions, *(iii)* $\mathsf{hlt}(j)$ returns true $(\top)$ if exceptional halting has occurred after executing j instructions, and false $(\bot)$ otherwise, and *(iv)* $\mathsf{str}(j, \vec{x}, k)$ returns the word at key k after executing j instructions on $\vec{x}$. Note that these functions represent all states throughout an execution, *i.e.*, $\boldsymbol{\sigma}$, while to obtain σ_j for some j, we simply apply them to j thus: $\sigma_j = \langle \mathsf{sk}(j), \mathsf{c}(j), \mathsf{hlt}(j), \mathsf{str}(j)\rangle$.

For a program ρ which takes d arguments on the stack we add d fresh variables to represent the input $\vec{x}$ and add the following constraint to $\mathsf{enc\_init}(\vec{x}, \boldsymbol{\sigma})$:

$$\bigwedge_{0 \leqslant i < d} \mathsf{sk}_\sigma(\vec{x}, 0, i) = x_i \wedge \mathsf{c}_\sigma(0) = d \wedge \mathsf{hlt}_\sigma(0) = \bot$$

The storage str is initialised similarly using an Ackermann encoding.

Example 3.2. Next we instantiate the operational semantics of the instructions. The constraint $\mathsf{enc\_stack}(\iota, \sigma_j, \sigma_{j+1})$ describes effect that ι has on stack. Here we give as example the instruction `SUB` and refer to [14] or [12] for details. Let $-_{\mathsf{bv}}$ denote subtraction on bit-vectors.

[3]Available at github.com/juliannagele/ebso.

Then we have

$$\begin{aligned}\mathsf{enc\_stack}(\texttt{SUB}, \sigma_j, \sigma_{j+1}) := \mathsf{sk}_\sigma(\vec{x}, j+1, \mathsf{c}_\sigma(j+1) - 1)\\ = \mathsf{sk}_\sigma(\vec{x}, j, \mathsf{c}_\sigma(j) - 1) -_{\mathsf{bv}} \mathsf{sk}_\sigma(\vec{x}, j, \mathsf{c}_\sigma(j) - 2)\end{aligned}$$

Using enc_stack we can formulate the constraint enc_step. Here $\delta(\iota)$ and $\alpha(\iota)$ refer to the number of words which ι deletes from, and adds to the stack respectively. For all instructions except `SSTORE` we have:

$$\begin{aligned}\mathsf{enc\_step}(\iota, \sigma_j, \sigma_{j+1}) := \ & \mathsf{enc\_stack}(\iota, \sigma, j) \land\\ & \mathsf{c}_\sigma(j+1) = \mathsf{c}_\sigma(j) + \alpha(\iota) - \delta(\iota) \land\\ & \forall\, n.\, n < \mathsf{c}_\sigma(j) - \delta(\iota) \to \mathsf{sk}_\sigma(\vec{x}, j+1, n) = \mathsf{sk}_\sigma(\vec{x}, j, n) \land\\ & \mathsf{hlt}_\sigma(j+1) = \mathsf{hlt}_\sigma(j) \lor \mathsf{c}_\sigma(j) - \delta(\iota) < 0 \lor \mathsf{c}_\sigma(j) - \delta(\iota) + \alpha(\iota) > 2^{10} \land\\ & \forall w.\, \mathsf{str}_\sigma(\vec{x}, j+1, w) = \mathsf{str}_\sigma(\vec{x}, j, w)\end{aligned}$$

Here the second line updates the counter for the number of words on the stack according to the number of words added and deleted. The third line expresses that all words on the stack below $\mathsf{c}_\sigma(j) - \delta(\iota)$ are preserved. The fourth line captures that exceptions relevant to the stack can occur through either an underflow or an overflow, and that once it has occurred, an exceptional halt state persists. Finally the last line states that all $\iota \neq$ `SSTORE` do not change the storage.

Example 3.3. The final ingredient we need to instantiate is the equivalence relation on states. For two states at steps j_1 and j_2 where $\sigma_{j_1} = \langle \mathsf{sk}(j_1), \mathsf{c}(j_1), \mathsf{hlt}(j_1), \mathsf{str}(j_1)\rangle$ and $\sigma'_{j_2} = \langle \mathsf{sk}'(j_2), \mathsf{c}'(j_2), \mathsf{hlt}'(j_2), \mathsf{str}'(j_2)\rangle$ and input $\vec{x}$ we define the constraint $\mathsf{enc\_eq\_state}(\sigma_{j_1}, \sigma'_{j_2})$ as

$$\begin{aligned}&\mathsf{c}(j_1) = \mathsf{c}'(j_2) \land \mathsf{hlt}(j_1) = \mathsf{hlt}'(j_2)\\ &\land \forall\, w.\, \mathsf{str}(j_1, \vec{x}, w) = \mathsf{str}'(j_2, \vec{x}, w)\\ &\land \forall\, n.\, n < \mathsf{c}(j_1) \to \mathsf{sk}(j_1, \vec{x}, n) = \mathsf{sk}'(j_2, \vec{x}, n)\end{aligned}$$

With the presented encoding, ebso, and an SMT solver we can now automatically find optimizations for EVM bytecode. Next, we also want to automatically generate rules.

3.2 Generate Rules with sorg

To generate rules for EVM bytecode we implemented sorg, a superoptimization based rule generator.[4] Like ebso, sorg is implemented in OCaml; sorg depends on ebso for the representation of EVM bytecode and calls to determine equivalence. We encode equivalence with components from ebso similar to Definition 2.6. For two program schemas ρ and τ, we have $\rho \equiv \tau$ if there are no inputs that distinguish them. That is

$$\begin{aligned}&\exists \vec{x}.\, \mathsf{enc\_init}(\vec{x}, \boldsymbol{\sigma}) \land \mathsf{enc\_init}(\vec{x}, \boldsymbol{\sigma'})\\ &\quad \land \mathsf{enc\_progr}(\rho, \boldsymbol{\sigma}) \land \mathsf{enc\_progr}(\tau, \boldsymbol{\sigma'})\\ &\quad \land \neg(\mathsf{enc\_eq\_state}(\sigma_{|\rho|+1}, \sigma'_{|\tau|+1})\end{aligned}$$

The main contribution of sorg is to provide notions of program schema, substitutions, and context in order to implement the two main procedures of Section 2.2: generalize and strip. For generalize, Definition 2.7, we implement the procedure from the proof of Lemma 2.2.

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

Example 3.4. In our evaluation in Section 4, we found the following optimization:

$$\texttt{SWAP1 POP PUSH}\,0\ \texttt{PUSH}\,1\ \texttt{MUL PUSH}\,0 \gtreqless \texttt{SWAP1 POP PUSH}\,0\ \texttt{DUP1}$$

Generalising immediate arguments and dropping the prefix SWAP1 POP yields two optimization rules: PUSH x PUSH 1 MUL PUSH x $\Rrightarrow$ PUSH x DUP1 as well as PUSH 0 PUSH x MUL PUSH 0 => PUSH 0 DUP1.

For strip, Definition 2.8, we implement the procedure from the proof of Lemma 2.3.

Example 3.5. From the rule CALLVALUE DUP1 POP $\Rrightarrow$ CALLVALUE CALLVALUE POP we can either strip the postfix POP or the prefix CALLVALUE, obtaining the rules CALLVALUE DUP1 $\Rrightarrow$ CALLVALUE CALLVALUE and DUP1 POP $\Rrightarrow$ CALLVALUE POP.

With sorg we can now automatically generate rules, but it remains to glue the tools together and implement a feedback mechanism.

3.3 Coordinate with ppltr

To coordinate our tools ebso and sorg we implemented the tool ppltr, a populator for a peephole optimizer.[5] As ebso and sorg, ppltr is implemented in OCaml. The tool has two main tasks. The first is to manage the interfaces, *i.e.*, *(i)* to generate ebso blocks from smart contracts, *(ii)* to generate ebso blocks for a given size k, *(iii)* to prepare optimizations generated by ebso as input for sorg, and *(iv)* to analyse and de-duplicate a set of rules produced by sorg. The second main task is to feed back the optimization rules, *i.e.*, *(v)* to rewrite right-hand sides of the optimization rules themselves, and *(vi)* to apply the optimization rules on ebso blocks. To achieve the latter task, ppltr implements a rewrite engine.

4 Evaluation

We evaluate our pipeline by generating optimization rules for EVM bytecode. We collected the 250 most called smart contracts until block 9 786 000 at Apr-01-2020 12:17:26 PM +UTC from the Ethereum blockchain using Google BigQuery.[6]

We split the 250 contracts into 106 798 ebso blocks $\mathcal{E}$. As peephole optimization rules typically span only few instructions, we restrict the size of a block: using a sliding window we split every block larger than 6 instructions into k blocks of at most 6 instructions. To reduce the noise, we remove blocks which are only different in the arguments of PUSH keeping only those with words of size smaller than 5 bit. We so obtain 54 301 ebso blocks, from which we (1) find 1580 optimizations with ebso, run on a cluster with Intel Xeon Gold 6126 CPUs at 2.60 GHz, 2 GB of memory and a time-out of 15 min. From these optimizations, we (2) generate 1525 rules with sorg, run on the same set-up. For 48 optimizations sorg timed out and could not generate rules and we removed roughly half the rules, as they were duplicates generated from different optimizations. Thus we arrive at 758 rules $\mathcal{R}_0$, which we use with the rewrite engine of ppltr (3) to (a) rewrite the right-hand sides of $\mathcal{R}_0$ reducing 4 rules, and (b) rewrite our original ebso blocks in $\mathcal{E}$. Thereby 17 255 ebso blocks changed and we again use the same window-size and noise reduction to get 25 585 new ebso blocks. Going through the same procedure, we (1) find 452 optimizations with ebso, and (2) generate 435 rules $\mathcal{R}_1$ with sorg with 16 timeouts. Combining the results we get 993 rules $\mathcal{R}_2 = \mathcal{R}_0 \cup \mathcal{R}_1$ which are available at

[5] Available github.com/mariaschett/ppltr.

[6] *cf.*. cloud.google.com/blog/products/data-analytics/ethereum-bigquery-public-dataset-smart-contract-analytics.

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

Table 1: Savings when applying the rules in $\mathcal{R}_2$ on most called contracts.

github.com/mariaschett/ppltr/blob/master/eval/17-reduced-rules.csv

We right-reduced 31 rules in $\mathcal{R}_2$ and discarded 967 replicated rules originating from different optimizations. One optimization generated two rules (*cf.* Example 3.4).

To estimate gas and size saving on a contract level we apply the rules in $\mathcal{R}_2$ to (1) our original 250 most called smart contracts, and (2) extend the data set to the 1000 most called contracts. Table 1 shows our results. The first column shows the accumulated gas savings over all contracts, and the second column shows the accumulated length savings. Note that results depend on the order in which the rules are applied (*cf.* Section 5). First, we can observe that the rules translate well from 250 to 1000 contracts, achieving roughly 4 times higher savings, which demonstrates that $\mathcal{R}_2$ also extends beyond the original data set, from which it was generated.

Now let us consider the gas savings. In Table 1 we accumulate the cost of all the removed instructions for each contract. How much is actually saved, however, depends on how often the contract is called and which parts are executed. Unfortunately we lack the resources to replay all the transactions to determine the exact savings. Taking into account how often a contract was called, we save 7.41×10^{10} g for the former and 1.02×10^{11} g for the latter. Assuming that about 10 % of a contract is executed per call and that savings are uniformly distributed, this translates to 41 049.33 \$ and 56 505.15 \$ for a gas price of 27.6 gwei and an ETH-USD course of 200.62 \$, which are averages from etherscan.io/charts.

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

We also analyse which rules are applied to the contracts. Applying rules may lead to the the applicability of other rules, but exploring all rewrite sequences is intractable, and we assume that initial applicability on a contract is a reasonable proxy. Figure 2 groups rules in $\mathcal{R}_2$ by their applicability to the 1000 most called contracts. We can observe a long tail: more than half of the nearly $1k$ rules are applicable only 10 times or less, whereas the top 50 rules are applicable more than 500 times. This suggests that, if a smaller set of rules is desired, this analysis can guide which rules to discard.

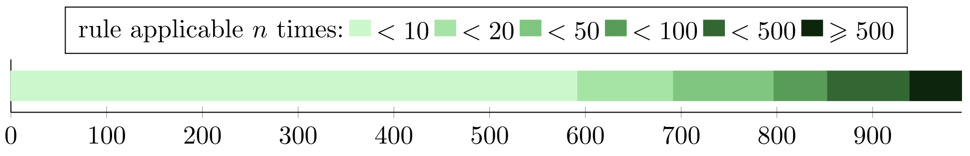


Figure 2: Applicability of rules in $\mathcal{R}_2$ to 1000 most called contracts.

The five most applied rules for the 1000 most called contracts are listed in Figure 3 on the left. Most of these rules are relatively simple and should clearly be applied exhaustively. The fourth rule is perhaps a bit unexpected and may have been missed on manual inspection, but

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

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

Figure 3: Rules most applied to the 1000 most called contracts and saving most gas (right).

it is cheaper to execute `CALLVALUE` twice than duplicating its result. The last rule hints at a specific compiler produced anti-pattern. Our approach could also be leveraged to detect those.

Next we inspect the rules within $\mathcal{R}_2$. The right-hand side of Figure 3 shows the six rules with the highest gas savings, 17 g and 15 g. We consider two of these rules in more detail. The rule `PUSH 1 MUL PUSH 0 NOT AND` $\Rrightarrow$ ε combines two observations—that 1 and `PUSH 0 NOT` are neutral elements for multiplication and `AND` respectively. Depending on the implementation of the peephole optimizer it may be desirable to split this rule which could be achieved by left-reducing the rules. Key to the rule `PUSH 0 DUP6 DUP5 SUB LT ISZERO` $\Rrightarrow$ `PUSH 1` is the less-than comparison `LT` with the smallest element 0 always evaluating to false. The rule does not depend on the result of `DUP6 DUP5 SUB`, and indeed this is replaced by `DUP2 PUSH` x `AND` in the otherwise identical rule in the last line. Generalising those two rules would require the use of higher-order patterns.

Rules may not only save gas, but also reduce the length the produced code. These often coincide, and indeed the top 14 length-reducing rules, removing 5 instructions each, subsume the above gas-saving rules. On the other end, there are also rules which save gas but do not reduce the length such as `CALLVALUE DUP1` $\Rrightarrow$ `CALLVALUE CALLVALUE` saving 1 g. In Table 2, we analyse the right-hand sides of $\mathcal{R}_2$. We investigated which instructions were *added*, *i.e.*, do not appear on the left-hand side, and *removed*, *i.e.*, appear on the left- but not the right-hand side of the rule. We group instructions for arithmetic, comparison, bitwise operations, and environment/memory. Unsurprisingly, many more instructions were removed than added,

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

Table 2: Added and removed instructions by group.

which is expected, because removing instructions always saves gas. The majority of removed instructions is concerned with the stack layout. Surprisingly, also `ISZERO` is often redundant—as also observed in the second rule in Figure 3. Still, instructions are also synthesised on the right-hand side giving rise to optimizations taking the semantic of an instructions into account—potentially also interacting with stack manipulation, for example the rule `SWAP1 LT` $\Rrightarrow$ `GT`.

Finally, we also successfully validated all rules $\mathcal{R}_2$ by running a reference implementation of the EVM, `go-ethereum` version `1.9.14` on pseudo-random input.[7] Therefore, we run the bytecode of every block in $\mathcal{E}$ and the bytecode obtained by applying the rewrite rules to observe that both produce the same final state.

[7] github.com/ethereum/go-ethereum

5 Related and Future Work

Chen *et al.* [5] also developed a tool to rewrite optimization patterns in EVM bytecode. As opposed to our approach, they devised their 24 (anti-)patterns by manual inspection of the code base. Albert *et al.* [1] synthesise optimized straight-line EVM bytecode for operations on the stack with Max-SMT. To gain efficiency, they do not encode the semantics of bit-vector instructions, and instead employ hand-crafted simplification rules. These hand-crafted rules could be inspired by, or even automatically derived from, rules generated by ppltr, which do consider the semantic of bit-vector instructions. Bansal *et al.* [3] use superoptimization to automatically generate a peephole optimizer for x86 binaries. Aside from the application, the main difference of their approach to ppltr is that it does not process optimizations into rules but instead keeps them in an optimization database in order to reapply them. Moreover it uses an enumeration based superoptimizer, which is more exhaustive, but limits instruction sequences to length 3.

We believe our approach is also applicable for different smart contract languages. Facebook's Move [4] is a gas-metered and verification friendly designed language with an existing code base, such as for example from github.com/libra/libra/tree/master/language/move-lang/tests/functional. The machine model of Move is stack-based with typed locals. To adapt the presented approach the SMT encoding would need to be extended to incorporate types and locals. Michelson [10], the smart contract language for the Tezos blockchain, also comes with a detailed formal semantics. Like the EVM it is a stack-based language, but features high-level data types, like lists, sets, and maps. To use the presented approach these data types need to be handled in the SMT encoding and SMT solvers do support complex theories such as sets and lists. Moreover, type information could be used to prune search space, resulting in a positive performance impact.

Our definitions in Section 2.2 are based on concepts from term rewriting [2] and thus we also look at the machinery of term rewriting. Termination of the rules ensures we can apply them exhaustively without looping. Intuitively all rules in $\mathcal{R}_2$ are terminating, since left-hand sides have a higher cost than right-hand sides, and indeed the termination prover WANDA [8] shows termination of all 993 rules in $\mathcal{R}_2$.[8] Confluence guarantees a unique result regardless of how the rules are applied. To check confluence one analyses critical pairs, situations where application of one rule potentially destroys the possibility for applying another one. The confluence checker CSI [11] reports 82 765 critical pairs, 14 973 of which are joinable and thus harmless. The remaining 67 792 are not, so the rules in $\mathcal{R}_2$ are not confluent. This is not surprising, since there are different ways to achieve the same with the same cost, *e.g.* `PUSH` x `PUSH` x and `PUSH` x `DUP1`. This may be resolved by defining an additional precedence on the rules, *e.g.*, based on the size of their bytecode. To make a terminating set of rules confluent, one can use *completion*—automatically if we employing tools such as Ctrl [13]. Finally, one could imagine more expressive rules such as `PUSH` x `PUSH` y `ADD` $\Rrightarrow$ `PUSH` z *where* $z = x + y$. Such rules allow to capture constant folding. To do so, rules in *constrained rewriting* [9] come with constraints over a theory as used in SMT solvers.

6 Conclusion

We propose a pipeline to populate the peephole optimizer of a smart contract compiler with three phases to (1) find optimizations, from which we (2) generate rules, and (3) a feedback

[8] We chose WANDA as its support for types allowed us to leverage that arguments of `PUSH` are words, which greatly aided the automated proof.

mechanism to apply the rules. We demonstrate our approach for EVM bytecode using the tools ebso, sorg, and ppltr, generating 993 peephole optimization rules from the 250 most called contracts of the Ethereum blockchain. We successfully applied our rules to the 1000 most called contracts and discarded 146 376 instructions, saving 435 002 g and 4.5 % storage space. An advantage of our approach lies in its modularity. On the one hand in the modularity of the phases. One could, for example, obtain additional optimizations in a different manner and incorporate them easily. On the other hand, there is the modularity inherent to peephole optimization rules being applied to short programs: it enables an iterative approach to encoding and optimizing instructions based on feasibility and profitability.

Our approach is tailored towards new, rapidly evolving languages and their compilers with clear cost models such as gas metering, and we believe readily applies to languages other than EVM bytecode such as Move and Michelson.

References

[1] Elvira Albert, Pablo Gordillo, Albert Rubio, and Maria A Schett. Synthesis of super-optimized smart contracts using Max-SMT. In *Proc. 32nd CAV*, 2020. To appear.

[2] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, New York, NY, USA, 1998.

[3] Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. In *Proc. 12th ASPLOS*, pages 394–403. ACM, 2006. doi: 10.1145/1168857.1168906.

[4] Sam Blackshear, Evan Cheng, David L Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Dario Russi, Stephane Sezer, Tim Zakian, and Runtian Zhou. Move: A language with programmable resources. https://developers.libra.org/docs/assets/papers/libra-move-a-language-with-programmable-resources/2020-04-09.pdf.

[5] Ting Chen, Zihao Li, Hao Zhou, Jiachi Chen, Xiapu Luo, Xiaoqi Li, and Xiaosong Zhang. Towards saving money in using smart contracts. In *Proc. 40th ICSE-NIER*, pages 81–84. ACM, 2018. doi: 10.1145/3183399.3183420.

[6] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proc. 14th TACAS*, pages 337–340. Springer-Verlag, 2008. doi: 10.1007/978-3-540-78800-3_24.

[7] Abhinav Jangda and Greta Yorsh. Unbounded superoptimization. In *Proc. Onward! 2017*, pages 78–88. ACM, 2017. doi: 10.1145/3133850.3133856.

[8] Cynthia Kop. *Higher Order Termination*. PhD thesis, Vrije Universiteit, Amsterdam, 2012.

[9] Cynthia Kop and Naoki Nishida. Constrained term rewriting tool. In *Proc. 20th LPAR*, volume 9450 of *LNCS*, pages 549–557, 2015. doi: 10.1007/978-3-662-48899-7.

[10] Nomadic Labs. Michelson: the language of smart contracts in tezos. https://tezos.gitlab.io/whitedoc/michelson.html.

[11] Julian Nagele, Bertram Felgenhauer, and Aart Middeldorp. CSI: New evidence – A progress report. In *Proc. CADE-26*, volume 10395 of *LNAI*, pages 385–397, 2017. doi: 10.1007/978-3-319-63046-5.

[12] Julian Nagele and Maria A. Schett. Blockchain superoptimizer. In *Preproc. 29th LOPSTR*, pages 166–180, 2019. arXiv:2005.05912.

[13] Sarah Winkler and Aart Middeldorp. Completion for logically constrained rewriting. In *Proc. 3rd FSCD*, volume 108 of *LIPIcs*, pages 30:1–30:18, 2018. doi: 10.4230/LIPIcs.FSCD.2018.30.

[14] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. Technical Report Byzantium Version e94ebda, 2018. https://ethereum.github.io/yellowpaper/paper.pdf.

Formal Specification and Verification of Solidity Contracts with Events

Ákos Hajdu[1], Dejan Jovanović[2], and Gabriela Ciocarlie[2]

[1] Budapest University of Technology and Economics, Budapest, Hungary
`hajdua@mit.bme.hu`
[2] SRI International, New York City, USA
`{dejan.jovanovic,gabriela.ciocarlie}@sri.com`

Abstract

Events in the Solidity language provide a means of communication between the on-chain services of decentralized applications and the users of those services. Events are commonly used as an abstraction of contract execution that is relevant from the users' perspective. Users must, therefore, be able to understand the meaning and trust the validity of the emitted events. This paper presents a source-level approach for the formal specification and verification of Solidity contracts with the primary focus on events. Our approach allows specification of events in terms of the on-chain data that they track, and predicates that define the correspondence between the blockchain state and the abstract view provided by the events. The approach is implemented in SOLC-VERIFY, a modular verifier for Solidity, and we demonstrate its applicability with various examples.

1 Introduction

Ethereum is a public, blockchain-based computing platform supporting the development of decentralized applications [11]. The core of such applications are programs – termed smart contracts [10] – deployed on the blockchain. While Ethereum nodes run a low-level virtual machine (EVM [11]), smart contracts are usually written in a high-level, contract-oriented language, most notably Solidity [9]. The contract code can be executed by issuing transactions to the network, which are then processed by the participating nodes. Results of a completed transaction are provided to the issuing user, and other interested parties observing the contract, through transaction receipts. While the blockchain is publicly available for users to inspect and replay the transactions, the contracts can communicate important state changes, including intermediate changes, by emitting events [1]. Events usually represent a limited abstract view of the transaction execution that is relevant for the users, and they can be read off the transaction receipts. The common expectation is that by observing the events, the user can reconstruct the relevant parts of the current state of the contracts. Technically, events can be viewed as special triggers with arguments that are stored in the blockchain logs. While these logs are programmatically inaccessible from contracts, the users can easily subscribe to and observe the events with the accompanying data. For example, a token exchange application can monitor the current state of token balances by tracking transfer events in the individual token contracts.

Smart contracts, as any software, are also prone to bugs and errors. In the Ethereum context, any flaws in contracts come with potentially devastating financial consequences, as demonstrated by various infamous examples [2]. While there has been a great interest in applying formal methods to smart contracts [2, 4], events are usually considered merely a logging mechanism that is not relevant for functional correctness. However, since events are a central state-change notification mechanism for users of decentralized applications, it is crucial that the users are able to understand the meaning and trust the validity of the emitted events.

In this paper, we propose a source-level approach for the formal specification and verification of Solidity contracts with the primary focus on events. Our approach provides in-code annotations to specify events in terms of the blockchain data they track, and to declare events possibly emitted by functions. We verify that (1) whenever tracked data changes, a corresponding event is emitted, and (2) an event can only be emitted if there was indeed a change. Furthermore, to establish the correspondence between the abstract view provided by events and the actual execution, we allow events to be annotated with predicates (conditions) that must hold before or after the data change. We implemented the proposed approach in the open-source[1] SOLC-VERIFY [7, 6] tool and demonstrated its applicability via various examples. SOLC-VERIFY is based on modular program verification, but we present our idea in a more general setting that can serve as a building block for alternative verification approaches.

2 Background

Solidity [9] is a high-level, contract-oriented programming language supporting the rapid development of smart contracts for the Ethereum platform. We briefly introduce Solidity by restricting our presentation to the aspects relevant for events. An example contract (`Registry`) is shown in Figure 1. Contracts are similar to classes in object-oriented programming. A contract can define additional types, such as the `Entry` struct in the example, consisting of a Boolean flag and an integer data. The persistent data stored on the blockchain can be defined with *state variables*. The example contract declares a single variable `entries`, which is a mapping from addresses to `Entry` structs. Contracts can also define *events* including possible arguments. The example declares two events, `new_entry` and `updated_entry`, to signal a new or an updated entry, respectively. Both events take the address and the new value for the data as their arguments. Finally, functions are defined that can be called as transactions to act on the contract state. The example defines two functions: `add` and `update`. The `add` function first checks with a `require` that the data corresponding to the caller address (`msg.sender`) is not yet set. If the condition of `require` does not hold, the transaction is reverted. Otherwise, the function sets the data and the flag, and emits the `new_entry` event. The `update` function is similar to `add`, with the exception that the data must already be set, and the new value should be larger than the old one (for illustrative purposes).

Note that Solidity puts no restrictions on the emitted events, and a faulty (or malicious) contract could both emit events that do not correspond to state changes or miss triggering an event on some change [5], potentially misleading users. In the case of the `Registry` contract, the events are emitted correctly, and the user can reproduce the changes in `entries` by relying solely on the emitted events and their arguments.

SOLC-VERIFY [7] is a source-level verification tool for checking functional correctness of smart contracts. SOLC-VERIFY takes contracts written in Solidity and provides various in-code annotations to specify functional behavior (e.g., pre- and postconditions, invariants). SOLC-VERIFY translates the annotated contracts to the Boogie Intermediate Verification Language (IVL) and uses the Boogie verifier [3] to perform modular verification by discharging verification conditions to SMT solvers. This paper presents extensions to the specification and translation capabilities of SOLC-VERIFY that enable reasoning about Solidity events. We propose event-specific annotations (Section 3) and use them to instrument the code during translation with additional conditions to be verified (Section 4).

[1] https://github.com/SRI-CSL/solidity/tree/merge

```
contract Registry {
  struct Entry { bool set; int data; } // User-defined type

  mapping(address=>Entry) entries; // State variable

  /// @notice tracks-changes-in entries
  /// @notice precondition !entries[at].set
  /// @notice postcondition entries[at].set && entries[at].data == value
  event new_entry(address at, int value);

  /// @notice tracks-changes-in entries
  /// @notice precondition entries[at].set && entries[at].data < value
  /// @notice postcondition entries[at].set && entries[at].data == value
  event updated_entry(address at, int value);

  /// @notice emits new_entry
  function add(int value) public {
    require(!entries[msg.sender].set);
    entries[msg.sender].set = true;
    entries[msg.sender].data = value;
    emit new_entry(msg.sender, value);
  }

  /// @notice emits updated_entry
  function update(int value) public {
    require(entries[msg.sender].set && entries[msg.sender].data < value);
    entries[msg.sender].data = value;
    emit updated_entry(msg.sender, value);
  }
}
```

Figure 1: An example contract illustrating Solidity events. Users of the contract can associate an integer value to their address and can later update it with a larger value.

3 Specification of Events

Our approach provides in-code annotations to specify events in terms of the on-chain data that they track for changes. Furthermore, additional predicates can specify the correspondence between the abstract view provided by events and the actual data, before and after the change.

Data changes and checkpoints. Each event can declare a set of contract state variables that it *tracks* for changes. In the `Registry` example (Figure 1), both events track the single state variable `entries`, as specified by the `tracks-changes-in` annotations. Intuitively, we use the tracking of changes to make sure that (1) if a tracked variable changes, a corresponding event must be emitted after; and (2) an event should be emitted only if some of its tracked variables have changed before. As data changes often occur in multiple steps (e.g., updating both members of a struct in the function `add` of Figure 1), or conditionally, events cannot always be emitted directly after a single modifying statement. Therefore, we define the precise semantics of "before" and "after" by introducing *before-* and *after-checkpoints*. Before-checkpoints of an event are determined dynamically by the first change in a variable they track. In contrast, after-checkpoints are defined by static barriers, marking the latest point in code where the emitting should be fulfilled. Currently, we define loop and transaction boundaries (external calls to public functions and function return) as after-checkpoints. The semantics of checkpoints is that an event corresponding to a state variable change must be emitted at some point between before- and after-checkpoints, which also clears the before-checkpoint. Conversely, an event can only be emitted if a tracked variable indeed changed (there was a before-checkpoint).

Event pre- and postconditions. In addition to the set of tracked variables, events can also be annotated with *predicates* that define conditions over the state variables and the arguments of the event. There are two kinds of predicates: *pre-* and *postconditions*. Preconditions capture the values of state variables at the before-checkpoint, while postconditions correspond to the point when the event is emitted. In the `Registry` contract (Figure 1), both events (`new_entry` and `updated_entry`) have the same postcondition, namely that the data at the given address must be set and its value must match the value in the argument. The precondition of `new_entry` is that the data must not yet be set, while for `updated_entry`, it must be set and its value should be smaller than the event argument. Postcondition expressions often need to connect the state at the point of emit and before the change. As an example, consider the `transfer` function of the token contract in Figure 2 that deducts the sender's balance and increases the receiver's. To specify the postcondition of the `Transfer` event, we need to relate the new balances to the previous balances. We provide a special `before` function – to be used in postconditions – that refers to previous values of state variables.

Functions. We require contract functions to be annotated with the events that they possibly emit using the `emits` keyword. For example, the `add` and `update` functions in Figure 1 can emit `new_entry` and `updated_entry` respectively. If a function calls other functions (including base constructors), the callee's emitted events must also be included in the caller's specifications.

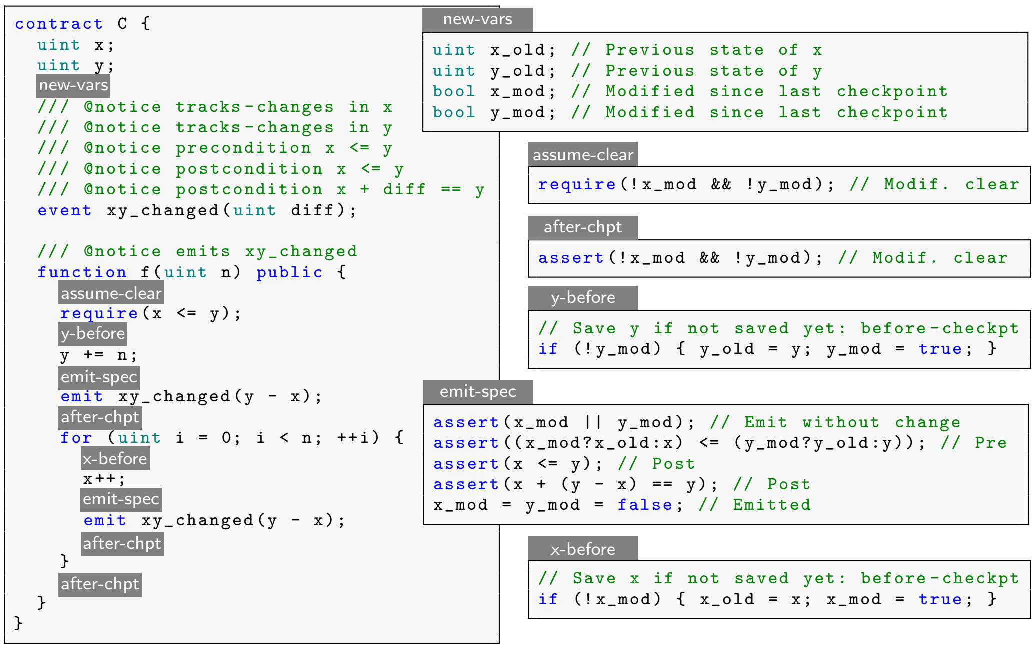
```
contract Token {
  mapping(address=>uint) balances;

  /// @notice tracks-changes-in balances
  /// @notice precondition balances[from] >= amount
  /// @notice postcondition balances[from] == before(balances[from]) - amount
  /// @notice postcondition balances[to] == before(balances[to]) + amount
  event Transfer(address from, address to, uint amount);

  /// @notice emits Transfer
  function transfer(address to, uint amount) public {
    require(balances[msg.sender] >= amount && msg.sender != to);
    balances[msg.sender] -= amount;
    balances[to] += amount;
    emit Transfer(msg.sender, to, amount);
  }
}
```

Figure 2: A token contract illustrating event postconditions that refer to previous state.

4 Verification

A contract with events and specifications is checked in two steps. First, a syntactical check is performed to ensure that functions only emit events that they specified (via `emits` annotations). Then, we check the data tracking specifications and predicates by translating the contract to the input language of a verifier and instrumenting the code with the checks and the required bookkeeping. In our implementation, we use the Boogie IVL and verifier [3], but we present our solution in a general way that can be reused in other Solidity verifiers.

Function emits. We first check whether functions only emit those events that are specified via `emits` annotations. This is a syntactic check on the Solidity AST: we find all emit statements



```
contract C {
  uint x;
  uint y;
  new-vars
  /// @notice tracks-changes in x
  /// @notice tracks-changes in y
  /// @notice precondition x <= y
  /// @notice postcondition x <= y
  /// @notice postcondition x + diff == y
  event xy_changed(uint diff);

  /// @notice emits xy_changed
  function f(uint n) public {
    assume-clear
    require(x <= y);
    y-before
    y += n;
    emit-spec
    emit xy_changed(y - x);
    after-chpt
    for (uint i = 0; i < n; ++i) {
      x-before
      x++;
      emit-spec
      emit xy_changed(y - x);
      after-chpt
    }
    after-chpt
  }
}
```

new-vars
```
uint x_old; // Previous state of x
uint y_old; // Previous state of y
bool x_mod; // Modified since last checkpoint
bool y_mod; // Modified since last checkpoint
```

assume-clear
```
require(!x_mod && !y_mod); // Modif. clear
```

after-chpt
```
assert(!x_mod && !y_mod); // Modif. clear
```

y-before
```
// Save y if not saved yet: before-checkpt
if (!y_mod) { y_old = y; y_mod = true; }
```

emit-spec
```
assert(x_mod || y_mod); // Emit without change
assert((x_mod?x_old:x) <= (y_mod?y_old:y)); // Pre
assert(x <= y); // Post
assert(x + (y - x) == y); // Post
x_mod = y_mod = false; // Emitted
```

x-before
```
// Save x if not saved yet: before-checkpt
if (!x_mod) { x_old = x; x_mod = true; }
```

Figure 3: Example contract with instrumentation snippets for checking event specifications.

in the function and check whether the corresponding events are specified to be emitted. When a function calls other functions *internally* (i.e., from the same contract), we apply a modular check based on the call graph: all events specified to be emitted by the callee must also be specified by the caller. On the other hand, we currently ignore *external* calls (such as `.call()` or `.transfer()`). Such external calls cannot modify state variables or trigger events from the current contract directly (as they are non-public). Indirect modifications and emits are possible by calling back public functions, but those are specified and checked independently (modularity of reasoning). Finally, we also verify at each assignment (to a tracked variable), whether the function specifies a corresponding event to be emitted.

Data tracking and predicates. Verification of data tracking and predicates is performed by instrumenting the contract code with additional variables and statements to save state and to make extra checks at checkpoints. For clarity, we describe the instrumentation on the Solidity level. We illustrate the approach through the example contract in Figure 3, which has two state variables `x` and `y`, and whenever one of them changes, an event is emitted with their current difference. Furthermore, `x <= y` should hold both at the before- and the after-checkpoint. The extra instructions are displayed as labels where they are injected, while the corresponding code can be found in the snippets to the right.

For each state variable that is tracked by any event, we introduce two additional variables in the contract: one with the same type to save the before-state, and a Boolean flag to indicate whether the data has been modified (snippet new-vars in Figure 3). Functions are then instrumented with extra statements to save state, enforce after-checkpoints (barriers) and to perform specification checks when events are emitted. Functions ensure on entry that none of the variables tracked by their specified events have been modified since the checkpoint before the call (snippet assume-clear). In other words, all relevant events must have been emitted before making the call. In modular verification, this assumption becomes a precondition to the function. Before each modification (assignment statement), if the state variable is not modified yet, the current value is stored[2] in the helper variable and the flag for modification is set, introducing a before-checkpoint (snippets y-before and x-before).

At each `emit` statement, several checks are added (snippet emit-spec). First, we check that the data has indeed been modified, otherwise the event should not be emitted. Then we check each pre- and postcondition. By default, preconditions refer to the before-state and postconditions to the current values, except if the variable is explicitly wrapped with `before()`. Note that we refer to the previous value of a variable `v` with `v_mod ? v_old : v` because in general there might be variables that were not modified (e.g., `x` at the first emit in Figure 3). After performing the checks, emitting the event clears the flags (before-checkpoints). Finally, before returning, functions enforce after-checkpoints by asserting that no state variable is in a modified state, i.e., the function cannot end in debt with events (snippet after-chpt). In modular verification, this check becomes a postcondition to the function. We also insert an after-checkpoint before the loop and at the end of every iteration (serving as loop invariant).

Discussion. One potential limitation of our approach is that we consider loop boundaries after-checkpoints: some contracts change the data many times in the loop but only emit a single summarizing event after the loop. This limitation could be alleviated with annotations to "allow delaying" the emit after the loop, but we do not support this as it leads to more complex specification and verification.

Our approach is not tied to Boogie or modular verification. The instrumentation can be performed on the Solidity level, and the correctness of the specification is reduced to checking assertions at particular points in the code. This means that the instrumented code can be fed into any Solidity verifier that can check for assertion failures. The event specifications are deemed correct if and only if there are no related assertion failures.

A possible future use-case of our approach lies in the behavioral analysis of contracts based on logs. Such analyses could reveal relationships individually and across contracts that are not otherwise apparent (e.g., exposing entities that control the blockchain interactions) or attack evidence. Application-level log analysis has been used for a long time for monitoring and security purposes, and most existing techniques assume that application logs can be trusted or, if applications are subverted by attackers, the subversion can be captured [8]. Our approach guarantees the validity of the emitted events, making them even more suitable for such analysis.

5 Conclusion

We presented an approach for the formal specification and verification of Solidity smart contracts that rely on events to communicate with their users, providing an abstract view of their state. We proposed in-code annotations to specify events in terms of the state variables they track for changes. Furthermore, we introduced additional predicates (pre- and postconditions) for specifying conditions on the state before and after the change, establishing the correspondence between the blockchain state and the emitted events. The approach is implemented in SOLC-VERIFY and we demonstrated its applicability with various examples.

[2]Saving data (e.g., mappings) with assignments might not yield valid Solidity code. This code is for clarity of presentation and is handled by SOLC-VERIFY internally.

References

[1] A guide to events and logs in Ethereum smart contracts. https://consensys.net/blog/blockchain-development/guide-to-events-and-logs-in-ethereum-smart-contracts, 2016.

[2] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on Ethereum smart contracts. In *Principles of Security and Trust*, volume 10204 of *Lecture Notes in Computer Science*, pages 164–186. Springer, 2017.

[3] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects*, volume 4111 of *LNCS*, pages 364–387. Springer, 2006.

[4] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. A survey on Ethereum systems security: Vulnerabilities, attacks and defenses, 2019.

[5] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen. Defining smart contract defects on Ethereum. *IEEE Transactions on Software Engineering*, 2020. Early access.

[6] Ákos Hajdu and Dejan Jovanović. SMT-friendly formalization of the solidity memory model. In *Programming Languages and System*, volume 12075 of *Lecture Notes in Computer Science*, pages 224–250. Springer, 2020.

[7] Ákos Hajdu and Dejan Jovanović. solc-verify: A modular verifier for Solidity smart contracts. In *Verified Software. Theories, Tools, and Experiments*, volume 12301 of *Lecture Notes in Computer Science*, pages 161–179. Springer, 2020.

[8] Shiqing Ma, Juan Zhai, Fei Wang, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. MPI: Multiple perspective attack investigation with semantic aware execution partitioning. In *Proceedings of the 26th USENIX Security Symposium*, pages 1111–1128. USENIX Association, 2017.

[9] Solidity documentation. https://solidity.readthedocs.io, 2020.

[10] Nick Szabo. Smart contracts, 1994.

[11] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. https://ethereum.github.io/yellowpaper/paper.pdf, 2019.

Part II.

Merkel trees and Bitcoin

Authenticated Data Structures as Functors in Isabelle/HOL

Andreas Lochbihler
Digital Asset, Zurich, Switzerland
andreas.lochbihler@digitalasset.com,mail@andreas-lochbihler.de

Ognjen Marić
Digital Asset, Zurich, Switzerland
ognjen.maric@digitalasset.com

Abstract

Merkle trees are ubiquitous in blockchains and other distributed ledger technologies (DLTs). They guarantee that the involved systems are referring to the same binary tree, even if each of them knows only a subtree. Inclusion proofs allow knowledgeable systems to share subtrees with other systems and the latter can verify the subtrees' authenticity. Often, blockchains and DLTs use data structures more complicated than binary trees; *authenticated data structures* generalize Merkle trees to such structures.

We show how to formally define and reason about authenticated data structures, their inclusion proofs, and operations thereon as datatypes in Isabelle/HOL. Our approach is modular and allows us to construct complicated trees from reusable building blocks, which we call Merkle functors. Merkle functors include sums, products, and function spaces and are closed under composition and least fixpoints. As a practical application, we model the hierarchical transactions of Canton, a practical interoperability protocol for distributed ledgers, as authenticated data structures. This is a first step towards formalizing the Canton protocol and verifying its integrity and security guarantees.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Higher order logic; Theory of computation → Cryptographic primitives

Keywords and phrases Merkle tree, functor, distributed ledger, datatypes, higher-order logic

Related Version An extended version is available at `www.canton.io/publications/iw2020.pdf`.

Supplementary Material The formalization is available in the Archive of Formal Proofs [12].

1 Introduction

Authenticated data structures (ADSs) allow systems to use succinct digests to ensure that they are referring to the same data structure, even if each system knows only a part of the data structure. The benefits are twofold. First, this saves storage and bandwidth: the systems can store only the structure's parts that are relevant for them, and transmit just digests, not the whole structure. Blockchains use ADSs for this reason, both in the core design and in various optimizations (e.g., Bitcoin's lightweight clients). Second, ADSs can keep parts of the structure confidential to the subset of the systems involved in processing the structure. For example, distributed ledger technology (DLT) promises to keep multiple organizations synchronized on their shared business data. Synchronization requires transactions, i.e., atomic changes to the shared state. Yet organizations often do not want to share their full state with all involved parties. Some DLT protocols such as the Canton interoperability protocol [5] and Corda [6] leverage ADSs to provide both transactionality and varying levels of confidentiality. Formal reasoning about blockchains and DLTs thus often requires mechanised theories of ADSs. In fact, the formalization of Canton was the starting point for this work.

Merkle trees [14] are the prime example of an ADS. They are binary trees of digests, i.e., cryptographic hashes. Leaves contain data hashes, and inner nodes combine their children's

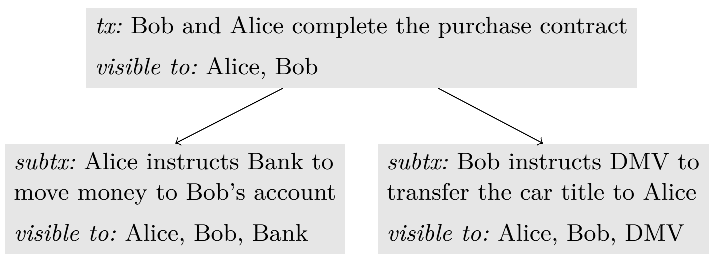

tx: Bob and Alice complete the purchase contract
visible to: Alice, Bob
subtx: Alice instructs Bank to move money to Bob's account
visible to: Alice, Bob, Bank
subtx: Bob instructs DMV to transfer the car title to Alice
visible to: Alice, Bob, DMV

Figure 1 A hierarchical Canton transaction. DMV is the department of motor vehicles.

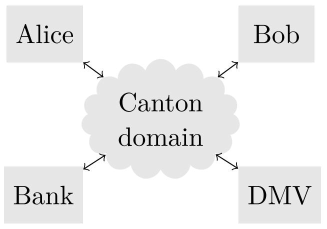


Figure 2 Example topology of a Canton-based distributed ledger

hashes using a hash function h. An *inclusion proof* shows that a tree t includes a subtree st. It consists of the roots of t and st and the siblings of nodes on the path between these roots. The proof is valid if the hash of every node on the path is h of the children's hashes. It is sound, i.e., does prove inclusion, if h is collision-resistant. It keeps the rest of the tree confidential if h is preimage-resistant and the hashed data contains sufficient entropy.

ADSs [15] generalize these ideas to arbitrary finite tree data structures, whose hierarchies can conveniently encode more complex relationships between data. Our main example are the hierarchical transactions [2] in the Canton protocol. Suppose that Alice wants to sell a car title to Bob. Figure 1 shows the corresponding Canton transaction for exchanging the money and the title. (We take significant liberties in the presentation of Canton in this paper and focus on parts relevant for the construction of ADSs and for reasoning about them.) The transaction is generated from a smart contract (written in the DAML [7] programming language) implementing the purchase agreement.

The transactions' hierarchical nature benefits Canton in three crucial ways. First, complex transactions can be composed from simpler building blocks, which are transactions themselves. The purchase transaction above composes two such sub-transactions: the money transfer and the title transfer. Second, participants learn only the contents of subtransactions they are involved in. Above, the Bank only sees the money transfer, but not what Alice bought; similarly, the DMV does not learn the car's price. This also improves scalability, as everyone processes only the subtransactions they are involved in. Third, the hierarchy enables correct delegation in Canton's built-in authorization logic even in a Byzantine setting. Canton encodes this hierarchy, enriched with some additional data, in ADSs, and exchanges inclusion proofs for subtransactions. We give more details throughout the paper, but summarize the resulting requirements on the formalization here:

1. It must support ADS digests, to check that two inclusion proofs refer to the same ADS. This allows the example transaction to commit atomically, even if the Bank and the DMV see only a part of it.
2. Proofs must enable proving inclusion for multiple subtrees simultaneously, not just single subtree as standard. Canton uses such inclusion multi-proofs to save bandwidth.
3. Inclusion proofs refering to the same ADS must be mergeable into one multi-proof. In the example of Figure 1, Alice receives inclusion proofs for the entire transaction as well as both sub-transactions, and merges them to a single data structure, the entire transaction.

In this work, we show how to modularly define ADSs as datatypes in Isabelle/HOL. The modular approach is our main theoretical contribution. It allows us to construct complicated trees from small reusable building blocks, for which properties are easy to prove. To that end, we consider authenticated data structures as so-called *Merkle functors* and equip them with appropriate operations and their specifications. The class of Merkle functors includes sums, products, and function spaces, and is closed under composition and least fixpoints. Hence, the construction works for any inductive datatype (sums of products and exponentials). Concrete functors are defined as algebraic datatypes using Isabelle/HOL's datatype package [1]. This shallow embedding is a significant practical benefit, as it enables the use of Isabelle's rich reasoning infrastructure for datatypes. The construction lives in the symbolic model, i.e., we assume that no hash collisions occur. Finally, we show that the theory is suitable for constructing concrete real-world instances such as Canton's transaction trees. Our formalization is available in the Isabelle AFP [12].

The rest of the paper is structured as follows. In Section 2, we provide the background on Canton and use it to motivate our abstract interface for ADSs. Section 3 shows how to construct such interfaces for tree-like structures in a modular fashion. Section 4 demonstrates how to create inclusion proofs for general rose trees and Canton transactions in particular. We discuss the related work in Section 5 and conclude in Section 6.

2 Operations on Authenticated Data Structures

We now present the interfaces for ADSs, motivated by their application to Canton. Figure 2 shows a suitable Canton-based deployment for our example transaction. The participants transact using Canton, a distributed commit protocol similar to two-phase commit. The protocol is run over a Canton *domain* operated by a third party that acts as the commit coordinator. While the participants may be Byzantine, the domain is assumed to be honest-but-curious. That is, it is trusted to correctly execute the protocol, but it should not learn the contents of a transaction (e.g., how much Alice pays to Bob). Unlike in most other DLT solutions, participants share business data only on a need-to-know basis [4]. In particular, the domain receives business data only in encrypted form or as a digest. The domain may only learn the metadata that allows the protocol to tolerate Byzantine participants.

These privacy requirements motivate the hierarchical transactions that Canton uses, which are encoded in *transaction trees*. The tree for the example transaction from Figure 1 is shown in Figure 3. Each (sub)-transaction of Figure 1 is turned into a *view* in Figure 3, which consists of the view *data* and view *metadata*. For example, the node labeled by 1 in Figure 3 is the view corresponding to the top-level transaction in Figure 1. Its first two children contain the view's data and metadata. The metadata lists who is affected by the view and should therefore participate in the commit protocol (here, Alice and Bob), and is shared with Alice, Bob and the domain. The view data contains the confidential data with the actual state updates, and is shared only with Alice and Bob. This view also has two *subviews*, which correspond to the sub-transactions in Figure 1 as expected. Views can have an arbitrary number of subviews; e.g., the views labeled by 1.1 and 1.2 have no subviews.

Additionally, the two leaf children of the tree root store metadata describing transaction-wide parameters that apply to all views. The first is visible to the domain and the participants involved in the transaction; the second only to the latter. Formally, the transaction tree can be modelled by the following datatypes, for some types *common-metadata*, *participant-metadata*, *view-metadata*, and *view-data* whose contents are irrelevant for this paper.

datatype *view* = *View* ‹*view-metadata*› ‹*view-data*› (*subviews*: ‹*view list*›)
datatype *transaction* =
  Transaction ‹*common-metadata*› ‹*participant-metadata*› (*views*: ‹*view list*›)

In Figure 3, the *Transaction* and *View* constructors become the inner nodes (black circles) and the data sits at the leaves (grey rectangles).

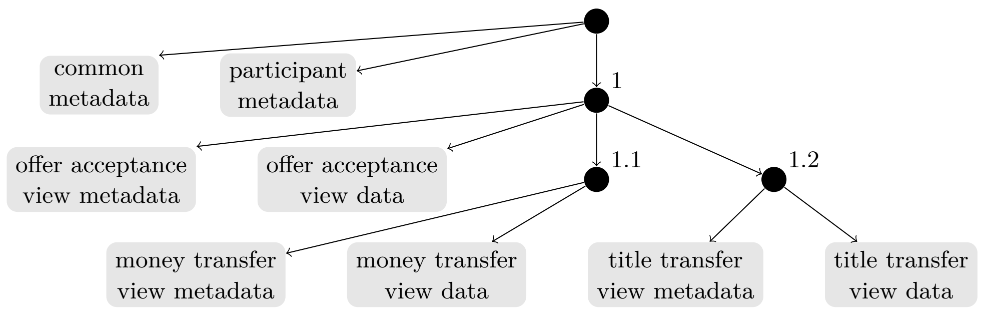

common metadata
participant metadata
1
offer acceptance view metadata
offer acceptance view data
1.1
1.2
money transfer view metadata
money transfer view data
title transfer view metadata
title transfer view data

Figure 3 Simplified Canton transaction tree for car title sale of Figure 1

The participants and the domain can use a root hash of an ADS over a *Transaction* to ensure that they are all referring to the same transaction tree. When constructing ADS hashes, we need to consider ADSs with multiple roots (i.e., forests) rather than just a single root like in a Merkle tree. For example, computing the hash of an inner node in a Merkle tree requires taking a hash over both of its children, i.e., over the forest constructed from its two children. The concrete hash operation depends on the shape of the forest (a pair in this case). The new root is again a degenerate forest of a single tree with a single root hash. This view underlies our modular construction principle in Section 3.

In this paper, we use the following Isabelle notations: Type variables $'a$, $'b$ are prefixed by $'$ like in Standard ML. Type constructors like *list* are usually written postfix as in *string list*. Exceptions are the function space $\Rightarrow$, sums $+$, and products $\times$, all written infix. The notation $t :: \tau$ denotes that the term t has the type τ. In our construction, we will use the following decorations. Raw data to be arranged in an ADS is written as usual, e.g., $'a$, $'a$ *list*. Hashes and forests of hashes carry a subscript $_h$ as in $'a_h$. We leave hashes for now abstract as type variables and define them only in Section 3. Since the root hash identifies an ADS, we represent ADSs by their hashes.

A root hash makes communication more efficient, but we require more. For example, the Bank does not know the contents or participants of view 1.2; the domain hides the latter. Still, the Bank must ensure that the view 1.1 is really included in the transaction tree. In general, the views visible to a participant are called the participant's *projection* of the transaction. Canton aims to achieve the following integrity guarantee [2]: There exists a shared ledger that adheres to the underlying DAML smart contracts such that its projection to each honest participant consists exactly of the updates that have passed the participant's local checks. This requires the ability to prove that a substructure is included in a root hash.

Inclusion proofs are therefore the main workhorse in our formalization and the focus of this paper. We denote the type of inclusion proofs over the source type with the subscript $_m$, e.g., $'a_m$, $('a_m, 'a_h)$ $tree_m$. We need two operations on inclusion proofs:

1. Computing the (forest of) root hashes of an inclusion proof, in order to identify the ADS to which the inclusion proof corresponds.
2. Merging two inclusion proof with the same root hash.

Accordingly, we introduce two type synonyms for these operations:

type_synonym $('a_m, 'a_h)$ *hash* = ‹$'a_m \Rightarrow 'a_h$›
type_synonym $'a_m$ *merge* = ‹$'a_m \Rightarrow 'a_m \Rightarrow 'a_m$ *option*›

We model the merge operation as a partial function using the *option* that returns *None* iff the inclusion proofs have different (forests of) root hashes. We require that merging is idempotent, commutative, and associative. The merge operation makes inclusion proofs with the same hash into a semi-lattice, where the induced order treats an inclusion proof as smaller than another if it reveals less. In that case, we say that the smaller is a *blinding* of the larger inclusion proof.

type_synonym $'a_m$ *blinding-of* $= \langle 'a_m \Rightarrow 'a_m \Rightarrow bool\rangle$

▶ **Definition 1.** *A* Merkle interface *consists of three operations* $h :: ('a_m, 'a_h)$ *hash and* $m :: 'a_m$ *merge and* $bo :: 'a_m$ *blinding-of with the following properties:*

1. *Merge respects hashes, i.e.,* $(h\ a = h\ b) = (\exists\, ab.\ m\ a\ b = Some\ ab)$.
2. *Merge is idempotent, i.e.,* $m\ a\ a = Some\ a$.
3. *Merge is commutative, i.e.,* $m\ a\ b = m\ b\ a$.
4. *Merge is associative, i.e.,* $m\ a\ b \gg\!= m\ c = m\ b\ c \gg\!= m\ a$,
   where $(\gg\!=)$ *is the monadic bind on the option type.*
5. *Blinding is induced by merge, i.e.,* $bo\ a\ b = (m\ a\ b = Some\ b)$.

So merge is the least upper bound in the blinding relation:

$$(m\ a\ b = Some\ ab) = (bo\ a\ ab \wedge bo\ b\ ab \wedge (\forall\, u.\ bo\ a\ u \longrightarrow bo\ b\ u \longrightarrow bo\ ab\ u))$$

Also, the equivalence closure of the blinding relation gives the equivalence classes of the inclusion proofs under the hash function: *equivclp bo* $=$ *vimage2p* $h\ h\ (=)$ where *equivclp* R denotes the equivalence closure of R and *vimage2p* $f\ g\ R = (\lambda x\ y.\ R\ (f\ x)\ (g\ y))$ the preimage of a relation under a pair of functions.

Isabelle/HOL's term language is not expressive enough to automatically create the ADS and inclusion proof types of arbitrary tree-shaped data, define the interface's operation, or build inclusion proofs for subtrees of tree-shaped data. Instead, in the next two sections, we show how to systematically construct these types and operations.

3 Modularly Constructing Forests of Authenticated Data Structures

In this section, we develop the theory to modularly construct ADSs, their inclusion proofs as HOL datatypes, and Merkle interfaces over them. We start with the concept of a blindable position (Section 3.1), which models an ADS node, and show how we obtain ADSs for Canton's transaction trees by introducing blindable positions in the right spots of the datatype definitions (Section 3.2).

The Merkle interface specification is not inductive and therefore not preserved by datatype constructions. We thus generalize it and show that functor composition and least fixpoint preserve the generalization (Section 3.4). Finally, we show that sums, products and function spaces preserve the generalization (Section 3.5) and compose these preservation results to obtain the Merkle interface properties for Canton transactions (Section 3.6).

3.1 Blindable position

A *blindable position* represents a node (inner node or leaf) in an ADS. Recall that "blinding" allows an inclusion proof to hide the node contents by using just the root hash of the node. In this work, we model such hashes symbolically, that is, as injective functions, and assume that no hash collisions occur. We do not assume surjectivity though: some hashes do not correspond to any value. We model such values as garbage coming from a countable set (the naturals). This suffices as digests contain only a finite amount of information.

datatype $'a_h$ *blindable*$_h$ = *Content* ⟨$'a_h$⟩ | *Garbage* ⟨*nat*⟩

Since the hash function is injective, we can identify the values $'a$ with a subset of the hashes, namely those of form *Content*. Accordingly, we could also have written $'a$ *blindable*$_h$ instead of $'a_h$ *blindable*$_h$. However, as an ADS contains hashes of hashes, $'a_h$ is more accurate here. For example, a degenerate Merkle tree with a single leaf, which stores some data x, has the root hash *Content* x.

What does an inclusion proof for this tree look like? It can take two forms. Either it reveals x, i.e., the leaf is not blinded, or it does not reveal x, i.e., the leaf is blinded. The following datatype formalizes these cases.

datatype ($'a_m$, $'a_h$) *blindable*$_m$ = *Unblinded* ⟨$'a_m$⟩ | *Blinded* ⟨$'a_h$ *blindable*$_h$⟩

Similar to *blindable*$_h$, inclusion proofs are nested, e.g., if a Merkle tree leaf contains another Merkle tree as data. We therefore use the inclusion proof type variable $'a_m$ instead of $'a$. In the second case, the hash could be garbage, so we use $'a_h$.

Note that our *blindable*$_h$ hashes are typed: hashes of those ADSs that store *int*s and those that store *string*s in their leaves always differ. In the real world, they can be equal as hashes are just bitstrings. However, for systems which follow security best practices, type flaw attacks lead to different hashes unless a hash collision occurs. Garbage hashes adequately model such confusion possibilities: a hash of the *int* Leaf would be treated as garbage in the type of hashes for the ADS of *string*s. This is adequate for inclusion proofs because we care about the contents of a hash only if the position is unblinded and thus of shape *Content*.

Having introduced the types for blindable positions, we now define the corresponding operations and show that they satisfy the specification *merkle-interface*. The hash operation *hash-blindable* :: ($'a_m$, $'a_h$) *hash* ⇒ (($'a_m$, $'a_h$) *blindable*$_m$, $'a_h$ *blindable*$_h$) *hash* converts an inclusion proof into the root hash of the tree. It is parameterized by a hash function h_a that converts nested inclusion proofs $'a_m$ into their root hashes $'a_h$. Its definition is straightforward: for unblinded nodes, apply h_a, and for blinded nodes, just take the contained hash. Similarly, the blinding order *blinding-of-blindable* :: ($'a_m$, $'a_h$) *hash* ⇒ $'a_m$ *blinding-of* ⇒ ($'a_m$, $'a_h$) *blindable*$_m$ *blinding-of* is parametrized by the hash h_a and the blinding order bo_a for the nested inclusion proofs, as well as the blindable inclusion proofs to be compared. If both of the compared inclusion proofs unblind the contents, then we compare the contents using bo_a. Otherwise, the first argument is a blinding of the second one only if it is blinded, and if its hash matches the hash of the second argument. Merging of blindable positions is also similar. If both positions are unblinded, *merge-blindable* tries to merge the contents. If both are blinded, it succeeds iff the hashes are the same. Otherwise, it checks that the hashes are the same and, if so, returns the unblinded version. It is straightforward to show the following lemma.

▶ **Lemma 2.** *If h_a, bo_a, and m_a jointly form a Merkle interface, then so do hash-blindable h_a, blinding-of-blindable h_a bo_a, and merge-blindable h_a m_a.*

3.2 Example: Canton transaction trees

We now illustrate how to use *blindable*$_h$ and *blindable*$_m$ to define the ADSs and inclusion proofs for the Canton transaction trees from Section 2. As shown in Figure 3, the transaction tree contains a node for the transaction tree as a whole, every view, and every leaf (*common-metadata*, *participant-metadata view-metadata*, and *view-data*). Yet, the datatype declarations do not contain the information what should become a separate node in the ADS. To make the construction systematic, we start from an isomorphic representation of *view* and *transaction*, where we mark the blindable positions with the type constructor *blindable*, which is just the identity functor:

datatype *view* = *View*
  ‹((*view-metadata blindable* × *view-data blindable*) × *view list*) *blindable*›
datatype *transaction* = *Transaction*
  ‹((*common-metadata blindable* × *participant-metadata blindable*) × *view list*) *blindable*›

To define the hashes and inclusion proofs, we simply replace each type constructor τ with its counterparts τ_h and τ_m. For views, this looks as follows. Here $\times_h$, $\times_m$, list_h, and list_m are type synonyms for × and *list*; Section 3.5 introduces them formally. We abuse notation by writing $\mathit{view\text{-}metadata}_h$ and $\mathit{view\text{-}metadata}_m$ for the blindable position of *view-metadata*.

type_synonym $\mathit{view\text{-}metadata}_h$ = ‹$\mathit{view\text{-}metadata}\ \mathit{blindable}_h$›
type_synonym $\mathit{view\text{-}data}_h$ = ‹$\mathit{view\text{-}data}\ \mathit{blindable}_h$›
datatype view_h = View_h ‹$((\mathit{view\text{-}metadata}_h \times_h \mathit{view\text{-}data}_h) \times_h \mathit{view}_h\ \mathit{list}_h)\ \mathit{blindable}_h$›
type_synonym $\mathit{view\text{-}metadata}_m$ = ‹$(\mathit{view\text{-}metadata},\ \mathit{view\text{-}metadata})\ \mathit{blindable}_m$›
type_synonym $\mathit{view\text{-}data}_m$ = ‹$(\mathit{view\text{-}data},\ \mathit{view\text{-}data})\ \mathit{blindable}_m$›
datatype view_m = View_m
  ‹$((\mathit{view\text{-}metadata}_m \times_m \mathit{view\text{-}data}_m) \times_m \mathit{view}_m\ \mathit{list}_m,$
   $(\mathit{view\text{-}metadata}_h \times_h \mathit{view\text{-}data}_h) \times_h \mathit{view}_h\ \mathit{list}_h)\ \mathit{blindable}_m$›

These types nest hashes and inclusion proofs: A view node, e.g., nests hashes and inclusion proofs for the metadata, the data, and all the subviews. In particular, the view_h and view_m datatypes recurse through the $\mathit{blindable}_h$ and $\mathit{blindable}_m$ type constructors. This works because $\mathit{blindable}_h$ and $\mathit{blindable}_m$ are bounded natural functors (BNFs) [1]. In fact, this transformation works for any datatype declaration thanks to the compositionality of BNFs. The construction for transaction trees is similar.

3.3 Composition

Having defined the types of ADSs, we next must define the operations on ADSs and prove that they form a Merkle interface. Doing so directly is possible, but prohibitively complex. Instead, we modularize the proofs following the structure of the types. We can derive preservation lemmas for all involved type constructors analogous to *merkle-blindable*.

The preservation lemmas are compositional by construction: if $'a_h\ \tau_h/('a_m,\ 'a_h)\ \tau_m$ and $'b_h\ \sigma_h/('b_m,\ 'b_h)\ \sigma_m$ satisfy *merkle-interface*, then so does their composition $'a_h\ \tau_h\ \sigma_h/(('a_m,\ 'a_h)\ \tau_m,\ 'a_h\ \tau_h)\ \sigma_m$. For example, we can define the instance for blindable nodes of type *view-data* compositionally. First, we exploit the fact that every nullary functor satisfies *merkle-interface* with the discrete ordering (=), hash *id* and *merge* defined only for equal operands. Second, we compose *view-data*, viewed as a nullary functor with *blindable*. For example, we define:

abbreviation *hash-view-data* :: ‹$(\mathit{view\text{-}data}_m,\ \mathit{view\text{-}data}_h)\ \mathit{hash}$› **where**
  ‹*hash-view-data* ≡ *hash-blindable id*›

We perform the same constructions on *view-metadata*, and then use composition for the pair *view-metadata* × *view-data*, to get the following (the operations for products will be introduced in Section 3.5).

▶ **Lemma 3.** *The following three operations form a Merkle interface:*

- *hash-prod hash-view-metadata hash-view-data*
- *blinding-of-prod blinding-of-view-metadata blinding-of-view-data*
- *merge-prod merge-view-metadata merge-view-data*

3.4 Inductive generalization for least fixpoints

The *view* datatype is the least fixpoint of the functor

$$'a\ F = ((\textit{view-metadata blindable} \times \textit{view-data blindable}) \times {'a}\ \textit{list})\ \textit{blindable}$$

and so are $\textit{view}_h$ and $\textit{view}_m$ of analogous functors F_h and F_m. Composition gives us a preservation theorem for F, but we need another one for least fixpoints.

Yet, the Merkle interface specification is not inductive and thus not preserved by fixpoints. We now generalize it. Simultaneously, we make the generalization more amenable to Isabelle's proof automation by focusing on the blinding order and characterizing merge as its join. Our generalization splits the Merkle interface into three:

1. The interface *blinding-respects-hashes* assumes that $bo \le \textit{vimage2p}\ h\ h\ (=)$ where $(\le)$ denotes inclusion on binary predicates.
2. The interface *blinding-of-on* formalizes the order properties of the blinding relation bo: Reflexivity $bo\ x\ x$, transitivity $bo\ x\ y \Longrightarrow bo\ y\ z \Longrightarrow bo\ x\ z$, and antisymmetriy $bo\ x\ y \Longrightarrow bo\ y\ x \Longrightarrow x = y$ hold for all $x \in A$ and all y, z: The restriction of x to the set A makes the statement inductive, as A can be instantiated to the set of smaller values in structural induction proofs.
3. The interface *merge-on* extends *blinding-of-on* applied to the type's universal set $\textit{UNIV}$ with the characterization of merge as the join, but now again restricted by a set A. In the unrestricted case $A = \textit{UNIV}$, *merge-on* is equivalent to the Merkle interface.

We are now ready to define the class of Merkle functors. For readability, we only spell out the case of unary functors. The generalization to n-ary functors is as expected.

▶ **Definition 4** (Merkle functor). *A unary BNF F_h and binary BNF F_m constitute a unary Merkle functor if there exist operations $\textit{hash}'_F$ :: $(({'a_h}, {'a_h})\ F_m, {'a_h}\ F_h)$ hash and $\textit{blinding-of}_F$ :: $({'a_m}, {'a_h})$ hash ⇒ ${'a_m}$ blinding-of ⇒ $({'a_m}, {'a_h})\ F_m$ blinding-of and $\textit{merge}_F$ :: $({'a_m}, {'a_h})$ hash ⇒ ${'a_m}$ merge ⇒ $({'a_m}, {'a_h})\ F_m$ merge with the following properties*

Monotonicity
$$\frac{bo \le bo'}{\textit{blinding-of}_F\ h\ bo \le \textit{blinding-of}_F\ h\ bo'}$$

Congruence
$$\frac{\forall a{\in}A.\ \forall b.\ m\ a\ b = m'\ a\ b}{\forall x \in \{y.\ \textit{set}_1\textit{-}F_m\ y \subseteq A\}.\ \forall b.\ \textit{merge}_F\ h\ m\ x\ y = \textit{merge}_F\ h\ m'\ x\ y}$$

Hashes
$$\frac{\textit{blinding-respects-hashes}\ h\ bo}{\textit{blinding-respects-hashes}\ (\textit{hash}_F\ h)\ (\textit{blinding-of}_F\ h\ bo)}$$

Blinding order
$$\frac{\textit{blinding-of-on}\ A\ h\ bo}{\textit{blinding-of-on}\ \{x.\ \textit{set}_1\textit{-}F_m\ x \subseteq A\}\ (\textit{hash}_F\ h)\ (\textit{blinding-of}_F\ h\ bo)}$$

Merge
$$\frac{\textit{merge-on}\ A\ h\ bo\ m}{\textit{merge-on}\ \{x.\ \textit{set}_1\textit{-}F_m\ x \subseteq A\}\ (\textit{hash}_F\ h)\ (\textit{blinding-of}_F\ h\ bo)\ (\textit{merge}_F\ h\ m)}$$

where $\textit{hash}_F\ h = \textit{hash}'_F \circ \textit{map-}F_m\ h\ id$ for the BNF mapper $\textit{map-}F_m$ and the BNF setter $\textit{set}_1\textit{-}F_m\ x$ returns all atoms of type ${'a_m}$ in x :: $({'a_m}, {'a_h})\ F_m$.

Every Merkle functor preserves the Merkle interface specification: set $A = \textit{UNIV}$ in the merge property and use the equivalence between the Merkle interface and *merge-on.* With this, we now state the main theoretical contribution of this paper.

▶ **Theorem 5.** *Merkle functors of arbitrary arity are closed under composition and least fixpoints.*

Proof. (Sketch) Closure under composition is obvious from the shape of the properties and the fact that BNFs are closed under composition. For closure under least fixpoints, we consider a functor F and its least fixpoint T through one of F's arguments. say **datatype** $T = T \langle T\ F \rangle$, and similarly for T_h and T_m. The operations are defined as follows, where we omit all Merkle operation parameters for type parameters that are not affected.

- The hash operation $\mathit{hash\text{-}T'}$ is defined by primitive recursion:

$$\mathit{hash\text{-}T'}\ (T_m\ x) = T_h\ (\mathit{hash\text{-}F'}\ (\mathit{map\text{-}F_m}\ \mathit{hash\text{-}T'}\ x)).$$

- The blinding order $\mathit{blinding\text{-}of\text{-}T}$ is defined inductively by the following rule:

$$\frac{\mathit{blinding\text{-}of\text{-}F}\ \mathit{hash\text{-}T}\ \mathit{blinding\text{-}of\text{-}T}\ x\ y}{\mathit{blinding\text{-}of\text{-}T}\ (T_m\ x)\ (T_m\ y)}$$

  Monotonicity ensures that $\mathit{blinding\text{-}of\text{-}T}$ is well-defined.
- Merge $\mathit{merge\text{-}T}$ is defined by well-founded recursion over the subterm relation on T_m:

$$\mathit{merge\text{-}T}\ (T_m\ x)\ (T_m\ y) = \mathit{map\text{-}option}\ T_m\ (\mathit{merge\text{-}F}\ \mathit{hash\text{-}T}\ \mathit{merge\text{-}T}\ x\ y)$$

  Congruence ensures that $\mathit{merge\text{-}F}$ calls $\mathit{merge\text{-}T}$ recursively only on smaller arguments.

Monotonicity and preservation of *blinding-respects-hashes* are proven by rule induction on $\mathit{blinding\text{-}of\text{-}T}$. Congruence, *blinding-of-on*, and *merge-on* are shown by structural induction on the argument that is constrained by A. ◀

Isabelle/HOL lacks the abstraction over type constructors necessary to formalize this theorem. Instead, we adopt an approach similar to Blanchette et al. [1]. We axiomatize a binary Merkle functor and carry out the construction and proofs for least fixpoints and composition, illustrating how the definition and proofs generalize to functors with several type arguments. The example ADS constructions in Section 3.6 then merely adapt these proofs to the concrete functors at hand.

3.5 Concrete Merkle functors

We now present concrete Merkle functors. They show that the class of Merkle functors is sufficiently large to be of interest. In particular, it contains all inductive datatypes (least fixpoints of sums of products). We have formalized all of the following.

- The discrete functor from Section 3.3 with hash operation *id* and the discrete blinding order $(=)$ is a nullary Merkle functor.
- Blindable positions $\mathit{blindable}_h$ and $\mathit{blindable}_m$ are a unary Merkle functor.
- Sums and products are binary Merkle functors. We set $\times_h = \times_m = \times$ and $+_h = +_m = +$. The hash operations *hash-prod* and *hash-sum* are the mappers *map-prod* and *map-sum*, respectively. The blinding orders *blinding-of-prod* and *blinding-of-sum* are the relators *rel-prod* and *rel-sum*. The merge operation *merge-of-prod* attempts to merge each component separately, while *merge-of-sum* can only merge left and left, or right and right values. (Formally, $\times_m$ and $+_m$ should take four type arguments. However, as sums and products do not themselves contain blindable positions, the type arguments $'a_h$ and $'b_h$ are ignored in inclusion proofs and we therefore omit them.)
- The function space $'a \Rightarrow\ 'b$ is a unary Merkle functor in the codomain. Like for sums and products, $\Rightarrow_h = \Rightarrow_m = \Rightarrow$ and no additional type arguments are added. Hashing is function composition and the blinding order is pointwise.

3.6 Case study: Merkle rose trees and Canton's transactions

Theorem 5 shows that all datatypes built from the Merkle functors in the previous section are Merkle functors. We apply the construction sketched in the proof to concrete datatypes that build on top of each other. For example, lists, rose trees, and Canton transactions are all Merkle functors. We prove that *'a list* is a Merkle functor with the help of an isomorphic data type that is the least fixpoint $\mu X.\ 1 + {'a} \times X$ and following the fixpoint construction of Theorem 5. We transfer the definitions and theorems to *list* using the transfer package [11]. Rose trees are then given by the datatype

datatype *'a rose-tree* = *Tree* ‹(*'a* × *'a rose-tree list*) *blindable*›

Applying the construction gives us Merkle rose trees with the corresponding operations and their properties. From here, it is only a small step to transactions in Canton. Views are isomorphic to Merkle rose trees where the data at the nodes is instantiated, i.e., composed, with the Merkle functor corresponding to *view-metadata blindable* × *view-data blindable*. Then, transactions compose the Merkle functor for *common-metadata blindable* × *participant-metadata blindable* × - *list* with views. We have lifted our machinery from these raw Merkle functors to the datatypes view_m and $\mathit{transaction}_m$ using the lifting and transfer packages [11].

4 Creating Inclusion Proofs

So far, given a tree-like data type $'t$, we showed how to systematically construct the corresponding type of ADSs $'t_h$ and their inclusion proofs $'t_m$. To make use of this construction in practice, we must also be able to create values of type $'t_m$ from values of type $'t$. As in the case of our composition and fixpoint theorem, HOL's lack of abstraction over type constructors makes it impossible to express this process in HOL in its full generality. Instead, we sketch how it works on rose trees, as these are the most general type of tree in terms of branching. The construction can be easily adapted for other kinds of trees.

There are three basic operations. Digesting, *digest*, returns the root hash for a source tree. Embedding, *embed-source-tree* returns the inclusion proof that proves inclusion of the whole tree. Finally, fully blinding, *blind-source-tree* returns the inclusion proof that proves no inclusion at all. Digesting and fully blinding conceptually do the same thing, but their return types ($'a_h\ \mathit{rose\text{-}tree}_h$ and $('a_m,\ 'a_h)\ \mathit{rose\text{-}tree}_m$) differ. As rose trees are parameterized by their node label type, digesting, embedding, and fully blinding take parameters which digest or embed the node labels. The expected properties hold: the embedded and fully blinded versions of the same source tree have the same hash, namely the digest of the source tree, and the former is a blinding of the latter.

The more interesting operations concern creating an inclusion proof for a subtree of a tree. For example, with Canton's hierarchical transactions, we would like to prove that a subtransaction is really part of the entire transaction. Such a proof consists of the subtree itself, together with a path connecting the tree's root to the subtree's root. As noticed by Seefried [17], this corresponds to a zipper [10] focused on the subtree. This enables simple manipulation of such proofs in a functional programming style, well-suited to HOL. We define operations to turn rose trees into zippers focused on the root, and zippers into their rose trees. Zippers can be defined both for source trees and inclusion proofs, and zippers on source trees can be turned into zippers on inclusion proofs. This allows us to easily model the messages that the initiator of a transaction sends in the first phase of Canton's commit protocol. The initiator constructs all zippers for the views in the transaction tree, and then turns each such zipper into an inclusion proof. Finally, the initiator merges each view proof with the proof from the zipper for the transaction metadata and ships it to the recipients.

5 Related Work

Miller et al. developed a lambda calculus with authentication primitives for generic tree structures [15]. The calculus was formalized in Isabelle/HOL by Brun and Traytel [3]. In the calculus, the programmer annotates the structures with authentication tags. Given a value of such a structure, and a function operating on it, their presented method automatically creates a correctness proof accompanying a result. The proof allows a verifier that holds only a digest of values with authentication tags (but not the values themselves) to check the function's result for correctness. The proof is a stream of inclusion proofs, one for each tagged value that the function operates on. Merging of inclusion proofs is not considered, although the streams can be optimized by sharing. Unlike Brun and Traytel [3] who use a deep embedding with the Nominal library, our embedding is shallow. Furthermore, our ADSs can provide inclusion proofs for multiple sub-structures simultaneously. However, we do not aim to derive generic correctness proofs for functions on the data structures.

Several other works formalize (binary) Merkle trees. White [18] formalized them as part of a Coq model of a cryptographic ledger, with the model tailored to the specific structure used. Our generic development can be instantiated to cover this structure. Yu et al. [19] use Merkle constructions on different binary trees to implement logs with inclusion and exclusion proofs. The constructions are proved correct using a pen-and-paper approach. The proved properties are then used in the Tamarin verification tool to analyze a security protocol. Ogawa et al [16] formalize binary Merkle trees as used in a timestamping protocol. They automatically verify parts of the protocol using the Mona theorem prover.

Seefried [17] observed that inclusion proofs in a Merkle tree correspond to Huet-style zippers [10], where the subtrees in zipper context have been replaced by the Merkle root hashes. McBride showed that zippers represent one-hole contexts [13]. In this analogy, our inclusion multi-proofs correspond to contexts with arbitrarily many holes.

6 Conclusion and Future Work

We have presented a modular construction principle for authenticated data structures over tree-shaped HOL datatypes (i.e., functors), and basic operations over these structures. The class of supported functors includes sums, products, and functions, and is closed under composition and least fixpoints. The supported operations are root hash computations and merging of inclusion proofs. We showed how to instantiate the construction to rose trees, as well as to real-world structures used in Canton, a Byzantine fault tolerant commit protocol.

The ongoing formalization of the Canton protocol will continue to test our abstractions and trigger further improvements. As noted earlier, ADSs not only improve storage efficiency, but also provide confidentiality. For example, Canton uses them to keep parts of a transaction confidential to a subset of the transaction's participants. However, reasoning about confidentiality is not straightforward. As hashing is injective, we can simply write *inv h* in HOL to invert hash functions. In fact, our current model does not even distinguish between the authenticated data structure and its digest because of this. A sound confidentiality analysis must therefore restrict the adversary using an appropriate calculus, e.g., a Dolev-Yao style deduction relation [8]. The analysis must take into account situations such as a Merkle tree node with two children with identical hashes; unblinding one child automatically unblinds the other. However, our representation distinguishes between the two, which might represent a problem. Another situation where this might be a problem is when merging inclusion proofs for commutative structures. One option is to consider Merkle functors as quotients with respect to a normalization function that collects all unblinding information and propagates the unblinding across the whole inclusion proof. The normalized inclusion proofs then serve as the canonical representatives. We have not yet worked out whether such a construction can still be modular and whether the quotients are still BNFs [9].

Moreover, our representation of hashes as terms makes hashing injective. While this is "morally equivalent" to standard cryptographic assumptions, an alternative (followed by [3]) would be to prove results about authentication as a disjunction: either the result holds, or a hash collision was found. The advantage of such a statement would be that hash collisions become explicit, which simplifies the soundness argument for the formalization. As is, nothing prevents us from conceptually "evaluating" the hash function on arbitrarily many inputs, which would not be cryptographically sound. To make hash collisions explicit, we must make hashes explicit, i.e., use a type like *bitstring*s instead of terms. We do not expect problems with extending our constructions to such a model, but it is unclear how severely the indirection through *bitstring*s impacts our proofs, in particular the Canton formalization.

References

1 Jasmin Christian Blanchette, Johannes Hölzl, Andreas Lochbihler, Lorenz Panny, Andrei Popescu, and Dmitriy Traytel. Truly modular (co)datatypes for Isabelle/HOL. In *Interactive Theorem Proving (ITP 2014)*, pages 93–110, 2014.

2 Sören Bleikertz, Andreas Lochbihler, Ognjen Marić, Simon Meier, Matthias Schmalz, and Ratko G. Veprek. A structured semantic domain for smart contracts. Computer Security Foundations poster session (CSF 2019), `https://www.canton.io/publications/csf2019-abstract.pdf`, 2019.

3 Matthias Brun and Dmitriy Traytel. Generic authenticated data structures, formally. In *Interactive Theorem Proving (ITP 2019)*, pages 10:1—10:18, 2019.

4 Canton: A private, scalable, and composable smart contract platform. `https://www.canton.io/publications/canton-whitepaper.pdf`, 2019.

5 Canton: Global synchronization beyond blockchain. `https://www.canton.io/`, 2020.

6 Corda: Open source blockchain platform for business. `https://www.corda.net/`, 2020.

7 Digital Asset. Daml programming language. `https://daml.com`, 2020.

8 D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

9 Basil Fürer, Andreas Lochbihler, Joshua Schneider, and Dmitriy Traytel. Quotients of bounded natural functors. In *Automated Reasoning (IJCAR 2020)*, 2020. To appear.

10 Gérard Huet. The zipper. *Journal of Functional Programming*, 7(5):549—554, 1997.

11 Brian Huffman and Ondřej Kunčar. Lifting and transfer: A modular design for quotients in Isabelle/HOL. In *Certified Programs and Proofs (CPP 2013)*, pages 131—-146, 2013.

12 Andreas Lochbihler and Ognjen Marić. Authenticated data structures as functors. *Archive of Formal Proofs*, April 2020. `http://isa-afp.org/entries/ADS_Functor.html`.

13 Conor McBride. The derivative of a regular type is its type of one-hole contexts, 2001.

14 Ralph C. Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology (CRYPTO 1987)*, pages 369–378, 1987.

15 Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi. Authenticated data structures, generically. In *Principles of Programming Languages (POPL 2014)*, pages 411—423, 2014.

16 Mizuhito Ogawa, Eiichi Horita, and Satoshi Ono. Proving properties of incremental Merkle trees. In *Automated Deduction (CADE 2005)*, pages 424–440, 2005.

17 Sean Seefried. Merkle proofs for free! Functional Programming Sydney, `http://code.ouroborus.net/fp-syd/past/2017/2017-04-Seefried-Merkle.pdf`, 2017.

18 Bill White. A theory for lightweight cryptocurrency ledgers. `https://github.com/input-output-hk/qeditas-ledgertheory`, 2015.

19 Jiangshan Yu, Vincent Cheval, and Mark Ryan. DTKI: a new formalized PKI with no trusted parties. *The Computer Journal*, 59(11):1695–1713, November 2016. arXiv: 1408.1023.

Mechanized Formal Model of Bitcoin's Blockchain Validation Procedures

Kristijan Rupic, Lovro Rozic, and Ante Derek

Faculty of Electrical Engineering and Computing, University of Zagreb
`kristijan.rupic@fer.hr` `lorozic33@gmail.com` `ante.derek@fer.hr`

Abstract

We present the first formal model of Bitcoin's transaction and blockchain data structures including the formalization of the blockchain validation procedures. Our formal model, though still a simplified representation of an actual Bitcoin blockchain, includes regular and coinbase transactions, segregated witnesses, relative and absolute locktime, the Bitcoin Script language expressions together with a denotational semantics, transaction fees and block rewards. We formally specify the details of consistency and validity checks performed when adding new blocks to the blockchain. We assume perfect cryptography and use the symbolic approach for modeling hash functions and digital signatures.

To demonstrate the utility of the model, we formally state and prove several essential properties of a consistent blockchain — transactions are unique, each coin can be spent at most once and the new value is only created through block rewards.The model and the proofs are largely independent of Bitcoin specific details and easily generalize to any cryptocurrency blockchain based on the Unspent Transaction Output (UTXO) paradigm.

We mechanize all the results using the Coq proof assistant.

1 Introduction

In the past decade, due to the popularity of Bitcoin [17] and other cryptocurrencies, as well as new applications such as smart contracts [10], blockchain systems have attracted significant attention from the scientific community. The blockchain systems implement *distributed ledgers* where the data and transaction integrity is enforced using cryptography and consensus mechanisms.

Despite the openness of the Bitcoin system, serious design and implementation flaws have been discovered over the years. For example, a simple design flaw made it possible to include two different *coinbase* transactions with the same transaction identifier (TXID) into the blockchain [2]. The flaw was subsequently fixed in two *Bitcoin Improvement Proposals*: BIP 30 [2] made the older of the two transactions unspendable and included explicit checks for uniqueness of TXID's, BIP 34 [3] mandated that coinbase transactions must include block height information, thereby fixing the design flaw. More recently (and more seriously), an implementation error in the transaction and block verification logic of the official Bitcoin client [5] made it possible for malicious miners to launch double-spending attacks.

In this paper, we build a formal model of Bitcoin's blockchain validation logic and we fully mechanize it using the Coq proof assistant [23]. We use the model to verify essential properties of a consistent blockchain including the absence of both flaws described above.

Instead of starting from scratch, we take the formal model of Bitcoin transactions by Atzei et al. [8] as the reference point for our formalization and mechanization efforts. We extend the model by adding the *blockchain* data structure containing blocks of transactions linked by hash pointers. Our model includes the complete treatment of coinbase transactions, the block height information as mandated by BIP34, transaction fees and block rewards. Finally, we model the blockchain validation procedures by formally specifying the sanity and consistency checks performed by Bitcoin clients when adding new blocks; we define the blockchain to be *consistent* if it passes the said validation procedures.

Contributions. Contributions of this paper are as follows:

1. We propose a fully mechanized model for Bitcoin transaction and the blockchain data structures. While simplified, the model includes many important details such as multi-signatures, segregated witnesses, absolute and relative locktimes, coinbase transactions, transaction fees and block rewards.
2. We define a denotational semantics for symbolic typed variant of Bitcoin Script language.
3. We define the sanity and consistency checks performed by clients when adding new blocks to the blockchain.
4. We demonstrate the utility of the model by giving machine-verified proofs for three essential properties of a consistent blockchain — same coin cannot be spent twice, transactions are unique, the total value of unspent coins is equal to the total value of block rewards.
5. We mechanize all the above results using the Coq proof assistant. [1]

We make a number of simplifying assumptions. First, we use the Dolev-Yao [14] model of cryptography where hash functions and digital signatures are abstract operations with perfect security properties. We simplify the Blockchain data structure by ignoring the Merkle trees that are normally used to include transactions and witnesses in block headers. Instead of a stack-based Script language and the corresponding execution model, we formalize the output scripts using an expression language with typed denotational semantics. Finally, many important aspects of the Bitcoin system such as the proof-of-work consensus mechanism, peer-to-peer network protocol, transaction and block discovery methods, etc. are out of scope of this work. Note that, there are efforts underway to mechanize those aspects of the Bitcoin system [21] — we view them as complementary to results presented in this paper.

We assume the reader is familiar with the Bitcoin system in general as well as the details of transaction and blockchain data structures including the notions of *inputs*, *outputs*, *witness* scripts and *coinbase transactions*. Due to space constraints, we deffer details for the several aspects of the formal model (e.g., the semantics of the script language expressions) as well as proofs to the extended version of the paper. We also refer the reader to the Coq artifacts.

Outline. Section 2 presents our model of Bitcoin transactions formalized using the Coq proof assistant. In Section 3 we give the formal model of the blockchain data structure. In Section 4 we use the model to provide machine-verified proofs for the essential properties of a consistent blockchain. In Section 5 we discuss the limitations of our model. We address related work in Section 6 and conclude in Section 7.

2 Formal Model of Bitcoin Transactions and Blockchain

We present a Coq model of the Bitcoin blockchain and the Bitcoin Script language. For now, we are primarily interested in transaction and blockchain consistency.

Notation. For some type τ we use τ^* to denote the type of lists of elements of type τ. We denote the empty list as $[]$ and the singleton list containing some element x by $[x]$. We use '+' to denote list concatenation, $|\cdot|$ to denote list length, and $\in$ to denote list membership. Dot notation is used to denote access to individual members of structures. For example, we

[1] We provide the Coq artifacts to reviewers as supplementary material (`http://www.zemris.fer.hr/~aderek/rrd-fmbc-artifacts.tar.gz`) and plan to release them as an open-source project after publication.

$$\texttt{Satoshi}, \texttt{Index}, \texttt{Time} \triangleq \mathbb{N} \tag{1}$$

$$\texttt{PK}, \texttt{SK} \triangleq \mathbb{N} \tag{2}$$

$$\mathit{is\_key\_pair} : \texttt{PK} \to \texttt{SK} \to \texttt{bool} \tag{3}$$

$$\texttt{Modifier} \triangleq \{aa, an, as, sa, sn, ss\} \tag{4}$$

Figure 1: Basic definitions, key pairs and hash flags

write $T.wit(i)$ to access the i-th index of the witness field of some transaction T. We will abbreviate $T.stub.inputs$ with $T.inputs$ (and similarly with other fields of transaction stubs). These notations might differ slightly from our Coq code but correspond to it in a one-to-one fashion.

2.1 The Transaction Model

We start out with a model of transactions and transaction histories, i.e., lists of transactions ordered by logical time. We model the Bitcoin Script language in order to provide an end-to-end model of transaction verification, although the proofs of various properties of our model could be made parametric with respect to a choice of the script language with relative ease, since their details tend to not affect higher-level properties.

As mentioned in the introduction, we use the symbolic approach when modeling cryptographic primitives. This allows us to simplify hashes of objects to only the objects themselves equipped with a decidable equality predicate, making the hash function essentially be the identity function which is injective and therefore also collision-resistant in a trivial way.

We begin by listing the basic definitions (Equation 2.1) which we will use throughout the rest of the formalization. Amounts of money (Satoshis, the name of the smallest Bitcoin denomination) and logical time in the system are both modeled as natural numbers for simplicity (1). Next, we define key pairs (2) for public-key digital signatures as trivial inductive types wrapping a value with decidable equality (in particular, a natural number) and we define a public and secret key to belong to the same pair if and only if they wrap equal values (3). We also define modifiers (4) corresponding to SIGHASH flags used in transaction signing [8].

Next, we need to define transactions (22). A regular transaction definition should consist of at least the following: a list of transaction inputs (16); a list of transaction outputs (17); a list of witness data associated with the inputs (24). Since we model SegWit [4], in our model we will distinguish between transactions and transactions paired with their respective witnesses depending on the context. The model of transactions also includes the absolute lock time (18) (nLockTime), which is a constraint on the earliest time the transaction can appear in a valid blockchain. While Bitcoin allows this to be either a block height or a UNIX timestamp depending on the range of the value [1], we only model some abstract, logical time. The extension of the model to cover both options Bitcoin allows is trivial. We also model coinbase transactions. They contain outputs but no inputs. These outputs represent the reward for mining of blocks and should be the sole supply of money in the system. They also contain their *block height*, i.e., the number of the block they are contained in in order to make them distinct as in BIP 34 [3].

Inputs (16) are references to outputs of other transactions, i.e., pairs of the referenced transaction and an index into its output list, along with a relative lock time which is another temporal constraint used in transaction verification. Unlike a Bitcoin implementation, this reference contains referenced transactions themselves instead of their hashes. Therefore, we require a decidable equality predicate on transactions, as well as an induction principle for its proof of correctness, more involved than ones Coq can automatically generate; we write an induction principle for transactions and their mutually inductive types manually.

A transaction output (17) consists of its value in Satoshis and a script (5) for the verification of attempts to redeem the output. The Bitcoin Script language is a stack-based language that is used to write output scripts that verify that the conditions for redeeming the output are met. A script takes a fixed number of inputs which depend on the commands used; these inputs are called the witness and a redeeming transaction must provide them. Following the work of Atzei et al. [8], we model the script language as an expression-based language instead as that allows us to easily specify denotational semantics for the scripts.

In a Bitcoin implementation all script values are just byte vectors at most 520 bytes long and their interpretation is made by the stack commands as either numbers, truth values, signatures, hashes etc. As we model hashes and signatures symbolically, we need our script input value type `StackValue` (9) to represent those possibilities as well, so we choose to impose a rudimentary type system on the values and their denotations that allows for integers (10), booleans (11), transaction signatures (13), and hashes of any type of value (12). As a transaction signature (26) is simply a wrapper for a secret key and a transaction "hash", a value will possibly contain transactions as well, making `StackValue` mutually inductive with transactions in our model (Figure 2).

The output script expression language is relatively simple. Most notable expression types are variables (6), constants of any `StackValue` (7), a multi-signature verification primitive (8) and several other arithmetic and comparison operations. We model it with an inductive type `Exp` (5) mutually inductive with `StackValue` and `TxStub` due to the fact that arbitrary `StackValues` can be contained as constants in the expressions, which is made necessary by our imposed type system in order to be able to meaningfully define arithmetic and comparison operations. The final result are three mutually inductive types (Figure 2) together with a rather complex induction principle.

The witnesses (24) are data associated with inputs passed as input to output scripts in order to verify the redeeming attempt. Note that it is impossible to sign the witnesses along with the rest of transaction due to the fact that usually the witness data needs to contain the transaction signature itself. The witnesses not being signed implies that they can be changed before being included in a block, changing the hash of the transaction with witnesses included, a problem known as transaction malleability. This was resolved by the implementation of a protocol upgrade called SegWit (Segregated Witness) introduced by BIP141 [4]. We account for these subtleties in our model by separating the witnesses from input data in our model as well. In implementations of SegWit the witnesses are moved outside transaction data structures into their own Merkle tree stored in the containing block's coinbase transaction. To be able to talk about transaction history consistency, we will sometimes have to associate transactions with their corresponding witnesses regardless of SegWit; to achieve this, we separate the transaction model into two layers of inductive types: the type `TxStub` (14) containing the transaction data save for the witnesses, and full transaction `Tx` (22) containing its stub and a list of witnesses (24). The transaction hash for input referencing purposes (TXID) is modeled by the the `TxStub` type.

2.2 Signature Verification and Output Redeeming

We now define our model of transaction signatures and their verification (Figure 3). A transaction signature is the SK-signed hash of a transaction with some fields disregarded in a way controlled by SIGHASH flags; in particular, some of the inputs are disregarded depending on the exact flags. We model hashes computed in this manner with the inductive type `TxStubHash` (25) wrapping

$$\texttt{Exp} : \texttt{Set} ::= \tag{5}$$
$$e\_var : \texttt{string} \to \texttt{Exp} \tag{6}$$
$$e\_const : \texttt{StackValue} \to \texttt{Exp} \tag{7}$$
$$e\_plus : \texttt{Exp} \to \texttt{Exp} \to \texttt{Exp}$$
$$e\_minus : \texttt{Exp} \to \texttt{Exp} \to \texttt{Exp}$$
$$e\_equal : \texttt{Exp} \to \texttt{Exp} \to \texttt{Exp}$$
$$e\_less : \texttt{Exp} \to \texttt{Exp} \to \texttt{Exp}$$
$$e\_if : \texttt{Exp} \to \texttt{Exp} \to \texttt{Exp} \to \texttt{Exp}$$
$$e\_length : \texttt{Exp} \to \texttt{Exp}$$
$$e\_hash : \texttt{Exp} \to \texttt{Exp}$$
$$e\_versig : \texttt{PK}^* \to \texttt{Exp}^* \to \texttt{Exp} \tag{8}$$
$$e\_abs\_after : \texttt{Time} \to \texttt{Exp} \to \texttt{Exp}$$
$$e\_rel\_after : \texttt{Time} \to \texttt{Exp} \to \texttt{Exp}$$

$$\texttt{StackValue} : \texttt{Set} ::= \tag{9}$$
$$sv\_int : \mathbb{Z} \to \texttt{StackValue} \tag{10}$$
$$sv\_bool : \texttt{bool} \to \texttt{StackValue} \tag{11}$$
$$sv\_hash : \texttt{StackValue} \to \texttt{StackValue} \tag{12}$$
$$sv\_sig : \texttt{TxStub} \to \texttt{SK} \to \texttt{Modifier} \to \texttt{Index} \to \texttt{StackValue} \tag{13}$$
$$\texttt{TxStub} : \texttt{Set} ::= \tag{14}$$
$$tx\_stub \; \{ \tag{15}$$
$$inputs : (\texttt{TxStub} \times \texttt{Index} \times \texttt{Time})^*; \tag{16}$$
$$outputs : (\texttt{Exp} \times \texttt{Satoshi})^*; \tag{17}$$
$$absLock : \texttt{Time} \; \} \tag{18}$$
$$coinbase \; \{ \tag{19}$$
$$block\_height : \mathbb{N} \; ; \tag{20}$$
$$outputs : (\texttt{Exp} \times \texttt{Satoshi})^* \; \} \tag{21}$$
$$\texttt{Tx} : \texttt{Set} ::= tx \; \{ \tag{22}$$
$$stub : \texttt{TxStub}; \tag{23}$$
$$witnesses : (\texttt{StackValue}^*)^* \; \} \tag{24}$$

Figure 2: Mutually inductive transaction, witness value and script definition.

$$\texttt{TxStubHash} ::= tx\_hash : \texttt{TxStub} \to \texttt{Modifier} \to \texttt{Index} \to \texttt{TxStubHash} \tag{25}$$
$$\texttt{Sig} ::= sig : \texttt{SK} \to \texttt{Modifier} \to \texttt{Index} \to \texttt{TxStub} \to \texttt{Sig} \tag{26}$$
$$ver : \texttt{PK} \to \texttt{Sig} \times \texttt{Modifier} \to \texttt{TxStub} \to \texttt{Index} \to \texttt{bool} \tag{27}$$
$$multi\_ver : \texttt{PK}^* \to (\texttt{Sig} \times \texttt{Modifier})^* \to \texttt{TxStub} \to \texttt{Index} \to \texttt{bool} \tag{28}$$

Figure 3: Signatures and routines for their verification.

the hashed transaction and the hashing flags, along with its decidable equality predicate that is based on transaction stub equality modulo hash flags. Signatures are represented by the inductive type `Sig` (26) wrapping everything a `TxStubHash` wraps, as well as the secret key. A signature needs to be paired with the hash flags used to compute it as they affect the result and are required for checking; this is implemented in Bitcoin by appending a byte denoting the hash flags to the signature and we model this explicitly by using `Sig` × `Modifier` even though we could introspect our inductive wrappers for their value.

We proceed to define single (27) and multiple (28) signature verification routines. We model successful signature verification with a public key using a simple check for pairedness of the given public key with the wrapped secret key with the function *is_key_pair* (3), and a check for hash equality by comparing both `TxStubHash` and the hash flags for equality; the verification succeeds if all comparisons do. Multiple signature verification tries to verify a list of signatures, in order, using an ordered list of public keys by repeatedly calling the single signature verification routine for each signature with successive public keys from the list until success; the whole routine succeeds if all signatures have been successfully verified.

We define a straightforward denotational semantics for the script language based on Atzei et al. [8]. We impose a type system onto the script language consisting of the same types as `StackValue`, as well as a bottom for failing computation or invalid types. We define the context of a witness $make\_context\ e\ T.wit(i)$ to be the mapping from variables $(free\_vars\ e)$ in order in which they first appear in a left to right traversal of the expression's syntax tree to the values in the witness. The denotation of a script expression depends on the redeeming transaction, the index of the redeeming input and the context constructed from the corresponding witness. We refer the reader to the Coq development for details due to space constraints.

Definition 1 (Script verification)**.** *We say a transaction T's i-th input verifies a script e if:*

$$verifies(T, i, e) \triangleq |free\_vars\ e| = |T.wit(i)| \wedge [\![e]\!]_{T,i,make\_context\ e\ T.wit(i)} = den\_bool\ \texttt{true}.$$

Definition 2 (Output redeeming)**.** *We say the j-th input of transaction T_2 at logical time t_2* redeems *the i-th output of transaction T_1 at logical time t_1 for a value of v Satoshis if:*

$$\begin{aligned} & redeems(T_1, i, t_1, v, T_2, j, t_2) \triangleq \\ (i) \quad & \exists\ relLock\ e\ v_1, T_2.inputs(j) = (T_1, i, relLock) \ \wedge\ T_1.outputs(i) = (e, v_1) \ \wedge \\ (ii) \quad & T_2.absLock \leq t_2 \ \wedge\ t_1 + relLock \leq\ t_2 \ \wedge \\ (iii) \quad & v \leq v_1 \ \wedge \\ (iv) \quad & verifies(T_2, j, e) \end{aligned}$$

3 Blockchain Model and Consistency

We begin our model of the Bitcoin blockchain by first considering transaction histories and their consistency. We then define our model of the blockchain and its consistency by requiring that the transaction history encoded by the blockchain be consistent, among other things.

For Bitcoin to function as a currency, it is crucial to control the way in which money is created. Only coinbase transactions should increase the total sum of money in the system. However, if a transaction output was to be spent more than once, it would essentially act as duplicated money. Therefore, it is necessary to ensure that transaction outputs can be spent at most once. Transactions attempting to spend an already spent output, or spend an unspent output multiple times at once must be disallowed in a consistent transaction history. We provide a formal definition of the transaction history consistency predicate that enforces this and certain other conditions necessary for a history to be considered valid. We later prove that this property indeed implies that no double spending of transaction outputs is happening within a consistent history, as well as that the total sum of unspent transaction outputs never exceeds supply, i.e., the sum of coinbase outputs.

We define a transaction history (29) as a list of transactions with witnesses and the logical time at which they occur. We also define the notions of spent and unspent transaction outputs; an output at index i of a transaction T_1 in the blockchain is *unspent* in a history TH if there is no transaction anywhere in TH that has an input (T_1, i), whereas an output is *spent* in TH if there such a transaction and input exist. We define functions $STXO$ and $UTXO$ (33, 31) on histories that compute respectively the list of spent and unspent outputs, with outputs represented as pair of the containing transaction and the output's index. We also formally prove the obvious fact that every output of every transaction in a blockchain is either spent or unspent. We define the sum of values of inputs (34) and outputs (35) of a transaction, as well as the sum of values of all UTXO-s (36) and all coinbase outputs (37) in a transaction history which should represent the total supply of money in a transaction history following some consistency rules which we will define. We define *coinbase_height* to be the number of coinbase transactions in a transaction history; note that this is going to be equal to the block height, but is formalized independently.

$$
\begin{aligned}
&\texttt{TxHistory} \triangleq (\texttt{Tx} \times \texttt{Time})^* && (29) \qquad & &\mathit{sum\_inputs} : \texttt{TxStub} \to \texttt{Satoshi} && (34)\\
&\mathit{UTXO} : \texttt{TxHistory} && (30) \qquad & &\mathit{sum\_outputs} : \texttt{TxStub} \to \texttt{Satoshi} && (35)\\
&\quad \to (\texttt{TxStub} \times \texttt{Index})^* && (31) \qquad & &\mathit{UTXO\_value} : \texttt{TxHistory} \to \texttt{Satoshi} && (36)\\
&\mathit{STXO} : \texttt{TxHistory} && (32) \qquad & &\mathit{coinbase\_value} : \texttt{TxHistory} \to \texttt{Satoshi} && (37)\\
&\quad \to (\texttt{TxStub} \times \texttt{Index})^* && (33) \qquad & &\mathit{coinbase\_height} : \texttt{TxHistory} \to \mathbb{N} && (38)
\end{aligned}
$$

Figure 4: Transaction history model

Using the work of Atzei et al. [8] as a reference point, we define *transaction history consistency* (4) inductively by requiring that each consistent transaction history is formed by a sequence of *consistent updates* (3) each extending the history by a single transaction in a way that enforces the necessary invariants.

Definition 3 (Consistent update for transaction histories)**.**

$$
\begin{aligned}
&\mathit{is\_consistent\_update}(TH, T, t) \triangleq\\
&(i)\quad \exists\ \mathit{block\_height}\ \mathit{outputs},\ T.\mathit{stub} = \mathit{coinbase}\ \mathit{block\_height}\ \mathit{outputs}\ \wedge\\
&(ii)\quad \forall\ TH'\ T'\ t', TH = TH' + [(T', t')] \implies t' \leq t\ \wedge\\
&(iii)\quad T.\mathit{block\_height} = \mathit{coinbase\_height}\ TH\\
&\vee\\
&(iv)\quad \mathit{sum\_inputs}(T) \geq \mathit{sum\_outputs}(T)\ \wedge\\
&(v)\quad \forall\ TH'\ T'\ t', TH = TH' + [(T', t')] \implies t' \leq t\ \wedge\\
&(vi)\quad T.\mathit{inputs} \neq []\ \wedge \forall\ i\ j\ T'_i\ o_i\ r_i\ T'_j\ o_j\ r_j, i \neq j\ \wedge\\
&\quad T.\mathit{inputs}(i) = (T'_i, o_i, r_i)\ \wedge\ T.\mathit{inputs}(j) = (T'_j, o_j, r_j) \implies (T'_i, o_i) \neq (T'_j, o_j)\ \wedge\\
&(vii)\quad \forall\ j\ T'\ o\ r\ t'\ s\ v, (T', t') \in TH\ \wedge T.\mathit{inputs}(j) = (T', i, r)\ \wedge\ T'.\mathit{outputs}(i) = (s, v)\\
&\quad \implies (T', i) \in \mathit{UTXO}(TH)\ \wedge\ \mathit{redeems}(T', i, t', v, T, j, t)
\end{aligned}
$$

Definition 4 (Transaction history consistency)**.**

$$
\begin{aligned}
&\mathit{tx\_history\_consistent}(TH) ::=\\
&\quad \mathit{bc\_empty} : TH = [] \to \mathit{tx\_history\_consistent}(TH)\\
&\quad\quad \mathit{bc\_cons} : \forall\ TH'\ T\ t,\ TH = TH' + [(T, t)] \to \mathit{tx\_history\_consistent}(TH')\\
&\quad\quad\quad \to \mathit{consistent\_update}(TH', T, t') \to \mathit{tx\_history\_consistent}(TH)
\end{aligned}
$$

Now we define a *blockchain* (Figure 5, 39) as an inductive type. Hash pointers to blocks are, as before, represented by the blocks themselves. As we do not deal with proof-of-work or consensus, the only contents of a block are the pointer to the previous block (42), the transactions (43) and witnesses (44) of the block, and the block's timestamp (45). Transactions and witnesses are both represented as lists instead of Merkle trees, but are separated according to SegWit. We also define *block_height* (46) to be the number of blocks in the blockchain, and *bc_to_tx_history* (47) to be a function that flattens a blockchain into the transaction history it represents by concatenating lists of transactions paired with their respective witnesses. We define the *block reward* (49), a function from block height of the block to be minted to the base value to include in the block's coinbase transaction; and *transaction_fees* (48) to be the sum of the differences between input and output value for each transaction in a list.

$$\begin{array}{rl}
\texttt{Blockchain} ::= & (39)\\
\mathit{Empty} & (40)\\
\mathit{Block}\ \{ & (41)\\
\mathit{prevBlock} : \texttt{Blockchain}; & (42)\\
\mathit{transactions} : \texttt{TxStub}^*; & (43)\\
\mathit{witnesses} : ((\texttt{StackValue}^*)^*)^*; & (44)\\
\mathit{timestamp} : \texttt{Time}\ \} & (45)
\end{array}$$

$$\begin{array}{rl}
\mathit{block\_height} : \texttt{Blockchain} \to \mathbb{N} & (46)\\
\mathit{bc\_to\_tx\_history} : \texttt{Blockchain} \to \texttt{TxHistory} & (47)\\
\mathit{transaction\_fees} : \texttt{TxStub}^* \to \texttt{Satoshi} & (48)\\
\mathit{block\_reward} : \mathbb{N} \to \texttt{Satoshi} & (49)
\end{array}$$

Figure 5: Blockchain model

The definitions of consistent updates of blockchains by blocks and consistent blockchains are analogous to the definitions for transaction histories.

Definition 5 (Consistent update for blockchains)**.** *A blockchain B is consistently updated with a new block containing $(\mathit{transactions}, \mathit{witnesses}, \mathit{timestamp})$ when*

1. *transactions list contains exactly one coinbase transaction CB as the first transaction*
2. *$CB.\mathit{block\_height} = \mathit{block\_height}\ B$*
3. *$\mathit{sum\_outputs}\ CB = \mathit{block\_reward}\ (\mathit{block\_height}\ B) + \mathit{transaction\_fees}\ \mathit{transactions}$*
4. *$\mathit{tx\_history\_consistent}\ (\mathit{bc\_to\_tx\_history}$ $(\mathit{Block}\ B\ \mathit{transactions}\ \mathit{witnesses}\ \mathit{timestamp}))$*

Definition 6 (Blockchain consistency)**.** *We define the consistency of a blockchain inductively.*

- *An Empty blockchain is consistent.*
- *A blockchain B with a block appended is consistent whenever B was consistent, the length of the block's transactions and witnesses lists is equal, and the appended block consistently updates B.*

4 Formally Verified Blockchain Properties

With all the definitions in place, we move on to state several important properties of consistent transaction histories and blockchains, which we have proven in our Coq development. Here we list only a part of the development due to space constraints; it can be seen in full along with the proofs in the accompanying materials.

First, we prove that blockchain consistency implies the consistency of the transaction history it stores. This follows directly from the definition of consistent updates with blocks applied to the last block in the chain, if any. This result allows us to reason about consistent transaction histories instead of blockchains, which can be more convenient e.g., when proving the impossibility of double spending in a consistent blockchain (and transaction history).

Theorem 1 (Blockchain consistency implies transaction history consistency)**.** *Let B be a consistent blockchain. Then $\mathit{bc\_to\_tx\_history}\ B$ is a consistent transaction history.*

Note that the definition of a transaction history does not order the transactions according to output spending. In a consistent transaction history, however, every transaction input refers to an output of a transaction earlier in the history, which we proved as a lemma.

The first key proprety of the blockchain we consider is the impossibility of spending the same output multiple times.

Theorem 2 (No double spending)**.** *Let B be a consistent blockchain, and TH be its (consistent) transaction history. Let T_i and T_j be two transactions in TH at indices i, j respectively. Then*

$$\forall k_i\ k_j,\ (T_i.inputs(k_i) = (T'_i, l_i, t_{rl,i}) \wedge T_j.inputs(k_j) = (T'_j, l_j, t_{rl,j}) \wedge (i, k_i) \neq (j, k_j)) \\ \implies (T'_i, l_i) \neq (T'_j, l_j).$$

Another key property of consistent transaction histories is that transactions identifiers are unique. While in reality we have to allow for the noninjectivity of hashes, in our model transactions are wholly unique within consistent histories.

Theorem 3 (Transaction uniqueness)**.** *Let B be a consistent blockchain, and TH be its (consistent) transaction history. Let txs be the list of* `TxStubs` *in the history (i.e., TH with timestamps and witnesses removed). Let T_i and T_j be transactions at indices i, j in txs, respectively. If $T_i = T_j$, then $i = j$.*

In the remainder of this section we consider properties of the total supply of money in the system. This should be equal to the sum of all coinbase output values, however it is also allowed to be smaller than that due to the presence of transaction fees.

Theorem 4 (Coinbase value bounds UTXO value above)**.** *Let TH be a consistent transaction history. Then*

$$UTXO\_value\ TH \leq coinbase\_value\ TH.$$

The following theorem illustrates the fact that only UTXO-s may be used as transaction inputs quantitatively. The proof follows from the definition of consistent updates.

Theorem 5 (UTXO value bounds input value sum)**.** *Let $TH + [(T, t)]$ be a consistent transaction history. Then*

$$sum\_inputs\ (stub\ T) \leq UTXO\_value\ TH.$$

The final theorem is a strenghtening of (4) shows that supply is exactly controlled by block rewards. It boils down to proving that transaction fees are properly collected in the coinbase transaction outputs of each block.

Theorem 6 (Total block reward equals UTXO value)**.** *Let B be a consistent blockchain, and $TH = bc\_to\_tx\_history\ B$. Then:*

$$UTXO\_value\ TH = \sum_{b=0}^{block\_height\ B-1} block\_reward\ b.$$

5 Limitations

Here we briefly discuss the limitations of our model and compare it to the Bitcoin client.

Since we use the symbolic model for digital signatures and hash functions, we are unable to prove the desired properties in the computational model of cryptography. Of course, we are also unable to extract the code for a verified client. We can overcome the latter by reusing Coq models of the cryptographic primitives (e.g. [7] for the SHA256 hash function). As for the former, since we are not concerned with the proof-of-work verification, we only rely on hash functions for data integrity and only need their collision resistance property. In the computational model, we could verify the properties under the assumption no collision occurred anywhere in that blockchain. For modeling properties of digital signatures in Coq we could attempt to use the toolset of the Foundational Cryptography Framework [20]. Alternatively, we could try to model the system within the universally composable (UC) security framework and replace the digital signature implementation with an ideal functionality similarly to the approach taken in [12] to develop mechanized analysis of a key echange protocol.

The blockchain verification procedures are currently modeled mostly as first order inductive predicates rather than decidable routines and, hence, cannot be used to extract verified code. We plan to address this by writing the missing decision routines to generates proofs or disproofs of our propositions as well as routines that parse our model from serialized data, which would give us verified extractable blockchain validation code.

Comparison with the official Bitcoin client. First, we do not attempt to model several important aspects of the validation logic, since we do not consider them to be relevant to the correctness properties we wished to tackle first. Most notably, we omit proof-of-work verification and the corresponding data fields from the model. Transactions and blocks in our model do not have version numbers accounting for protocol updates. We do not enforce block and transaction size limits, coinbase maturity and we do not reject transactions with absurdly high fees.

We make a number of technical choices that result in a simpler formal model and diverge from the Bitcoin client. For example, we have explicit coinbase transactions while in the Bitcoin client coinbase transactions are stored in the same data structure and are distinguished by a single input field with the zero hash pointer. We feel that addressing these differences is a technical matter, albeit tedious and time consuming.

In our model, only transactions with segregated witnesses are supported, while the Bitcoin client additionally supports legacy transactions where the witness is a part of the transaction's input field. There are several other examples of extensions where both current and legacy features are supported. Moreover, these are almost always implemented in a backward-compatible manner. From a consensus perspective it is desirable that the blockchain verification procedures are updated by a *soft-fork* — old nodes *must* recognize the new blocks as valid. Hence, new features often need to be *hacked* into the existing protocol in order to satisfy the old validation procedures (e.g., see the "segregated witness" implementation [4]).

We feel that the multitude of supported options along with backwards-compatible implementations present the most significant challenge for building a complete mechanized formal model that is faithful to the wire-level protocol. Hence, more research is needed to produce methods of building and using such models without the exploding complexity.

6 Related Work

Bitcoin and similar systems have received a lot of attention in the scientific community in recent years with many attempts to formally specify and verify various aspects of blockchain systems.

Formal treatment of the Bitcoin system. First, we give an overview of pen-and-paper formal models aimed at specification and verification of various aspects of the Bitcoin system.

In [8], Atzei et al. give a formal model of Bitcoin transactions that we use a starting point for our formalization and mechanization efforts. The model includes transaction and blockchain data structures, as well as the semantics for the Bitcoin Script language. The model is used to formally prove "well-formedness" properties of the Bitcoin blockchain including the impossibility of double-spending. In contrast to a simplified "linked list" model of [8], we fully model blocks and the blockchain including coinbase transactions in each block, block height information, block rewards and transaction fees. For transactions themselves, our models are different in several details where we try to be closer to the behavior of the Bitcoin client. Most notably, in [8] segregated witnesses are a part of the transaction structure, while we store them in blocks, independently of transactions. Mechanization using the Coq proof assistant forces us to carefully specify all the details of the model. For example, mutually inductive definitions have to be explicitly taken into account. Similarly, we need to explicitly state and prove many assumptions that are implicit in [8], such as the temporal properties of spent outputs and explicit encoding of witnesses and hash functions. We replicate the no-double-spent result given in [8] but in a more general setting (with a blockchain data structure) and with a proof that is machine-verified. More importantly, we prove two additional properties of a consistent blockchain.

Cachin et al. give an alternative model for the semantics of blockchain transactions by using directed acyclic graphs to abstract the interactions of an incoming transaction with the blockchain [11]. They provide a general blockchain model which they instantiate to to Bitcoin, Ethereum and Hyperledger Fabric systems.

Formal models of the Bitcoin Script language have also been an area of active research. In [9], the model of [8] is applied to development of a high-level domain specific language which then compiles into Bitcoin Script language, with the goal of systematically analyzing actual smart contracts proposed by researchers and Bitcoin developers. In [16] autors formalize the Bitcoin Script language with the goal of automatically finding inputs that satisfy a given script.

Finally, formal pen-and-paper treatments of Bitcoin's consensus mechanism include [15] where the focus is on quantifying the *quality* of the blockchain system by determining how many adversarial blocks are expected on the blockchain; and [13] where the authors work out the probability of a successful double-spending attack (assuming some nodes are malicious) and use the UPPAAL model checker to verify the results.

Consensus mechanization. In [21], Pirlea and Sergey focus on mechanizing protocols and data structures necessary for establishing distributed consensus in blockchain systems. They formally prove a form of eventual consistency in a network, while precisely characterizing all assumptions on implementations of underlying security primitives. In [22], authors build and mechanize a probabilistic model of blockchain consensus with the eventual goal of stating and proving probabilistic security properties in a Byzantine setting. Other efforts towards automated verification of blockchain consensus mechanisms include [18, 19] that focus on the proposed proof-of-stake mechanism for the Ethereum system. All above efforts use the Coq proof assistant.

7 Conclusions and Future Work

In this paper, we have presented a Coq formalization for the Bitcoin's blockchain validation procedures including the models of basic data structures of the Bitcoin blockchain system and the denotational semantics for the typed variant of the Bitcoin Script language. We have used the model to provide machine-verified proofs for three essential properties of a consistent blockchain: impossibility of double-spending, uniqueness of transactions and that cryptocurrency value is created only through block rewards.

In the future, we are going to discharge the number of simplifying assumptions and attempt to further bridge the gap between the abstract model and the reference client. In particular, we plan to model Merkle trees and use them to store transactions and witnesses in blocks. We also plan to make segregated witnesses optional and investigate the interaction between different types of transactions. More generally, we wish to investigate the scenarios where validity checks are updated. This will enable us to formally model the notion of soft-forks and evaluate proposed changes to the Bitcoin protocol such as spending rules based on Taproot, Schnorr signatures, and Merkle branches [6]. Finally, we will investigate the possibility of integrating our efforts with the Toychain system of Pirlea and Sergey [21].

References

[1] Bitcoin documentation. https://en.bitcoin.it/wiki/Protocol_documentation, 2010.

[2] Bitcoin improvement proposal 30: Duplicate transactions. bip-0030.mediawiki, 2012.

[3] Bitcoin improvement proposal 34: Block v2, height in coinbase. bip-0034.mediawiki, 2012.

[4] Bitcoin improvement proposal 141: Segregated witness (consensus layer). bip-0141.mediawiki, 2015.

[5] Double spending in bitcoin clients. CVE-2018-17144, 2018.

[6] Bitcoin improvement proposal 341: Segwit version 1 spending rules. bip-0341.mediawiki, 2020.

[7] A. W. Appel. Verification of a cryptographic primitive: Sha-256. *ACM Trans. Program. Lang. Syst.*, 37(2), Apr. 2015. ISSN 0164-0925. URL https://doi.org/10.1145/2701415.

[8] N. Atzei, M. Bartoletti, S. Lande, and R. Zunino. A formal model of bitcoin transactions. *IACR Cryptology ePrint Archive*, 2017:1124, 2017.

[9] N. Atzei, M. Bartoletti, T. Cimoli, S. Lande, and R. Zunino. Sok: Unraveling bitcoin smart contracts. In *Principles of Security and Trust*, pages 217–242, Cham, 2018. Springer International Publishing.

[10] V. Buterin. A next-generation smart contract and decentralized application platform. 2014. White-paper.

[11] C. Cachin, A. D. Caro, P. Moreno-Sanchez, B. Tackmann, and M. Vukolic. The transaction graph for modeling blockchain semantics. *IACR Cryptology ePrint Archive*, 2017:1070, 2017.

[12] R. Canetti, A. Stoughton, and M. Varia. Easyuc: Using easycrypt to mechanize proofs of universally composable security. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, pages 167–16716, June 2019.

[13] K. Chaudhary, A. Fehnker, J. van de Pol, and M. Stoelinga. Modeling and verification of the bitcoin protocol. In *Proceedings of the Workshop on Models for Formal Analysis of Real Systems (MARS 2015)*, Electronic Proceedings in Theoretical Computer Science, pages 46–60, Australia, 11 2015. Open Publishing Association.

[14] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, March 1983.

[15] J. Garay, A. Kiayias, and N. Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Advances in Cryptology - EUROCRYPT 2015*, pages 281–310, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[16] R. Klomp and A. Bracciali. On symbolic verification of bitcoin's SCRIPT language. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pages 38–56, Cham, 2018. Springer International Publishing.

[17] S. Nakamoto. Bitcoin: a peer-to-peer electronic cash system. 2008.

[18] K. Palmskog, M. Gligoric, L. Pena, B. Moore, and G. Rosu. Verification of casper in the coq proof assistant. 2018. Technical report.

[19] K. Palmskog, M. Gligoric, L. Pena, B. Moore, and G. Rosu. Verifying finality for blockchain systems. In *CoqPL'19*, 2019.

[20] A. Petcher and G. Morrisett. The foundational cryptography framework. In *Principles of Security and Trust*, pages 53–72, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[21] G. Pirlea and I. Sergey. Mechanising blockchain concensus. In *7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM New York, 2018.

[22] I. Sergey and K. Gopinathan. Towards mechanising probabilistic properties of a blockchain. In *CoqPL'19*, 2019.

[23] T. C. D. Team. The coq proof assistant, version 8.10.0, Oct. 2019. URL https://doi.org/10.5281/zenodo.3476303.

Towards Verifying the Bitcoin-S Library

Ramon Boss, Kai Brünnler, and Anna Doukmak

Bern University of Applied Sciences, CH-2501 Biel, Switzerland
`ramon.boss@outlook.com`, `kai.bruennler@bfh.ch`,
`anna.doukmak@gmail.com`

Abstract. We try to verify properties of the bitcoin-s library, a Scala implementation of parts of the Bitcoin protocol. We use the Stainless verifier which supports programs in a fragment of Scala called *Pure Scala*. Since bitcoin-s is not written in this fragment, we extract the relevant code from it and perform a series of equivalent transformations until we arrive at code that we successfully verify. In that process we find and fix two bugs in bitcoin-s.

Keywords: Bitcoin · Scala · bitcoin-s · Stainless.

1 Introduction

For software handling cryptocurrency, correctness is clearly crucial. However, even in very well-tested software such as Bitcoin Core, serious bugs occur. The most recent example is the bug found in September 2018 [10] which essentially allowed to arbitrarily create new coins. Such software is thus a worthwhile target for formal verification. In this work, we set out to verify properties of the bitcoin-s library with the Stainless verifier.

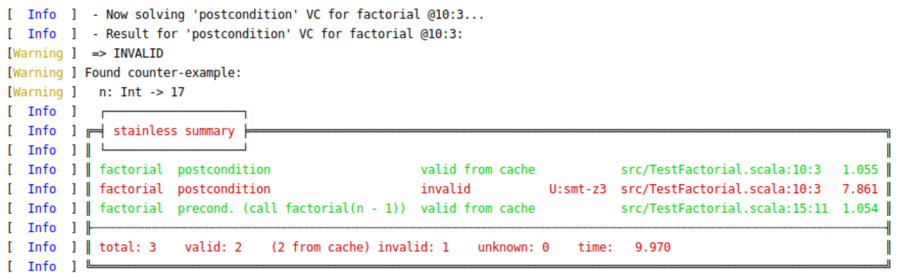
The Bitcoin-S Library. The bitcoin-s library is an implementation of parts of the Bitcoin protocol in Scala [11,12]. In particular, it allows to serialize, deserialize, sign and validate Bitcoin transactions. The library uses immutable data structures and algebraic data types but is not specifically written with formal verification in mind. According to the website, the library is used in production, handling significant amounts of cryptocurrency each day [11].

The Stainless Verifier. Stainless is the successor of the Leon verifier and is developed at EPF Lausanne [2,14,1]. It is intended to be used by programmers without training in formal verification. To facilitate that, it accepts specifications written in the programming language itself (Scala). Also, it focusses on counterexample finding in addition to proving correctness. Counterexamples are useful to programmers while correctness proofs are not – correctness is obvious or does not hold, and often both at the same time.

The example in Figure 1 adapted from the Stainless documentation [8] shows how the verifier is used. Note how a precondition is specified using `require` and a postcondition using `ensuring`. Our function does not satisfy the specification. An overflow in the 32-bit integer type leads to a negative result for the input 17,

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

Fig. 1. Factorial function with specification



```
[  Info  ]  - Now solving 'postcondition' VC for factorial @10:3...
[  Info  ]  - Result for 'postcondition' VC for factorial @10:3:
[Warning ]  => INVALID
[Warning ] Found counter-example:
[Warning ]   n: Int -> 17
[  Info  ]   ┌─────────────────┐
[  Info  ] ╔═╡ stainless summary ╞══════════════════════════════════════════════════════════════════════════╗
[  Info  ] ║ └─────────────────┘                                                                            ║
[  Info  ] ║ factorial  postcondition                      valid from cache           src/TestFactorial.scala:10:3   1.055 ║
[  Info  ] ║ factorial  postcondition                      invalid          U:smt-z3  src/TestFactorial.scala:10:3   7.861 ║
[  Info  ] ║ factorial  precond. (call factorial(n - 1))  valid from cache           src/TestFactorial.scala:15:11  1.054 ║
[  Info  ] ╟┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄╢
[  Info  ] ║ total: 3    valid: 2    (2 from cache) invalid: 1    unknown: 0    time:   9.970                  ║
[  Info  ] ╚══════════════════════════════════════════════════════════════════════════════════════════════════╝
```

Fig. 2. Stainless output for the factorial function

as Stainless reports in Figure 2. Changing the type `Int` to `BigInt` will result in a successful verification.

The Pure Scala Fragment. The Scala fragment supported by Stainless comprises algebraic data types in the form of abstract classes, case classes and case objects, objects for grouping classes and functions, boolean expressions with short-circuit interpretation, generics with invariant type parameters, pattern matching, local and anonymous classes and more. In addition to Pure Scala Stainless also supports some imperative features, such as using a (mutable) variable in a local scope of a function and while loops. They turn out not to be relevant for our current work.

What will turn out to be more relevant for us are the Scala features which Stainless does not support, such as: inheritance by objects, abstract type members, and inner classes in case objects. Also, Stainless has its own library of some core data types and functions which are mapped to corresponding data types and functions inside of the SMT solver that Stainless ultimately relies on. Those data types in general do not have all the methods of the Scala data types. For example, the `BigInt` type in Scala has methods for bitwise operations while the `BigInt` type in Stainless does not.

Outline and Properties to Verify. In the next section we try to verify the property that a regular (non-coinbase) transaction can not generate new coins. We call it the *no-inflation property*. Trying to verify it, we uncover and fix a bug in the bitcoin-s library. We then find that there is too much code involved that lies outside of the supported fragment to currently make this verification feasible. So we turn to a simpler property to verify. The simplest possible property we can think of is the fact that adding zero satoshis to a given amount of satoshis yields the given amount of satoshis. We call it the *addition-with-zero property* and we try to verify it in Section 3. Here as well we see that a significant part of the code lies outside of the supported fragment. We perform a series of equivalent transformations on it until we arrive at code that we successfully verify. In that process we find and fix a second bug in bitcoin-s.

2 The No-Inflation Property

The `checkTransaction` function shown in Figure 3 is crucial for the verification of the no-inflation property. Given a transaction it returns true if some basic checks succeed, otherwise false. For example, one of those checks is that both the list of inputs and list of outputs need to be non-empty.

To better understand the validation of a transaction in bitcoin-s, it is useful to review how transactions are represented and created.

Creating a Transaction. To create a transaction, we first need some coins – an unspent transaction output. We could load an actual unspent transaction output from the bitcoin network, but we create one manually in order to see this process. So we first create an (invalid) transaction with one output in Figure 4.

We first create a keypair, then a lock script with its public key, then the amount of satoshis, then a transaction output (`utxo`) for that amount and locked with that script. Finally we create a transaction with that output and no inputs. Of course, that is not a valid transaction, because it creates coins out of nothing. In particular, `checkTransaction(prevTx)` returns false, simply because the list of inputs is empty.

Now that we have a transaction output, we create a transaction to spend it in Figure 5. First, we need a reference to an output of a previous transaction, here called `outPoint`. Second, we add some information on how to spend that output, in particular, how to sign the transaction. Now we assemble the list of unspent transaction outputs (`utxos`), in our case just one.

We then set the amount of satoshis that we want to spend. The `Int64` class aims to emulate a C data type in Scala, and we will look at it more closely in the next section.

We then create a lock script (`destinationSPK`) to receive the coins, create our list of transaction outputs (`destinations`), define the fee rate and set some bitcoin network parameters.

Now we create a transaction builder with those data and we tell it to start signing the transaction in line 34.

Finally, we get the actual signed transaction. We could serialize it and send it to the Bitcoin network. We can also pass it to the checkTransaction function, which will return true.

A Bug in the checkTransaction Function. Note lines 15-17 in Figure 3. Here, the value `prevOutputTxIds` gathers a list of all transaction identifiers referenced

```scala
def checkTransaction(transaction: Transaction): Boolean = {
  val inputOutputsNotZero =
    !(transaction.inputs.isEmpty || transaction.outputs.isEmpty)
  val txNotLargerThanBlock =
    transaction.bytes.size < Consensus.maxBlockSize
  val outputsSpendValidAmountsOfMoney =
    !transaction.outputs.exists(o =>
      o.value < CurrencyUnits.zero || o.value > Consensus.maxMoney)

  val outputValues = transaction.outputs.map(_.value)
  val totalSpentByOutputs: CurrencyUnit =
    outputValues.fold(CurrencyUnits.zero)(_ + _)
  val allOutputsValidMoneyRange =
    validMoneyRange(totalSpentByOutputs)
  val prevOutputTxIds = transaction.inputs.map(_.previousOutput.txId)
  val noDuplicateInputs =
    prevOutputTxIds.distinct.size == prevOutputTxIds.size

  val isValidScriptSigForCoinbaseTx = transaction.isCoinbase match {
    case true =>
      transaction.inputs.head.scriptSignature.asmBytes.size >= 2 &&
        transaction.inputs.head.scriptSignature.asmBytes.size <= 100
    case false =>
      !transaction.inputs.exists(
        _.previousOutput == EmptyTransactionOutPoint)
  }
  inputOutputsNotZero && txNotLargerThanBlock &&
  outputsSpendValidAmountsOfMoney && noDuplicateInputs &&
  allOutputsValidMoneyRange && noDuplicateInputs &&
  isValidScriptSigForCoinbaseTx
}
```

Fig. 3. The checkTransaction function

```scala
val privKey = ECPrivateKey.freshPrivateKey
val creditingSPK = P2PKHScriptPubKey(pubKey = privKey.publicKey)

val amount = Satoshis(Int64(10000))

val utxo = TransactionOutput(currencyUnit = amount, scriptPubKey =
    creditingSPK)

val prevTx = BaseTransaction(
  version = Int32.one,
  inputs = List.empty,
  outputs = List(utxo),
  lockTime = UInt32.zero
)
```

Fig. 4. Creating a transaction output to spend

by the inputs of the current transaction. If the size of this list is the same as the size of this list with duplicates removed, we know that no transaction has been referenced twice. This prevents a transaction from spending two different outputs of the same previous transaction. The check is too strict: `checkTransaction` returns false for valid transactions.

The fix is simple: we perform the duplicate check on the `TransactionOutPoint` instances instead of on their transaction identifiers. Note that `TransactionOutPoint` is a case class and thus its notion of equality is just what we need: equality of of both the transaction identifier and the output index.

Specifically, we replace lines 15-17 as follows:

```scala
15  val prevOutputs = transaction.inputs.map(_.previousOutput)
16  val noDuplicateInputs =
17    prevOutputs.distinct.size == prevOutputs.size
```

We submitted this fix together with a corresponding unit test to the bitcoin-s project in a pull request, which has been merged [5].

An Attempt at Verification. Naively trying Stainless on the entire bitcoin-s codebase results in many errors – as was to be expected. We tried to extract only the code relevant to the no-inflation-property and to verify that. However, even the extracted code has more than 1500 lines and liberally uses Scala features outside of the supported fragment. We tried to transform the code into the supported fragment, but quickly realized that a better approach is to first verify a simpler property depending on less code and later come back to the no-inflation property with more experience. So we now turn to the addition-with-zero property.

```scala
val outPoint = TransactionOutPoint(prevTx.txId, UInt32.zero)

val utxoSpendingInfo = BitcoinUTXOSpendingInfo(
  outPoint = outPoint,
  output = utxo,
  signers = List(privKey),
  redeemScriptOpt = None,
  scriptWitnessOpt = None,
  hashType = HashType.sigHashAll
)

val utxos = List(utxoSpendingInfo)

val destinationAmount = Satoshis(Int64(5000))

val destinationSPK = P2PKHScriptPubKey(pubKey = ECPrivateKey.
    freshPrivateKey.publicKey)

val destinations = List(
  TransactionOutput(currencyUnit = destinationAmount, scriptPubKey
      = destinationSPK)
)

val feeRate = SatoshisPerByte(Satoshis.one)

val networkParams = RegTest // some static values for testing

val txBuilderF: Future[BitcoinTxBuilder] = BitcoinTxBuilder(
  destinations = destinations,
  utxos = utxos,
  feeRate = feeRate,
  changeSPK = creditingSPK,    // where to send the change
  network = networkParams
)

val txF: Future[Transaction] = txBuilderF.flatMap(_.sign)

val tx: Transaction = Await.result(txF, 1 second)
```

Fig. 5. Creating a transaction

3 The Addition-with-Zero Property

It is of course a crucial property we are verifying here: if zero satoshis were credited to your account, you would not want your balance to change! It is also the simplest meaningful property to verify that we can think of. However, the code involved in performing the addition of two satoshi amounts in bitcoin-s is nontrivial. The reason for that is a peculiarity of consensus code: agreement with the majority is more important than correctness, whatever correctness might mean. The most widely used bitcoin implementation by far is the reference implementation Bitcoin Core, written in C++. For consensus code, bitcoin-s has little choice but to be in strict agreement with the reference implementation. To achieve that, it implements C-like data types in Scala and then implements functionality using those C-like data types. For example, the Satoshis class, which represents an amount of satoshis, is implemented using the class `Int64` which aims to represent the C-type `int64_t`.

Extracting the Relevant Code. The relevant code for the addition of satoshis is in two files: CurrencyUnits.scala and NumberType.scala. From those files we removed the majority of the code because it is not needed for the verification of our property. For example, we removed all number types except for `Int64` (so `Int32`, `UInt64`, etc.) because they are not used. We also removed the superclasses `Factory` and `NetworkElement` of `CurrencyUnit` and `Number`, respectively, because the inherited members are not used. We further removed all binary operations on `Number` that are not used, like subtraction and multiplication. The extracted code is shown in Figure 6 and Figure 7.

A Bug in the checkResult Function. Note the `checkResult` function on line 12 and the value `andMask` on line 23 of NumberType.scala. The function is intended to catch overflows by performing a bitwise conjunction of its argument with `andMask` and comparing the result with the argument. However, because of the way Java BigIntegers are represented [15] and because bitwise operations implicitly perform a sign extension [9] on the shorter operand, the function does not actually catch overflows.

While this is a potentially serious bug, it turns out that `checkResult` is only ever called inside a constructor call for a number type which contains the intended range check, see lines 32-35. The `checkResult` function thus can, and should, be removed entirely. The bitcoin-s developers have acknowledged the bug and we submitted a pull request to fix it [4].

Transforming the Code. We now turn to the list of Scala features used by the extracted code which are not supported by Stainless and how to transform the code into the supported fragment. All transformations are *equivalent* in the sense that if the addition-with-zero property holds for the transformed code, then it also holds for the code before the transformation.

Inheriting Objects. In both files we have objects extending the BaseNumbers trait, on lines 30 and 23 respectively, which Stainless does not support. We simply turn those objects into case objects. That transformation is equivalent: case objects have various additional properties (for example, being serializable) but none of our code depends on the absence of those.

```scala
package extracted.number

sealed abstract class Number[T <: Number[T]] {
  type A = BigInt
  protected def underlying: A
  def toLong: Long = toBigInt.bigInteger.longValueExact()
  def toBigInt: BigInt = underlying
  def andMask: BigInt
  def apply: A => T
  def +(num: T): T = apply(checkResult(underlying + num.underlying))

  private def checkResult(result: BigInt): A = {
    require((result & andMask) == result,
      "Result␣was␣out␣of␣bounds,␣got:␣" + result)
    result
  }
}

sealed abstract class SignedNumber[T <: Number[T]] extends Number[T]

sealed abstract class Int64 extends SignedNumber[Int64] {
  override def apply: A => Int64 = Int64(_)
  override def andMask = 0xffffffffffffffffL
}

trait BaseNumbers[T] {
  def zero: T
}

object Int64 extends BaseNumbers[Int64] {
  private case class Int64Impl(underlying: BigInt) extends Int64 {
    require(underlying >= -9223372036854775808L,
      "Number␣was␣too␣small␣for␣a␣int64,␣got:␣" + underlying)
    require(underlying <= 9223372036854775807L,
      "Number␣was␣too␣big␣for␣a␣int64,␣got:␣" + underlying)
  }

  lazy val zero = Int64(0)
  def apply(long: Long): Int64 = Int64(BigInt(long))
  def apply(bigInt: BigInt): Int64 = Int64Impl(bigInt)
}
```

Fig. 6. Extracted Code from NumberType.scala

```scala
package extracted.currency

import extracted.number.{BaseNumbers, Int64}

sealed abstract class CurrencyUnit {
  type A
  def satoshis: Satoshis
  def ==(c: CurrencyUnit): Boolean = satoshis == c.satoshis
  def +(c: CurrencyUnit): CurrencyUnit = {
    Satoshis(satoshis.underlying + c.satoshis.underlying)
  }
  protected def underlying: A
}

sealed abstract class Satoshis extends CurrencyUnit {
  override type A = Int64
  override def satoshis: Satoshis = this
  def toBigInt: BigInt = BigInt(toLong)
  def toLong: Long = underlying.toLong
  def ==(satoshis: Satoshis): Boolean = underlying == satoshis.
      underlying
}

object Satoshis extends BaseNumbers[Satoshis] {
  val zero = Satoshis(Int64.zero)
  def apply(int64: Int64): Satoshis = SatoshisImpl(int64)
  private case class SatoshisImpl(underlying: Int64) extends Satoshis
}
```

Fig. 7. Extracted Code from CurrencyUnits.scala

Abstract Type Members. In CurrencyUnits.scala on line 6 there is an abstract type that is not supported. Note that we can not simply replace it with a (supported) type parameter since the CurrencyUnit class uses one of its implementing classes: Satoshis. Since the Satoshis class overrides `A` with `Int64` anyway, we just remove the abstract type declaration and replace `A` by `Int64` everywhere.

Non-Literal BigInt Constructor Argument. In CurrencyUnits.scala on line 18 the BigInt constructor is called with a non-literal argument. As described before, the types in the Stainless library are more restricted than their Scala library counterparts. In particular, the Stainless BigInt constructor is restricted to literal arguments. So we simply replace `toLong` by `underlying.toBigInt`: instead of converting the underlying `Int64` (which in turn has an underlying `BigInt`) to `Long` and then back to `BigInt` we simply directly return the `BigInt`. This is an equivalent transformation: the only thing that might go wrong in the detour via `Long` is that the underlying `BigInt` does not fit into a `Long`. However, the only constructor of `Int64Impl` ensures exactly that and all functions producing `Int64` do so via this constructor.

Self-Reference in Type Parameter Bound. In NumberTypes.scala both on lines 3 and 19 is a class with a type parameter and a type boundary that contains that type parameter itself. Stainless does not currently support such self-referential type boundaries. We opened an issue [3] on the Stainless repository and the developers have targeted version 0.4 to support self-referential type boundaries. Since our code only uses Number with type parameter `T` instantiated to `Int64`, we just remove the type parameter declaration and replace all its occurrences by `Int64`.

Missing Member `bigInteger` in BigInt. In NumberType on line 6 there is a reference to `bigInteger`. The Scala `BigInt` class is essentially a wrapper around `java.math.BigInteger`. `BigInt` has a member `bigInteger` which is the underlying instance of the Java class. The Java class has a method `longValueExact` which returns a `long` only if the `BigInteger` fits into a `long`, otherwise throws exception. Stainless does not support Java classes and in particular its `BigInt` has no member `bigInteger`. However, our code does not call `toLong` anymore, so we just remove it.

Type Members. In NumberType.scala there is a type member on line 4. Our version of Stainless (0.1) does not support type members. We just remove the declaration and replace all occurrences of `A` with `BigInt`, since `A` is never overwritten in an implementing class. Note that in the meantime Stainless has implemented support for type members [13]. Since version 0.2 verification should succeed without this change.

Missing Bitwise-And Method on BigInt. Contrary to Scala `BigInt`, the Stainless `BigInt` class does not support bitwise operations, in particular not the &-method used in NumberType.scala on line 13. However, as described above, the `checkResult` function is both broken and redundant, so we remove it and all calls to it.

Inner Class in Case Object. We have inner classes in NumberType.scala on line 31 and in CurrencyUnits.scala on line 26. Stainless does not support inner classes in a case object. We just move the inner classes out of the case objects. They do not interfere with any other code.

Message Parameter in Require. The calls of the require function on lines 32 and 34 in CurrencyUnits.scala have a second parameter: the error message. Stainless does not support the message parameter. We simply remove it.

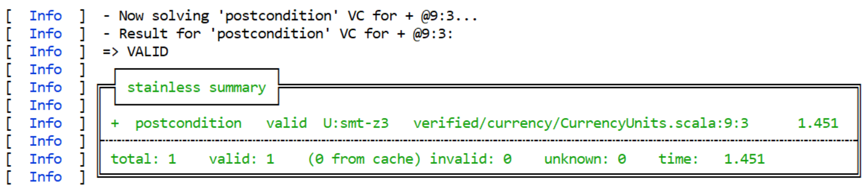
Missing Implicit Long to BigInt Conversion. The Scala `BigInt` class has implict conversions from `Long` which NumberType.scala uses on lines 32 and 34. They are missing in the Stainless `BigInt`. A `BigInt` constructor with a `Long` argument is also missing. We thus replace the `Long` literals by an explicit call to the `BigInt` constructor with a literal string argument, e.g. `BigInt("-9223...5808")`.

The Specification. Now that all our code has been transformed into the supported fragment, we can finally write our specification, shown in Figure 8, and verify it with Stainless, as the output in Figure 9 shows.

The original bitcoin-s code we started from, the extracted code, and the finally verified code are available in our GitHub repository [6].

```
 9   def +(c: CurrencyUnit): CurrencyUnit = {
10     Satoshis(satoshis.underlying + c.satoshis.underlying)
11   } ensuring (res =>
12     (c == Satoshis.zero) ==> (res == this))
```

Fig. 8. Addition function with specification



```
[  Info  ]  - Now solving 'postcondition' VC for + @9:3...
[  Info  ]  - Result for 'postcondition' VC for + @9:3:
[  Info  ]  => VALID
[  Info  ]
[  Info  ]  stainless summary
[  Info  ]
[  Info  ]  +  postcondition   valid  U:smt-z3   verified/currency/CurrencyUnits.scala:9:3      1.451
[  Info  ]
[  Info  ]  total: 1    valid: 1    (0 from cache) invalid: 0    unknown: 0    time:   1.451
[  Info  ]
```

Fig. 9. Stainless output for the transformed code

4 Conclusion and Future Work

We are happy to see some friendly green verifier output. However, apart from the bugs we found, the main conclusion from this work is that we had to non-trivially transform even a very small portion of the code in order to verify it. At the moment, it is unrealistic to routinely formally verify properties as part of the bitcoin-s development process. However, Stainless development has already progressed (e.g. type members are supported in recent versions) and continues to do so (e.g. self-referential type bounds are on the roadmap). Some missing features that we identified are presumably very easy to support, like the message parameter in the require function. Some other features presumably require more substantial work, like bitwise operations on integer types.

On the other hand, bitcoin-s uses features that might not be supported even by future Stainless versions, such as calls to Java code. Here the bitcoin-s code can hopefully be adapted to accommodate formal verification.

From our results we conclude that formal verification of bitcoin libraries in general and bitcoin-s in particular is a worthwhile endeavour. We are looking forward to verifying more substantial parts of the code in future work.

References

1. Blanc, R., Kuncak, V.: Sound reasoning about integral data types with a reusable SMT solver interface. In: Haller and Miller [7], pp. 35–40. https://doi.org/10.1145/2774975.2774980
2. Blanc, R., Kuncak, V., Kneuss, E., Suter, P.: An overview of the leon verification system: verification by translation to recursive functions. In: Proceedings of the 4th Workshop on Scala, SCALA@ECOOP 2013, Montpellier, France, July 2, 2013. pp. 1:1–1:10. ACM (2013). https://doi.org/10.1145/2489837.2489838
3. Boss, R.: Issue 519: Unknown type parameter type T in self referencing generic, https://github.com/epfl-lara/stainless/issues/519, accessed 2019-06-27
4. Boss, R.: Remove redundant function checkresult, https://github.com/bitcoin-s/bitcoin-s/pull/565, accessed 2019-07-03
5. Boss, R.: Transaction can reference two different outputs of the same previous transaction, https://github.com/bitcoin-s/bitcoin-s/pull/435, accessed 2019-06-19
6. Boss, R., Brünnler, K., Doukmak, A.: The bitcoin-s-verification repository, https://github.com/kaibr/bitcoin-s-verification, accessed 2019-07-06
7. Haller, P., Miller, H. (eds.): Proceedings of the 6th ACM SIGPLAN Symposium on Scala, Scala@PLDI 2015, Portland, OR, USA, June 15-17, 2015. ACM (2015)
8. LARA Lab, École Polytechnique Fédérale de Lausanne: Stainless documentation, https://epfl-lara.github.io/stainless/, accessed 2019-06-19
9. Oracle and/or its affiliates: Class BigInteger, https://docs.oracle.com/javase/8/docs/api/java/math/BigInteger.html, accessed 2019-07-03
10. Song, J.: Bitcoin Core Bug CVE-2018–17144: An Analysis, https://hackernoon.com/bitcoin-core-bug-cve-2018-17144-an-analysis-f80d9d373362, accessed 2019-06-20
11. Suredbits & the bitcoin-s developers: The bitcoin-s website, https://bitcoin-s.org, accessed 2019-06-19
12. The bitcoin-s developers: The bitcoin-s repository, https://github.com/bitcoin-s, accessed 2019-06-19
13. The Stainless developers: Type aliases, type members, and dependent function types, https://github.com/epfl-lara/stainless/pull/470, accessed 2019-06-27
14. Voirol, N., Kneuss, E., Kuncak, V.: Counter-example complete verification for higher-order functions. In: Haller and Miller [7], pp. 18–29. https://doi.org/10.1145/2774975.2774978
15. Wikipedia contributors: Two's complement, https://en.wikipedia.org/wiki/Two%27s_complement, accessed 2019-07-03

Part III.

Consensus

On the Formal Verification of the Stellar Consensus Protocol

Giuliano Losa and Mike Dodds

Galois, Inc., Portland, Oregon

Abstract

The Stellar Consensus Protocol (SCP) is a quorum-based BFT consensus protocol. However, instead of using threshold-based quorums, SCP is permissionless and its quorum system emerges from participants' self-declared trust relationships. In this paper, we describe the methodology we deploy to formally verify the safety and liveness of SCP for arbitrary but fixed configurations.

The proof uses a combination of Ivy and Isabelle/HOL. In Ivy, we model SCP in first-order logic, and we verify safety and liveness under eventual synchrony. In Isabelle/HOL, we prove the validity of our first-order encoding with respect to a more direct higher-order model. SCP is currently deployed in the Stellar Network, and we believe this is the first mechanized proof of both safety and liveness, specified in LTL, for a deployed BFT protocol.

1 Introduction

Blockchains rely on Byzantine Fault-Tolerant (abbreviated BFT) consensus protocols to ensure that, despite the presence of malicious participants, the network of participants as a whole eventually reaches consensus on what block to append next to the blockchain. In many blockchains, the security of large amount of digital assets depend on the correctness of the blockchain's BFT consensus protocol, but designing BFT consensus protocols is notoriously difficult and serious flaws can remain undetected for years [1].

While formal verification can prevent many correctness issues in BFT consensus protocols, performing such verification is challenging for several reasons: BFT consensus protocols are designed to support an arbitrary number of participants; their executions and their reachable state-space are unbounded; they operate in asynchronous networks where the interleaving of messages is unpredictable; and finally, verifying termination is as important as verifying that participants never disagree.

In this paper, we summarize our approach to the formal verification of the main safety and liveness properties of the Stellar Consensus Protocol (abbreviated SCP) [7] using the Ivy methodology [10]. Both the safety and liveness proofs apply to a unique model of SCP. This model is parameterized by a fixed but arbitrary set of participants and denotes a set of infinite executions. To our knowledge, this is the first work that mechanically proves both safety and liveness, expressed in LTL, of a deployed BFT protocol under arbitrary configurations.

At a high level, verifying the safety of a protocol with Ivy entails 1) developing a set of axioms to express the protocol's underlying domain model as a first-order theory over uninterpreted sorts; 2) modeling the protocol in Ivy's procedural language; 3) developing an inductive invariant that implies the safety properties, while ensuring that verification conditions fall into the decidable first-order logic fragment EPR [5]. This is facilitated by Ivy's modular decomposition features [17].

For termination, or more generally liveness, Ivy provides a liveness to safety reduction [13] crafted specifically to help produce decidable verification conditions. Given a temporal property in First-Order Linear Temporal Logic (FO-LTL), Ivy automatically synthesises a transition system and an associated safety property such that if the synthesized system is safe, then the temporal property of the original system holds. The user can then verify that the synthesized transition system is safe using the safety verification methodology.

Producing EPR verification conditions ensures that Z3 can automatically and reliably determine their satisfiability. Compared to approaches that use automation but do not require decidability, Ivy's predictable automation greatly simplifies the mental model of the prover that the user must keep in mind when developing a proof. The user can thus stop worrying about the prover and instead focus on the properties of the protocol.

A key challenge in applying the Ivy methodology to SCP is to model SCP's permissionless Federated Byzantine Quorum Systems in first-order logic and in a way that is conductive to decidable reasoning in EPR. In SCP, every participant expresses agreement requirements with other nodes, and SCP relies on the properties of the resulting graph-like structure to solve consensus. At first sight, such a complex family of structures seem hard to axiomatize in first-order logic, let alone EPR.

The rest of the paper focuses on the first-order logic modeling of Federated Byzantine Quorum Systems. This model abstracts over significant aspects of SCP's quorum system. To provide evidence that the abstraction is sound, we verify some of its key properties with respect to a more concrete model in Isabelle/HOL.

With a first-order theory of Federated Byzantine Quorum Systems established, we verify that SCP's balloting protocol [7] satisfies its agreement property and that, under eventual synchrony, it satisfies its termination property. This safety and liveness proof largely follows patterns identified previously during the verification of other consensus protocols in Ivy [12, 13, 3], and it is not described in this paper.

Our paper supplies evidence that BFT consensus protocol can be verified with decidable logics, which enables powerful yet stable automation, using the Ivy methodology. The proof is available online [8], and, for a more complete reference, we plan to publish an extended version of this paper with a detailed account of the safety and liveness proof that are omitted here.

2 Solving Consensus in a Federated Byzantine Quorum System

SCP must solve consensus, guaranteeing agreement and termination, in a permissionless system where nodes can join or leave without any synchronization and without the permission of any gatekeeper. There is thus no common notion of the set of all nodes. Moreover, the system is susceptible to Sybil attacks, in which attackers create a large number of identities to try to overwhelm the system. In such an environment, traditional threshold-based quorum systems, defined in terms of the total number of nodes, are thus of no use.

Other permissionless protocols like Bitcoin or Algorand use Proof-of-Work or Proof-of-Stake to defend against Sybil attacks. Stellar takes a different approach. The Stellar Network is intended as a platform to exchange digitized real-world assets (e.g. land parcels, retail coupons, national currencies, agricultural goods, etc.). Most participants are thus expected to engage with recognized identities and have real-world relationships with some (but not all) other participants in the network. SCP leverages these real-world relationships to defend against Sybil attacks, counting on real-world relationships to be difficult for an attacker to establish.

Concretely, each node in the Stellar Network is required to independently declare a set of *slices*, where each slice is a set of nodes. The intent is that a node n trusts some new information it hears on the network if and only if one of its slices unanimously agrees that the information is correct. Nodes advertise their slices throughout the network, and each nodes forms its own, personal notion of quorum based on its own slices and on the slices of other nodes it knows about, as follows. A quorum of n is defined as a set Q of nodes such that a) n has a slice included in Q and b) each member of Q has a slice included in Q. In other words, thinking of slices as trustworthy sets, a quorum of n is a set that n trusts and that is trusted by all its members. The resulting quorum system is called a Federated Byzantine Quorum System (abbreviated FBQS).

2.1 Intact and Intertwined Sets

With the notion of quorum in place, it seems possible to take a traditional threshhold-based BFT consensus protocols, and only change how quorums are defined in order to obtain a consensus protocol for the Stellar Network. However, FBQS have some unusual properties that complicate the task. First, the notion of quorum is not global to the system; instead, each node has its own view of what a quorum is. Second, the quorums of a node depend on what slices other nodes declare; thus, Byzantine nodes can influence a well-behaved node's notion of what a quorum is. Third, it is possible that a subset of the nodes have quorums that intersect enough to guarantee safety, while some other subsets do not; thus, consensus may be solvable for only a strict subset of the system; there may even by two or more disjoint subsets of the system that form consensus islands that nevertheless diverge from each other.

What properties must a set of nodes satisfy in order for consensus to be solvable among its members? We do not know a precise answer to this question [9]. However, we can prove that SCP solves eventually synchronous consensus among sets of nodes called *intact sets*. A set I of well-behaved (non-Byzantine) nodes is intact when, regardless of what slices Byzantine nodes advertise: a) I enjoys quorum availability, i.e. the set I is a quorum for all its members, and b) I enjoys quorum intersection, i.e. if n_1 and n_2 are members of I, if Q_1 is a quorum of n_1, and if and Q_2 is a quorum of n_2, then the intersection of Q_1 and Q_2 contains a member of I. An important property of intact sets is that the union of two intact sects that intersect is also an intact set; thus maximal intact sets are disjoint and form consensus islands within the network.

Furthermore, SCP also guarantees that there will not be any disagreement among an *intertwined* set. A set S of nodes is intertwined when, if n_1 and n_2 are members of S, if Q_1 is a quorum of n_1, and if Q_2 is a quorum of n_2, then the intersection of Q_1 and Q_2 contains an intertwined member.

2.2 Termination and the Cascade Theorem

Thanks to the quorum intersection property, it is easy to guarantee agreement to an intertwined set. However, termination is more difficult to achieve. Traditional BFT consensus protocols often rely on eventual synchrony [4] to ensure termination. The idea is that, once the system becomes synchronous, the protocol can rely on all nodes having the same view of the system.

For example, suppose that, in a threshold quorum system, a quorum Q unanimously agrees on statement X. If the network is synchronous, then all nodes shortly notice that Q unanimously agrees on X. In this sense, they all form the same view of the fact "there is a quorum that is unanimous about X". Instead, if the quorum system is not a threshold quorum system but an FBQS, then no such common view arises because Q may be a quorum only of some nodes but not others.

SCP circumvents this problem using an epidemic propagation phenomenon that guarantees that, once the system is synchronous, if an intact node witnesses a unanimous quorum, then the knowledge that there is such a quorum soon propagates to the entire intact set, and Byzantine nodes cannot prevent propagation.

The epidemic propagation phenomenon is enable by the Cascade Theorem. This theorem relies on the notion of slice-blocking set. A set B is a slice-blocking set for a node n when every slice of n intersects B. The cascading theorem states that if n is intact, Q is a quorum of n, and U is a superset of Q, then either all intact nodes belong to U, or U slice-blocks some intact node $m \notin U$.

Let us now get back to the example in which a quorum Q of an intact node unanimously agrees on statement X. We would like that all intact nodes to learn the fact "there exists a quorum of an intact node that unanimously agrees on X". By the Cascade Theorem, either all intact nodes already know the fact, or there must be an intact node n that does not know it but that is slice-blocked by a set of intact nodes that know it. Thus, if we add the rule that n must accept a fact if slice-blocked by a set that already accepted the fact, then n newly accepts the fact. This process then repeats until the knowledge of the existence of Q propagates to the entire intact set.

Finally, we must also be sure that malicious nodes cannot use epidemic propagation to propagate forged facts. This is guaranteed because if n is intact and S slice-blocks n, then S contains an intact node.

3 Modeling Federated Byzantine Quorum Systems in EPR

In this section, we describe the first-order theory of Federated Byzantine Quorum System. This is the model we use in our proofs of safety and liveness (the proofs themselves are not described in detail in this paper).

3.1 Enabling Decidable Reasoning

We craft the FBQS model to meet two constraints: on the one hand, the model must enable decidable automated reasoning in EPR; on the other hand, the model must accurately capture the properties that FBQSs have in practice. Our solution is to abstract over some important aspects of FBQSs to make decidable reasoning possible, while formally verifying that the model is sound with respect to a more faithful model developed in Isabelle/HOL. By doing so, we trade off a relatively small manual proof effort in Isabelle/HOL in exchange for decidable automated reasoning in Ivy.

To enable decidable reasoning with Ivy, we model FBQSs as a first-order theory consisting of: a) a set of uninterpreted sorts, b) constants, functions, and relations over those sorts, and c) first-order axioms that constrain the models of the theory to structures that have properties sufficient for the balloting protocol to be correct. Moreover, we must use quantifier alternations and functions carefully, as those will impact our ability to keep protocol verification conditions in EPR.

A verification condition is in EPR when its sorts are stratified: for every pair of sorts a and b, say that b depends on a if either a) an existential quantifier on sort b is in the scope of a universal quantifier on sort a, or b) there is a function symbol of type $a_1, \cdots, a_n \to b$ with $a = a_j$ for some $j \in 1 \cdots n$; sorts are stratified if the dependencies between sort do not form any loops or cycles. For example, in the formula $\forall x. \exists y. P(y)$ where x is of sort a and y is of sort b and P is a predicate symbol, sort b depends on sort a but the formula is stratified. However, if both x and y have the same sort, then there is a sort dependency loop and the formula is not stratified.

Protocol verification conditions are formulas of the form $A \wedge I \wedge T \wedge \neg I'$, where A is the conjunction of the FBQS theory axioms, I is a protocol invariant, T is the protocol's transition relation, and I' is the post-state version of I. Thus unstratified verification conditions can arise because of the interaction between axioms, invariants, their negation, and the protocol's transition relation. It is thus wise to minimize the use of function symbols and quantifier alternation when developing the EPR FBQS theory.

In our experience, stratification is nevertheless likely to become an issue during protocol verification. However, Ivy has modular decomposition features specifically designed to help keep verification conditions decidable. The process of structuring proof modularly to ensure decidability is explained in details by Taube et al. [17]. In the case of liveness proof, prophecy variables also help keep verification conditions decidable [14].

3.2 The Unique Challenges Posed by FBQSs

Developing an EPR theory of FBQSs is challenging because the notions we presented in Section 2, such as intact set, slice-blocking sets, or the cascading theorem, are naturally second-order concepts. I.e. they are naturally expressed by quantifying over sets. While we cannot precisely capture quantification over node sets in first-order logic, we can approximate it by using a first-order uninterpreted sort `nset`, a membership relation `member(N:node, S:nset)`, and appropriate axioms.

The full first-order theory of FBQSs appears in Figure 1. We model an arbitrary but fixed configuration, i.e. an arbitrary set of nodes with arbitrary slices and we consider a fixed intact set I among those nodes as well as a superset S of I such that S is intertwined (note that an intact set is inherently intertwined, so there is no inconsistency here). Instead of modeling slices explicitly, we only model the notions of intact node, intertwined node, quorum, and slice-blocking set.

Formally, in Figure 1, we introduce an uninterpreted sort `node`, denoting the set of all nodes, and an uninterpreted sort `nset`, denoting the powerset of the set of nodes (lines 1 and 2). Well-behaved, intertwined, and intact nodes are identified by corresponding unary relations (lines 3 to 5), and quorums of a node are identified by the binary relation `quorum_of` (line 7). Finally, the binary relation `member` (line 6) denotes set membership, and the binary relation `slice_blocking` identifies the slice-blocking sets of a node (line 8).

Given those sorts and relations, we obtain the first-order theory of FBQSs using the following axioms. First, line 9, we assert that intact nodes are intertwined, and that intertwined nodes are well-behaved. In line 10 and 11, we assert that quorums of well-behaved nodes are not empty. Then, line 12 to 15, we define two predicate to identify quorums of intertwined nodes and quorums of intact nodes. Then, line 16 and 17, we assert the quorum intersection property of intact nodes. Similarly, line 18 and 19, we assert the quorum intersection property of intertwined nodes. Line 20 and 21, we assert that if `N` is intact and `S` slice-blocks `N`, then `S` contains an intact node. Finally, line 22, we assert that the set of intact nodes is a quorum.

The conjunction of all the axioms is an EPR formula because the associated quantifier-alternation graph has a single dependency: sort `node` depends on sort `nset`. For example, lines 16 and 17 in Figure 1, the quorum intersection axiom for intertwined sets creates an dependency from sort `node` to sort `nset`. As explained in Section 3.1, this dependency may create a quantifier-alternation cycle when the axioms are conjoined with other formulas in a verification condition, and it is the user's responsibility to make use of Ivy's modularity features to avoid such a cycle when verifying a protocol; this process is explained in [17]. When proving liveness, the user can additionally introduce prophecy variables that help keep verification conditions decidable [14].

The reader may notice that the Cascade Theorem is missing from the axioms, and instead is expressed as an axiom schema in Figure 2. The reason is that we could not satisfactorily express it in first-order logic. The theorem states that if `p` is a predicate on nodes (i.e. a set of nodes) and `Q` is a quorum of an intact node whose intact members unanimously satisfy `p`, then either a) all intact nodes satisfy `p` or b) there exists an intact node `N` that does not satisfy `p` but that is slice-blocked by a set `S` whose members are exclusively intact and unanimously satisfy `p`. While other axioms quantify over a restricted family of sets, such as quorums or slice-blocking sets, the Cascade Theorem quantifies over all predicates `p`. It is thus inherently second-order. Ivy allows to express it as an axiom schema, but Ivy's proof automation cannot reason about such a second-order formula. Instead, Ivy allows to manually instantiate it, substituting `p` for a concrete predicate, to prove particular invariants. We use this technique in the termination proof of SCP. Note that, also when instantiating the Cascade Theorem, we must be careful not to introduce quantifier-alternation cycles.

Together, the axioms appearing in Figure 1 and the axiom schema of Figure 2 form the first-order theory of Federated Byzantine Quorum Systems.

3.3 Validating the Model

Asserting axioms instead of proving them as properties from basic definitions can be dangerous: even benign-looking axioms can turn out to be contradictory, e.g. because of a typo, thereby making any proof

```
type node # the type of nodes; this is an uninterpreted, arbitrary non-empty type
type nset # the type of node sets
relation well_behaved(N:node)
relation intertwined(N:node)
relation intact(N:node)
relation member(N:node, S:nset) # this is the set membership relation
relation quorum_of(Q:nset)
relation slice_blocking(S:nset, N:node)
axiom ∀ N. (intact(N) → intertwined(N)) & (intertwined(N) → well_behaved(N))
axiom (exists N . well_behaved(N) & quorum_of(N,Q))
    → exists N . well_behaved(N) & member(N,Q)
definition quorum_of_intertwined(Q) =
  (∃ N. intertwined(N) ∧ quorum_of(N,Q))
definition quorum_of_intact(Q) =
 (∃ N. intact(N) ∧ quorum_of(N,Q))
axiom ∀ Q1,Q2. quorum_of_intertwined(Q1) ∧ quorum_of_intertwined(Q2)
  → ∃ N. intertwined(N) ∧ member(N,Q1) ∧ member(N,Q2)
axiom ∀ Q1,Q2. quorum_of_intact(Q1) ∧ quorum_of_intact(Q2)
  → ∃ N. intact(N) ∧ member(N,Q1) ∧ member(N,Q2)
axiom ∀ S. (∃ N. intact(N) ∧ slice_blocking(S,N))
  → ∃ N2. member(N2,S) ∧ intact(N2)
axiom ∃ Q. ∀ N. member(N,Q) ↔ intact(N) & quorum_of(N,Q)
```

Figure 1: A model of Federated Byzantine Quorum Systems in the EPR fragment of first-order logic

```
axiom [cascade_thm] {
  function p(N:node):bool
  property (exists Q . quorum_of_intact(Q) & (forall N . intact(N) & member(N,Q) → p(N)))
    → ((forall N . intact(N) → p(N))
        | (exists N,S . intact(N) & ¬p(N) & slice_blocking(S,N)
           & (forall N2 . member(N2,S) → (intact(N2) & p(N2)))))
}
```

Figure 2: The second-order Cascade Theorem as an axiom schema in Ivy.

relying on them vacuous. To avoid this situation, we ask Ivy to find a model of the axioms of Figure 1 conjoined with the instantiations of the cascade_thm axiom schema that we use in the proof. Ivy confirms the existence of a model, which rules out any contradiction.

Another risk is that, although the axioms are not contradictory, they do not accurately model FBQSs. For instance, the first-order model abstracts over slices and instead considers that a node's quorums are fixed. This is limiting because, in reality, nodes are expected to change their slices in response to observed failures or changes in how much they trust other nodes. It is nevertheless interesting to prove that, under the assumption that well-behaved nodes do not change their slices, SCP is safe and live despite the arbitrary behavior of malicious nodes. However, at first glance, the first-order FBQS model does not seem to accurately capture that situation either. The issue is that, as we have noted in Section 2, FBQSs have the peculiar property that malicious nodes can, by advertising arbitrary slices, dynamically influence a well-behaved node's notion of quorum. But in our model, the quorums of a well-behaved

```
theorem cascade:
  fixes P
  assumes "∃ Q . ∃ n . intact n ∧ quorum_of n Q ∧ (∀ n ∈ Q . intact n ⟶ P n)"
  obtains "∀ n . intact n ⟶ P n" | "∃ n S . intact n ∧ ¬P n
    ∧ (∀ S1 ∈ slices n . S ∩ S1 ≠ {}) ∧ (∀ n ∈ S. intact n ∧ P n)"
```

Figure 3: The Cascade Theorem in Isabelle/HOL.

node are fixed. Surprisingly, as we show in Isabelle/HOL, the first-order model is nevertheless sound with respect to a model in which quorums can be shaped dynamically by malicious nodes advertising arbitrary slices.

The Isabelle/HOL model formalizes FBQSs from the notion of slice. It assumes that well-behaved nodes have fixed slices, but it accounts for the situation in which malicious nodes dynamically shape the quorums of well-behaved nodes. To do so, we define a quorum Q of a node n as a set of nodes such that a) n has a slice included in Q and b) every *well-behaved* member of Q has a slice included in Q. Note how this definition of quorum subtly differs from the one of Section 2. By placing requirements only on well-behaved nodes, we account for any possible slices that could be advertised by malicious nodes. We then prove in Isabelle/HOL that all the axioms of the first-order model (Figure 1) and the Cascade Theorem (Figure 2) hold. This Isabelle/HOL theory is purely definitional (i.e. it does not use axioms).

There is no mechanically-checked connection between Isabelle/HOL and Ivy, and thus the best we can do is to carefully check, by hand, that the Ivy axioms correspond to the properties proved in Isabelle/HOL. Fortunately, the syntax and semantics of first-order formulas in Isabelle/HOL is very close to that of Ivy. This can be seen by comparing the Ivy axiom schema of Figure 2 with its Isabelle/HOL counterpart appearing in Figure 3.

4 Related Work

Lokhava et al [7] discuss the Stellar Network in the broader context of global payments; they also describe at a high level the formal verification effort that is the subject of the present paper. The purpose of the present paper is to dig into the technical details necessary to apply this technique to future proofs of BFT protocols. Losa et al. [9] show that FBQSs are an instance of the more general Personal Byzantine Quorum System model, and we reuse some of the Isabelle/HOL theories developed for this work.

Other works verify safety properties of BFT consensus protocols using Dafny, Coq, or Isabelle/HOL. For example, Alturki et al. verify safety properties of Algorand in Coq [2]. Palmskog et al.[15] and Nakamura et al.[11] verify properties of Ethereum's Casper CBC in Coq and Isabelle/HOL, respectively. Rahli verifies safety properties of PBFT in the Velisarios framework [16], which is based on Coq. IronFleet [6] verifies safety and liveness of a crash-tolerant implementation of Multi-Paxos using Dafny.

Isabelle/HOL, Dafny, and Coq are not restricted by decidable logics, but they lack the specific features that allow Ivy users to restrict verification conditions to a decidable fragment and in turn benefit from reliable proof automation. A series of papers describe the different aspects of decidable reasoning about protocols in Ivy. [12] focuses on modeling and safety verification of consensus protocols at a high level of abstraction. [3] presents a tool to synthesize first-order axioms modeling threshold-based quorum systems.[17] present Ivy's modularity features, which enable decidable safety verification of more complex protocols and their implementations. Finally, Ivy's liveness-to-safety reduction [13] allows decidable reasoning about liveness properties expressed in LTL. Ivy's support for prophecy variables [14] offers an additional tool that helps preserve decidability. In an extended version of this paper, we plan to present the Ivy proofs of safety and liveness of SCP and compare with the works cited above.

References

[1] Ittai Abraham et al. “Revisiting Fast Practical Byzantine Fault Tolerance”. In: *arXiv preprint arXiv:1712.01367* (2017).

[2] Musab A. Alturki et al. “Towards a Verified Model of the Algorand Consensus Protocol in Coq”. In: *arXiv preprint arXiv:1907.05523* (2019).

[3] Idan Berkovits et al. “Verification of Threshold-Based Distributed Algorithms by Decomposition to Decidable Logics”. In: *Computer Aided Verification*. Ed. by Isil Dillig and Serdar Tasiran. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 245–266. ISBN: 978-3-030-25543-5.

[4] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. “Consensus in the Presence of Partial Synchrony”. In: *Journal of the ACM (JACM)* 35.2 (1988), pp. 288–323. URL: http://dl.acm.org/citation.cfm?id=42283 (visited on 01/06/2017).

[5] Yeting Ge and Leonardo De Moura. “Complete Instantiation for Quantified Formulas in Satisfiabiliby modulo Theories”. In: *Computer Aided Verification*. Springer, 2009, pp. 306–320. URL: http://link.springer.com/content/pdf/10.1007/978-3-642-02658-4.pdf#page=321.

[6] Chris Hawblitzel et al. “IronFleet: Proving Practical Distributed Systems Correct”. In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP ’15. New York, NY, USA: ACM, 2015, pp. 1–17. ISBN: 978-1-4503-3834-9. URL: http://doi.acm.org/10.1145/2815400.2815428 (visited on 01/03/2017).

[7] Marta Lokhava et al. “Fast and Secure Global Payments with Stellar”. In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. SOSP ’19. Huntsville, Ontario, Canada: Association for Computing Machinery, Oct. 27, 2019, pp. 80–96. ISBN: 978-1-4503-6873-5. URL: https://doi.org/10.1145/3341301.3359636 (visited on 05/11/2020).

[8] Giuliano Losa. 2019. URL: https://github.com/stellar/scp-proofs.

[9] Giuliano Losa, Eli Gafni, and David Mazières. “Stellar Consensus by Instantiation”. In: *33rd International Symposium on Distributed Computing (DISC 2019)*. Ed. by Jukka Suomela. Vol. 146. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 27:1–27:15. ISBN: 978-3-95977-126-9. URL: http://drops.dagstuhl.de/opus/volltexte/2019/11334 (visited on 03/23/2020).

[10] Kenneth L. McMillan and Oded Padon. “Deductive Verification in Decidable Fragments with Ivy”. In: *International Static Analysis Symposium*. Springer, 2018, pp. 43–55.

[11] Ryuya Nakamura, Takayuki Jimba, and Dominik Harz. *Refinement and Verification of CBC Casper*. 415. 2019. URL: https://eprint.iacr.org/2019/415 (visited on 05/21/2020).

[12] Oded Padon. “Paxos Made EPR: Decidable Reasoning about Distributed Protocols.” In: *PACMPL* 1 (OOPSLA 2017), 108:1–108:31.

[13] Oded Padon et al. “Reducing Liveness to Safety in First-Order Logic”. In: 45th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2018). Los Angeles, 2018.

[14] Oded Padon et al. “Temporal Prophecy for Proving Temporal Properties of Infinite-State Systems”. In: *2018 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 2018, pp. 1–11.

[15] Karl Palmskog et al. *Verification of Casper in the Coq Proof Assistant*. 2018. URL: https://github.com/runtimeverification/casper-proofs/blob/master/report/report.pdf.

[16] Vincent Rahli et al. “Velisarios: Byzantine Fault-Tolerant Protocols Powered by Coq”. In: *Programming Languages and Systems*. Ed. by Amal Ahmed. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 619–650. ISBN: 978-3-319-89884-1.

[17] Marcelo Taube et al. “Modularity for Decidability of Deductive Verification with Applications to Distributed Systems”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). 2018.

Formal Specification and Model Checking of the Tendermint Blockchain Synchronization Protocol (Short Paper) [*]

Sean Braithwaite, Ethan Buchman, Igor Konnov, Zarko Milosevic, Ilina Stoilkovska, Josef Widder, and Anca Zamfir

Informal Systems, Canada, Switzerland, Austria
{sean,ethan,igor,zarko,ilina,josef,anca}@informal.systems

Abstract

Blockchain synchronization is one of the core protocols of Tendermint blockchains. In this short paper, we discuss our recent efforts in formal specification of the protocol and its implementation, as well as some initial model checking results. We demonstrate that the protocol quality and understanding can be improved by writing specifications and model checking them.

1 Introduction

Tendermint is a state-of-the art Byzantine-fault-tolerant state machine replication (BFT SMR) engine equipped with a flexible interface supporting arbitrary state machines written in any programming language [4]. Tendermint is particularly popular for proof-of-stake blockchains, and constitutes a core component of the Cosmos Project [5]. At the heart of the Cosmos Project is the InterBlockchain Communication (IBC) protocol for reliable communication between independent BFT SMs; what TCP is for computers, IBC aims to be for blockchains.

Multiple Tendermint-based blockchains currently run in production on the public Internet and have for over a year, with new ones launching regularly, carrying billions of dollars of cumulative value in the market capitalizations of their respective cryptocurrencies. One of the primary deployments, the so-called Cosmos Hub blockchain, is operated by a diverse set of 125 consensus forming nodes connected to one another over an open-membership gossip network consisting of hundreds of other nodes.

Tendermint was the first to apply traditional BFT consensus protocols to blockchain systems [11]. The core Tendermint BFT consensus protocol constitutes a modern implementation of the consensus algorithm for Byzantine faults with Authentication from [8] built on top of an efficient gossiping layer. The latest description of the consensus protocol can be found in the technical report [6]. Tendermint consensus has been a source of inspiration for a wide variety of blockchain systems that have followed [15, 7], though few, if any, have achieved its level of maturity in production.

The reference implementation of the Tendermint software is written in Go [3]. Under the hood, it consists of several fault-tolerant distributed protocols that interact to ensure efficient operation:

Consensus. Core BFT consensus protocol including the gossiping of proposals, blocks, and votes.

[*]Supported by Interchain Foundation (Switzerland)

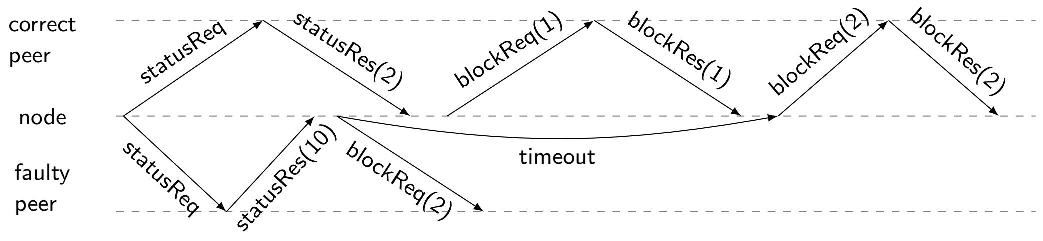


Figure 1: A Fastsync execution for a fully unsynchronized node of height 1

Evidence. To incentivize consensus participants to follow the consensus protocol (and not behave faulty), in the proof-of-stake systems, misbehavior is punished by destroying stake. This protocol gossips evidence of malicious behavior in the form of conflicting signatures.

Mempool. A protocol to gossip transactions, ensuring transactions eventually end up in a block are distributed to all participants.

Peer Exchange. Gossiping is based on communication only with a subset of the peers. Managing list of available peers and selecting the peers based on performance metrics is done by this protocol.

Blockchain synchronization (Fastsync). If a peer gets disconnected by the network for some time, it might miss the most recent blocks in the blockchain. A node that recovers from such a disconnection uses the blockchain synchronization protocol called Fastsync to learn blocks without going through consensus.

We are conducting a project to formally specify and model check these protocols. The first protocol we considered was the blockchain synchronization protocol called *Fastsync*. Specifications can be found in English [12] and TLA$^+$ [13].

Fastsync. A full node that connects to a Tendermint blockchain needs to synchronize its state to the latest global state of the network. One way to achieve this is to update its local copy of the blockchain and replay all transactions, using Fastsync: Initially, the node has a local copy of a blockchain prefix and the corresponding application state that may be out of date. The node queries its peers for the blocks that were decided on by the Tendermint blockchain since the time the full node was disconnected from the system. After receiving these blocks, the protocol executes the transactions that are stored in the blocks, in order to synchronize to the current height of the blockchain and the corresponding application state.

Figure 1 shows a typical execution of the Blockchain Synchronization protocol. In this execution, a new node connects to two full nodes: a correct peer and a faulty peer. The node requests the blockchain heights of the peers by issuing `statusReq`. Once a peer replies with its height, e.g., with `statusRes(10)`, the node can request for a block i by sending the message `blockReq(i)`. In our example, the correct peer receives the request `blockReq(1)` for block 1 and replies with the message `blockRes(1)` that contains the block. In a Tendermint blockchain, the commit for block (signed votes messages) `h` is contained in block `h+1`, and thus a node performing Fastsync must receive two sequential blocks before it can verify fully the first one. If verification succeeds, the first block is accepted; if it fails, both blocks are rejected, since it is not known which block was faulty. When the node rejects a block, it suspects the sending peer of being faulty and evicts this peer from the set of peers. The same happens,

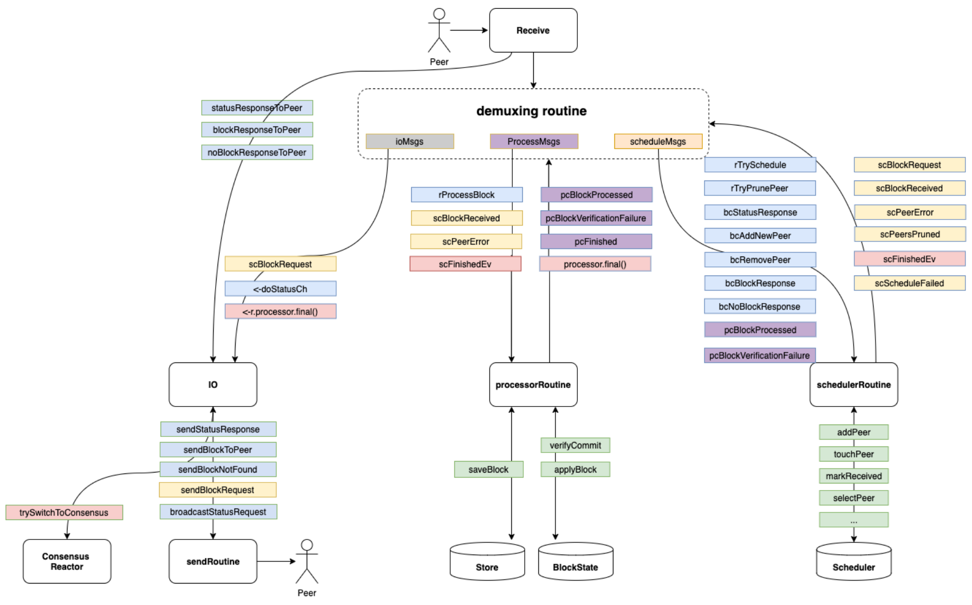


Figure 2: Concurrent threads of execution in the Fastsync implementation [1].

when a peer does not reply within a predefined time interval. In our example, the faulty peer is evicted, and the node finishes synchronization with the correct peer.

The above example may produce an impression that it is easy to specify and verify correctness of Fastsync. (The authors of this paper thought so.) By writing several protocol specifications in English and TLA$^+$ and by running model checkers, we have found that specifications in particular in the presence of faulty peers are intricate.

2 Architecture

The most recent implementation of the Fastsync protocol, called V2, is the result of significant refactoring to improve testability and determinism, as described in the Architectural Decision Record [1]. In the original design, a go-routine (thread of execution) was spawned for each block requested, and was responsible for both protocol logic and network IO. In the V2 design, protocol logic is decoupled from IO by using three concurrent threads of execution: a scheduler, a processor, and a demuxer, as per Figure 2. The scheduler contains the business logic for tracking peers and determining which block to request from whom, while the processor handles the computationally expensive block processing, including verification of consensus signatures and execution of all transactions. The demuxer is responsible for all IO, including translating between internal events and network IO messages, and routing events between components. Both the scheduler and processor are structured as finite state machines with input and output events. Inputs are received on an unbounded priority queue, with higher priority for error events. Output events are emitted on a blocking, bounded channel. Network IO is handled by the Tendermint p2p subsystem, where messages are sent in a non-blocking manner.

3 Specifications in English and TLA$^+$

Structured Specification in English. We start our formalization by a structured English specification [12], that consists of four parts:

1. *Blockchain.* Formalization of relevant properties of the blockchain and its blocks.
2. *Sequential problem statement.* Parts of the sequential safety specification read as follows:

   "Let bh be the height of the blockchain at the time Fastsync starts. When the protocol terminates, it outputs a list of all blocks from its initial height to some height $terminationHeight \geq bh - 1$". (Fastsync cannot synchronize to the maximum height bh as in Tendermint, verification of block at height h requires the commit from the block at height $h + 1$.)

   This specification is sequential, as it ignores that the blockchain is implemented in a distributed system, in which validators may be faulty. Even if they are correct, they locally have prefixes of different lengths, so that bh is not clearly defined a priori.
3. *Distributed aspects.* Here we introduce the computational model and the refinement of the problem statement. For instance, the above translates to:

   "Let $maxh$ be the maximum height of a correct peer to which the node is connected at the time Fastsync starts. If the protocol terminates successfully, it is at some height $terminationHeight \geq maxh - 1$."
4. *Distributed protocol.* Specification of the protocol, where we describe inputs, outputs, variables, and functions used by the protocol. We specify functions mainly by preconditions, postconditions, and error conditions. Further, we provide invariants over the protocol variables. These inform both the implementation and the verification effort.

Specifications in TLA$^+$. The structure of the English specification already highlighted interesting properties of the protocols and pointed to some issues. As it is written in natural language, the English specification is ambiguous. To eliminate the ambiguities, we have written three TLA$^+$ specifications, which focus on different aspects of the protocol and its architecture:

- *High-level specification (HLS).* This specification contains the minimal set of interactions in the synchronization protocol. Its primary purpose is to highlight safety and termination. HLS was mainly written by the researchers.
- *Low-level specification (LLS).* While HLS captures the distributed protocol, there was a significant gap between HLS and the implementation. For instance, the implementation uses additional messages and contains detailed error codes, which are missing in HLS. The low-level specification is much closer to the implementation, and it was mainly written by distributed system engineers.
- *Concurrency specification (CRS).* As discussed above, the V2 implementation utilizes several threads that communicate via queues. To formally capture this structure, we wrote a specification that models threads and message queues.

We discuss the modeling assumptions of HLS [13]. (1) The node starts with a finite set of peers, which can shrink, when the node suspects peers of being faulty. This set is partitioned in two subsets: correct and faulty. (2) The blockchain can grow up to a fixed height. By fixing the parameters of (1) and (2), we run finite-state model checking with TLC [10] and APALACHE [9, 2]. We model the distributed system as two components: the node and its peers. These two components communicate via two variables: `outMsg` that keeps an output message from the node to a peer, and `inMsg` that keeps an input message from a peer to the node; both variables may be set to None, indicating that there is no message. The components alternate their steps: The odd turns belong to the node, whereas the even turns belong to the peers.

This approach is simple but powerful. On one hand, it dramatically decreases the state space, as there are no queues, and alternation produces fewer states than disjunction. On the other hand, it does not decrease precision, as the peers consume and produce message at a high degree of non-determinism. Moreover, this approach allows us to easily formulate fairness in the system as weak fairness over the variable turn, which encodes the scheduled component.

Finally, V2 relies on several timeouts to guarantee termination. In TLA$^+$ specifications, we simply model timeouts with non-determinism and weak fairness.

4 Model Checking with TLC and Apalache

While developing TLA$^+$ specifications, we were using TLA$^+$ Toolbox and the TLC model checker [10]. We also checked the safety properties with the new symbolic model checker APALACHE [9, 2]. So far, we have checked the specifications for tiny parameters such as 1–3 peers and small Blockchain heights. Table 1 summarizes the results and running times of TLC and APALACHE. A central property is the protocol's Termination:

$$\mathrm{WF}_{turn}(\mathit{FlipTurn}) \Rightarrow \diamond(\mathit{state} = \text{"}\mathit{finished}\text{"}) \qquad (\mathsf{Termination})$$

TLC does not find a bug, and it is not surprising: A global timeout guarantees that the protocol terminates, no matter what happens. (Apalache only supports safety properties.) The more interesting property is "synchronization", whose intuitive meaning is that when Fastsync terminates, it has reached the height of the blockchain. Let's formalize this as Sync1: To see that our modeling is precise, let's start with a property we know to be slightly wrong, namely, when the protocol finishes, it reaches the maximum height among the heights of the correct peers, i.e.,

$$\mathit{state} = \text{"}\mathit{finished}\text{"} \Rightarrow \mathit{blockPool.height} \geq \mathit{MaxCorrectPeerHeight}(\mathit{blockPool}) \qquad (\mathsf{Sync1})$$

The model checkers report counterexamples. One reason is that to verify a block h, one needs the commit signatures from block $h+1$. We also observe, that we do not require that node that runs Fastsync needs to be connected to correct peers. Hence, we fix it in Sync2 by stating that height $\mathit{MaxCorrectPeerHeight}(\mathit{blockPool})-1$ should be reached when the node is connected to correct peers. This property also fails. This time we observe that a global timeout — that guarantees Termination — may terminate Fastsync before it has reached the maximal height. We add a precondition for "no timeout", and call the property Sync3. Neither TLC, nor APALACHE produce a counterexample.

The following two properties might appear to be intuitively correct, but the model checkers produce counterexamples. SyncFromCorrect states that the accepted blocks originate only from the correct processes. This property fails, as it does not consider that faulty peers may behave correct in an execution prefix (before they show faulty behavior). Thus, the initial intuition fails. CorrectNeverSuspected states that the correct peers are never removed from the peer set. This would be a desirable property, but the latest implementation V2 does not guarantee it.

5 Conclusions

We approach this work with a process-oriented goal in mind: By *Verification-Driven Development* [14] we understand a design process for distributed systems that makes it easier to test

Table 1: Model checking results for TLC and APALACHE against the high-level specification for the following parameters: 1 correct peer, 1 faulty peer, 4 blocks (Apple MacBook Pro 2019). The symbols in "result" are: found a bug ✗, and no bug up to given length [✓].

Property	TLC (6 CPUs, 13 GB)				APALACHE (1 CPU)		
	result	time	diameter	#states	result	time	length
Sync1	✗	4m36s	8	10M	✗	14s	8
Sync2	✗	5m06s	8	11M	✗	13s	8
Sync3	[✓]	$\geq$ 8h	14	$\geq$ 349M	[✓]	40m	21
Termination	[✓]	$\geq$ 1h	9	$\geq$ 2.7M	*not supported*		
SyncFromCorrect	[✓]	$\geq$ 1h	9	$\geq$ 80M	✗	2m43s	12
CorrectNeverSuspected	✗	7s	6	300K	✗	9s	6

and verify the software. The re-design of the Fastsync protocol that resulted in a decomposition into state machines should be understood under this aspect. The design documents, namely the English and the TLA$^+$ specifications, are artifacts of this design process, and are means of communication between researchers, software engineers, and verification engineers. The structured English specification strikes a balance between mathematical rigor and readability. It serves as the base (i) for formal verification efforts in TLA$^+$ that will give precise semantics, and (ii) for implementations. The annotations with invariants, preconditions, and postconditions are very helpful for the software engineers to guide the implementation.

The formalization also led to a better understanding of the liveness properties that we expect and want from a blockchain synchronization protocols, and a discrepancy to the current implementations (Fastsync V0, V1, and V2). We have found several liveness issues that come from unexpected behavior of faulty peers. For instance, rather than reporting bad blocks, faulty peers may be very slow in reporting good blocks. If they report them slower than the blockchain grows, but fast enough to not lead to a timeout at the node, V2 may never terminate. This highlights that a vital requirement had not been captured before, namely, a relationship between timeout duration, block generation rate, and message end-to-end delays. As this is closely related to real-time, we are not able to capture and reproduce this with TLA$^+$. However, TLA$^+$ counterexamples and the English specifications helped us in isolating this scenario.

For safety verification, we can replace a timeout by a non-deterministic event that may occur at any time. For liveness we have to treat the relation of timeouts to message delays and processing times precisely. The extensive use of timeouts in the current implementation poses a challenge to liveness verification. Some of our current research challenges are therefore timeouts, and we are interested in answering the following questions: How to limit timeouts in the implementation? What is the most effective way to use timeouts in the implementation in order to stay precise in the verification? How can we capture the relation of the (local) timeouts to (global) message delays in model checking? We keep these challenges for future work.

Model checkers and the produced counterexamples were quite helpful in understanding and refining the protocol properties. After refining the protocol, which results in larger state space, we found that TLC could not reach error states within the reasonable time frame of one hour. However, APALACHE was still finding errors within 10 minutes, which was still interactive enough for us. As future work, we also plan to find an inductive invariant and prove its correctness with APALACHE (for fixed but larger parameters).

References

[1] ADR 043: Blockhchain reactor riri-org, 2020. https://github.com/tendermint/tendermint/blob/master/docs/architecture/adr-043-blockchain-riri-org.md.

[2] APALACHE: a symbolic model checker for TLA$^+$. https://github.com/konnov/apalache/, 2020. Last accessed: May 20, 2020.

[3] Tendermint core, reference implementation in Go, 2020. https://github.com/tendermint/tendermint.

[4] Ethan Buchman. Tendermint: Byzantine fault tolerance in the age of Blockchains. Master's thesis, University of Guelph, 2016. http://hdl.handle.net/10214/9769.

[5] Ethan Buchman and Jae Kwon. Cosmos whitepaper: a network of distributed ledgers, 2016. https://cosmos.network/resources/whitepaper.

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

[7] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437*, 2017.

[8] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J.ACM*, 35(2):288–323, 1988.

[9] Igor Konnov, Jure Kukovec, and Thanh-Hai Tran. TLA+ model checking made symbolic. *PACMPL*, 3(OOPSLA):123:1–123:30, 2019.

[10] Markus Alexander Kuppe, Leslie Lamport, and Daniel Ricketts. The TLA+ toolbox. In *F-IDE@FM 2019*, pages 50–62, 2019.

[11] Jae Kwon. Tendermint: Consensus without mining. *Draft v. 0.6, fall*, 1(11), 2014.

[12] Informal Systems. Fastsync – English specification, 2020. https://github.com/informalsystems/tendermint-rs/blob/master/docs/spec/fastsync/fastsync.md.

[13] Informal Systems. Fastsync – TLA$^+$ specification, 2020. https://github.com/informalsystems/tendermint-rs/blob/master/docs/spec/fastsync/fastsync.tla.

[14] Informal Systems. Verification-Driven Development: An Informal Guide, 2020. https://github.com/informalsystems/VDD/blob/master/guide/guide.md.

[15] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: BFT consensus with linearity and responsiveness. In *PODC*, pages 347–356, 2019.

Inter-blockchain protocols with the Isabelle Infrastructure framework

Florian Kammüller
Middlesex University London, UK
TU Berlin, Germany
f.kammueller@mdx.ac.uk

Uwe Nestmann
Technische Universität Berlin, Germany
firstname.secondname@tu-berlin.de

Abstract

The main incentives of blockchain technology are distribution and distributed change, consistency, and consensus. Beyond just being a distributed ledger for digital currency, smart contracts add transaction protocols to blockchains to execute terms of a contract in a blockchain network. Inter-blockchain (IBC) protocols define and control exchanges between different blockchains.

The Isabelle Infrastructure framework serves security and privacy for IoT architectures by formal specification and stepwise attack analysis and refinement. A major case study of this framework is a distributed health care scenario for data consistency for GDPR compliance. This application led to the development of an abstract system specification of blockchains for IoT infrastructures.

In this paper, we first give a summary of the concept of IBC. We then introduce an instantiation of the Isabelle Infrastructure framework to model blockchains. Based on this we extend this model to instantiate different blockchains and formalise IBC protocols. We prove the concept by defining the generic property of global consistency and prove it in Isabelle.

2012 ACM Subject Classification Networks → Peer-to-peer protocols; Networks → Security protocols; Software and its engineering → Software verification and validation

Keywords and phrases Blockchain, smart contracts, interactive theorem proving, inter-blockchain protocols

Digital Object Identifier 10.4230/LIPIcs.FMBC.2020.23

Funding *Florian Kammüller*: Research for this paper has been supported by CHIST-ERA grant 101112, SUCCESS

1 Introduction

Inter-blockchain protocols (IBC) is a concept driven by industry. It serves to provide "reliable and secure communication between deterministic processes" [23] that run on independent blockchains or distributed ledgers. Practical application of IBC are for example the Cosmos Hub [5] "the first of thousands of interconnected blockchains" with the purpose of facilitating transfers between blockchains.

A formal specification of IBC within a Higher Order Logic theorem prover like Isabelle has the advantage that it provides a very rigorous model of the IBC concepts enabling mechanically verified properties. In principle, from such a formalisation, executable code into many standard programming languages like Haskell or Scala can be generated. However, such code generation would always be understood to provide only reference implementations. Moreover, the major insights from specifying a practice oriented concept like IBC is that the formal specification is mainly useful to provide a more abstract yet more precise model that carefully picks out the central concepts used within the application, here IBC. In doing this, the used methodology, here Isabelle, can provide as a framework existing work to

2nd Workshop on Formal Methods for Blockchain.
Editors: Bruno and Diego Marmsoler; Article No. 23; pp. 23:1–23:12
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

immediately support the IBC specification. We rely heavily on the Isabelle Infrastructure framework [14] as an existing instantiation of Isabelle/HOL (which we will simply refer to as Isabelle within this paper). This framework offers a range of predefined concepts like Kripke structures and CTL, as well as state transition relations, actors, and policies that can be readily instantiated to the current application of IBC. Besides extracting a more abstract but precise specification of IBC, the resulting scientific advantage is to show that as a product of this process it becomes feasible to lay open crucial basic properties that result from the application domain (blockchain security). As the main result of this kind, we formally establish a global consistency property, define it formally on our IBC model and prove a consistency preservation theorem that shows the safety of our formal IBC semantics.

The contributions of this paper are

- summarizing the main features of IBC into a logical conceptual model,
- building a formal model of IBC in Isabelle as an instance of the Isabelle Insider framework but extending it with sets of infrastructures,
- illustrating the feasibility of the formal model by expressing a global consistency property and formally proving it in Isabelle.

The last point seems to suggest that IBC can be seen as a "blockchain of blockchains".

1.1 Inter-blockchain protocols (IBC)

In this section, we summarise the main concepts of the IBC following the practice-oriented description [23]: we refer to the relevant section of the principal documentation[23], giving precise reference to section numbers. Figure 1 is a copy an overview architectural sketch provided by the main specification [23].

```
+------------------------------+                            +------------------------------+
| Distributed Ledger A         |                            | Distributed Ledger B         |
|                              |                            |                              |
| +--------------------------+ |                            | +--------------------------+ |
| | State Machine            | |                            | | State Machine            | |
| |                          | |                            | |                          | |
| | +----------+     +-----+ | |        +---------+         | | +-----+     +----------+ | |
| | | Module A | <-> | IBC | | | <----> | Relayer | <---->  | | | IBC | <-> | Module B | | |
| | +----------+     +-----+ | |        +---------+         | | +-----+     +----------+ | |
| +--------------------------+ |                            | +--------------------------+ |
+------------------------------+                            +------------------------------+
```

Figure 1 Architecture of IBC[23].

One of the main abstractions used in IBC comprising its architectural description is the *actor* [23, Section 1.1.1] which is the same as a *user*. Instances given to exemplify this are: a human end user, a module or smart contract running on a blockchain, or an off-chain *relayer* process. This relayer process represents the logical core of the IBC. It is a process that is outside any of the blockchains (”off-chain” [23]) that is responsible for "relaying" IBC data packets between blockchains. It can scan their states and submit data.

The notion of state machine is very central in IBC: the terms *machine*, *chain*, *blockchain*, or *ledger* are used interchangeably [23, Section 1.1.2] to denote a state machine that implements part or all of the IBC. In using the Isabelle Infrastructure framework – whose core part is the formal definition of a state machine semantics through a state transition relation – we follow this important architectural spirit.

Consensus is not explicitly defined but somewhat implicitly by the notion of consensus algorithm "the protocol used by the set of processes operating a distributed ledger to come to agreement on the same state" [23, 1.1.5] where "Consensus state" is defined next as information about the "state of a consensus algorithm" [23, 1.1.6]. We can safely understand consensus to mean the agreement of the actors on the next state with respect to the state transition relation.

1.2 Isabelle Infrastructure framework

The Isabelle Infrastructure is built in the interactive generic theorem prover Isabelle/HOL [18]. As a framework, it supports formalisation and proof of systems with actors and policies. It originally emerged from verification of insider threat scenarios but it soon became clear that the theoretical concepts, like temporal logic combined with Kripke structures and a generic notion of state transitions were very suitable to be combined with attack trees into a formal security engineering process [3] and framework [9].

Figure 2 gives an overview of the Isabelle Infrastructure framework with its layers of object-logics – each level below embeds the one above showing the novel contribution of this paper in blue on the top. The formal model of IBC in Isabelle uses the Isabelle

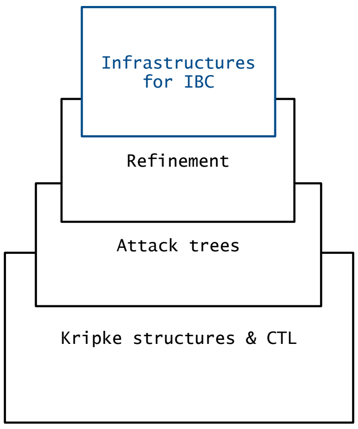


Figure 2 Generic Isabelle Infrastructure framework applied to Inter-blockchain protocols (IBC).

Infrastructure framework instantiating it by reusing its concept of *actors* for users, processes running on blockchains, or relayers running off-chain. Technically, an Isabelle theory file `IBC.thy` builds on top of the theories for Kripke structures and CTL (`MC.thy`), attack trees (`AT.thy`), and security refinement (`Refinement.thy`). Thus all these concepts can be used to specify the formal model for IBC, express relevant and interesting properties and conduct interactive proofs (with the full support of the powerful and highly automated proof support of Isabelle). The IBC theory itself is an adaptation of the Infrastructure theory of the Isabelle Infrastructure framework and reuses (or slightly adapts) existing concepts. In the remainder of this paper, we introduce the model that we conceived for IBC. All Isabelle sources are available online [11].

2 IBC in Isabelle

2.1 Overview

In the following, we give a detailed description of the central parts of the formal Isabelle theory of IBC, pointing out and motivating special design decisions. In addition to the short general intro to the Isabelle Infrastructure framework of the previous section, we provide explanations of all used Isabelle specific specification concepts on the fly.

The IBC is supposed to work for any type of blockchain, for example, Bitcoin or Ethereum, therefore the formal model abstracts from specific details of a specific blockchain. Similar to the IBC specification [23], the Isabelle formalisation focuses on the central IBC concepts as depicted in Figure 1: ledgers, actors or modules, respectively, and the relayer process interacting via the IBC protocol with the modules within the distributed ledgers. In our formal model based on the Isabelle Infrastructure framework, we represent each blockchain as an infrastructure containing nodes on which the modules (actors) are running. Data items are assigned to actors. The ledgers of each infrastructure keep control over the data items. That is, a ledger is a unique assignment that controls the access to a data item and keeps a record of where the data item resides within this and other blockchains. The IBC enables just that: a unified view over a whole range of heterogeneous blockchains that exchange data consistently. Therefore, our formal model goes beyond the usual application of the Isabelle Infrastructure framework, e.g. [8], and considers sets of infrastructures (representing different blockchains).

2.2 Ledgers

Actors are a general concept provided by the Isabelle Infrastructure framework and can be used directly to represent the *actor* concept in IBC.

```
typedecl actor
type_synonym identity = string
consts Actor :: string ⇒ actor
```

Similar to the general Infrastructure framework, actors can perform actions. However, in this instantiation to IBC we redefine the actions representing the central activities of the relayer scanning each blockchain's state and submitting transactions (see Section 2).

```
datatype action = scan | submit
```

The Decentralised Label Model (DLM) [16] allows labelling data with owners and readers. We also adopt this definition of security labeled data as already formalised in [9]. Labelled data is given by the type `dlm × data` where `data` can be any data type.

```
type_synonym data = string
type_synonym dlm = identity × identity set
```

One major achievement of a blockchain is that it acts like a distributed ledger, that is, a global accounting book. A distributed ledger is a unique consistent transcript keeping track of protected data across a distributed system. In our application, the ledger must mainly keep track of where the data resides for any labelled data item. To express the system requirement that processing may not change the security and privacy labels of data, we introduce a type of security and privacy preserving functions.

```
typedef label_fun = {f :: dlm × data ⇒ dlm × data.
                           ∀ x. fst x = fst (f x)}
```

We formalize a ledger thus as a type of partial functions that maps a data item to a pair of the data's label and the set of locations where the data item is registered. Since all function in HOL are total, we use a standard Isabelle way of representing partial functions using the type constructor `option`. This type constructor lifts every type α to the type α `option` which consists of the unique constant `None` and the range of elements `Some x` for all `x`$\in \alpha$.

```
type_synonym ledger = data ⇒ (dlm × node set)option
```

Since the type `ledger` is a function type, it automatically constrains each data item `d` in its domain to have at most one range element `Some(l,N)`, that is, at most one valid data label `l` of type `dlm` and a list of current blockchain nodes `N` at which this data item is transcribed.

```
lemma ledger_def_prop: ∀ lg:: ledger. ∀ d:: data.
     lg d = None | (∃! l. (∃! L. lg d = Some(l, L)))
```

In an earlier application of the Isabelle Infrastructure framework to IoT security and privacy[14], we established a formal notion of blockchain. However, there we used a more explicit logical characterisation in an Isabelle type definition which creates additional proof effort and makes formulas more complex. The current representation of the ledger type as a partial function type is more concise and implicitly carries the requested uniqueness properties. Note that the defining property of the ledger type is now proved from the used type constructors by the above lemma instead of being specified into the type as in the earlier formalisation [14].

2.3 Infrastructures as blockchains

As smart contracts `sc_fun` we formalise any action that is sent or received between different blockchains and may have effects on the labelled data. Therefore the inputs to the send and receive messages are two identities of sender and receiver as well as the dlm label and the concerned data.

```
datatype sc_fun = Send identity × identity × dlm × data
                | Receive identity × identity × dlm × data
```

In addition to specifying the potential types of smart contracts, we need to provide a way of keeping track of the transactions that are executed within a blockchain. To this end, we define the following type of `transaction_record` which is a list of all executed smart contracts.

```
type_synonym transaction_record = sc_fun list
```

The central component that builds the system state is an infrastructure. Since we use the Isabelle Infrastructure framework, we consider blockchains as infrastructures. The essential architecture of such an infrastructure is a simple graph of blockchain nodes on which the processes (actors) reside given as the first component `(node ×node)set` of the below datatype `igraph`. Besides this basic architecture, this infrastructure graph also stores the other components of the blockchain. The second input is a function that assigns a set of actor identities to each node in the graph representing the current location of the actors. The next input associates actors to a pair of string sets by a pair-valued function whose first range component is a set describing the credentials in the possession of an actor and the second component is a set defining the roles the actor can take on. An infrastructure graph also allows assigning a string to each location to represent some current state information of that location. Finally, the ledger is added as a separate component as well as the transaction record.

```
datatype igraph =
        Lgraph (node × node)set
               node ⇒ node set
               actor ⇒ (string set × string set)
```

```
                  node ⇒ string
                  ledger
                  transaction_record
```

Corresponding projection functions for each of the components of an infrastructure graph are provided. They are omitted here for brevity but are available in the online version [11]); they are named `gra` for the actual set of pairs of locations, `agra` for the actor map, `cgra` for the credentials, and `lgra` for the state of a location `ledgra` for the ledger component in the graph and `trec` for the transaction record. Infrastructures contain an infrastructure graph and a policy given by a function that assigns local policies over a graph to all locations of the graph.

```
datatype infrastructure =
  Infrastructure  igraph
                 [igraph, location] ⇒ apolicy set
```

There are projection functions `graphI` and `delta` when applied to an infrastructure return the graph and the policy, respectively.

Policies specify the expected behaviour of actors of an infrastructure. We define the behaviour of actors using a predicate `enables`: within infrastructure `I`, at location `l`, an actor `h` is enabled to perform an action `a` if there is a pair `(p,e)` in the local policy of `l` – `delta I l` projects to the local policy – such that action `a` is in the action set `e` and the policy predicate `p` holds for actor `h`.

```
enables I l h a = ∃ (p,e) ∈ delta I l. a ∈ e ∧ p h
```

Compared to the applications of the Isabelle Infrastructure framework, e.g. [8], we do not make use of policies to model the constraints of our application. However different to previous applications, the IBC challenges the framework in other ways leading to slight extensions.

2.4 Relayer and set of blockchains

To model the relayer, we also use infrastructures: the relayer is a distinguished infrastructure. It could be thought of as another distributed application with various relayer processes to avoid bottlenecks but for simplicity, we assume that there is one specific actor `''relayer''` that resides on a specific node in the relayer infrastructure.

We express protocols as traces of execution steps of IBC transaction steps, that is, lists of smart contracts `sc_fun` (see previous section). Using traces of execution steps to represent protocols, follows the classical method of the inductive approach to security protocol verification originally devised by Paulson [21] and already successfully used for the Isabelle Infrastructure framework, for example, [12] and more recently [10].

```
datatype ibc_protocol = Protocol sc_fun list set
```

The datatype `blockchainset` puts together the IBC protocol as a triple: as the first element it includes the IBC protocol, the second element is the list of infrastructures where each element is one blockchain involved in the IBC, and the third element is a single distinguished infrastructure, the relayer.

```
datatype blockchainset = Infs ibc_protocol
                              infrastructure list
                              infrastructure
```

To round off these new datatypes, we provide additional projection functions and constructors. For a given blockchain Il, the projection trcs Il returns the sc_fun list set representing the protocol, the projection the_Il returns the list of infrastructures of all involved blockchains, and relayer Il gives the distinguished infrastructure, the third element, which is the relayer infrastructure. To facilitate handling of data transactions, we define some update functions: the function application upd_ld d lN I updates a ledger at the data point d to now contain the pair lN of a dlm label and a set of nodes of residences of the data. Scaling this up to the level of infrastructures, the function application upd_Il d lN Il updates all blockchains in the infrastructure list of the blockchainset Il using the former ledger update upd_ld. A function replace allows to replace an infrastructure I in a blockchainset Il. See the online resources [11] for technical details and implementations of these definitions.

2.5 Consensus

The consensus algorithm may be different for each blockchain employed in the IBC. Therefore, we cannot make any assumptions at the general specification level of the IBC about it. Yet, we still want to use it in the description of the IBC protocol semantics. Therefore, we apply a trick: we declare Consensus to be a constant at the level of the specification of the IBC.

```
consts Consensus :: infrastructure ⇒ blockchainset ⇒ blockchainset
```

In Isabelle this means that Consensus is a function mapping an infrastructure and a system state of type blockchain to blockchain but there is no semantics attached to this constant. The constant is part of the theory IBC.thy and can be used in it like any other defined element but it has no meaning. However, a semantics can be later attached to it in an application of the IBC theory to specific blockchains. This could be done in the current context for example using a definition in a locale [13].

```
locale ConsensusExample =
fixes cons_algo :: infrastructure ⇒ blockchainset ⇒ infrastructure
defines cons_algo_def: cons_algo I Il = ...
fixes Consensus :: infrastructure ⇒ blockchainset ⇒ blockchainset
defines Consensus_def: Consensus I Il = replace (cons_algo I Il) I Il
```

The predicate Consensus redefines the semantics within the locale ConsensusExample. The first locale definition is omitted here for simplicity. We could imagine that it is a description of a consensus algorithm that can depend on all the state constituents, like actors, nodes, and policies of the blockchain I but also of the surrounding blockchainset including the relayer state and the current protocol state. The definition of the constant Consensus lifts the algorithm to the blockchain by using the replace function defined as part of the infrastructure for blockchainsets (see Section 2.4 or refer to the Isabelle code [11]).

2.6 IBC state transition semantics

The semantics of the IBC state machines is defined by a state transition relation over blockchain sets. That is, we define a syntactic infix notation Il → Il' to denote that blockchain sets Il and Il' are in this relation.

```
inductive state_transition_in ::
       [blockchainset, blockchainset] ⇒ bool "(_ → _)"
```

The rules of the inductive definition state_transition_in allow the definition of the intended behaviour of the relayer scanning an arbitrary blockchain (see Section 2). The relayer stores the results in its own transaction record. The following rule scan is the first of two inductive definition rules defining the transition relation →: if an infrastructure I is in the blockchainset Il, the actor (process, module) resides at node n in the graph G of I; R is the relayer and thus enabled to scan. The follow up state Il' of Il is given by extending any current protocol trace l using the specially defined function insertp by the transaction Send(a,b,(a,as), d). Also the relayer's trace record trec R is extended by the same transaction.

```
scan : inbc I Il ⟹ G = graphI I ⟹ a @G n ⟹ n ∈ nodes G ⟹
       R = graphI (relayer Il) ⟹ r @R n' ⟹ n' @R nodes R ⟹
       relrole (relayer Il) (Actor r) ⟹
       enables I n (Actor r) scan ⟹
       ledgra G d = Some ((a, as), N) ⟹ r ∈ as ⟹
       R' = Infrastructure
                 (Lgraph (gra R)(agra R)(cgra R)(lgra R)
                         ((ledgra R)(d := Some((a', as),N)))
                         ((Send(a,b,(a,as), d)) # (trec R)))
                 (delta (relayer Il)) ⟹
       l ∈ trcs Il ⟹ Consensus I Il = Il' ⟹
       Il' = insertp ((Send(a,b,(a,as), d)) # l) (replrel R' Il)
       ⟹ Il → Il'
```

Additionally, the relayer can submit data onto an arbitrary blockchain (see Section 2). The second rule submit of → defines its semantics: between the infrastructures I and J which are both in the blockchain set Il the relayer R can submit data d from an owner a to an owner b if the ledger component ledgra R of the relayer's infrastructure R is updated to the new owner in both blockchains. The update is achieved using the function update := of Isabelle's function theory updating the point d to the new value Some((b, bs), N). In the construction of the next state blockchainset Il' the specially defined update operators mentioned in Section 2.4 are used: replrel for updating the relayer and bc_upd for the infrastructure list representing the "client" blockchains. Note the latter realizes the consistent update in both involved infrastructures I and J.

```
submit : G = graphI I ⟹ inbc I Il ⟹ a @G n ⟹ n ∈ nodes G ⟹
         ledgra G d = Some ((a, as), N) ⟹
         H = graphI J ⟹ inbc J Il ⟹ b @H n' ⟹ n' ∈ nodes H ⟹
         ledgra H d = Some ((a, as), N) ⟹
         R = graphI (relayer Il) ⟹ r @R n'' ⟹ n'' ∈ nodes R ⟹
         relrole (relayer Il) (Actor r) ⟹
         enables J n' (Actor r) submit ⟹
         r ∈ as ⟹
         R' = Infrastructure
               (Lgraph (gra R)(agra R)(cgra R)(lgra R)
                       ((ledgra R)(d := Some((b, bs),N)))
                       ((Receive(a,b,(a,as), d)) # (trec R)))
             (delta (relayer Il))  ⟹
         Il' = insertp (Receive(a,b,(a,as),d)# l)
                       (replrel R' (bc_upd d ((b,as), N) Il)) ⟹
         Consensus (actors H) = Il
         ⟹ Il → Il'
```

The real advantage of the Isabelle Infrastructure framework comes into play when using the possibility of instantiation of axiomatic type classes provided by Isabelle. Since state transitions have been defined by an axiomatic type class in the framework within the theory for Kripke structures and CTL, we can now instantiate blockchainsets as state and thereby inherit the entire logic, constructors and theorems.

```
instantiation blockchainset :: state
```

3 Global consistency

To illustrate the use of the abstract formal model of IBC presented in this paper, we show that we can exhibit an important property: global consistency. That is, if the IBC scans and submits between blockchains it must not introduce inconsistencies.

Expressing this property alone represents a proof of concept since it shows that our IBC model is detailed enough to capture explicitly the notion of consistent data representation across different blockchains. *Proving* the property is a non-trivial contribution (see proof scripts [11]) that helped exhibiting a range of useful auxiliary definitions and lemmas as we will highlight in this section when discussing the global consistency theorem. The proofs were greatly helped by the recent advances in proof automation in Isabelle using sledgehammer [20]. The fact that the property is provable shows that the model and in particular its semantics conform to the intuition described in [23]. The formalisation and proof also highlight the pros and cons of our model as discussed in the Conclusions in Section 4.

We first define global consistency as the property that the individual ledgers in each blockchain in an IBC blockchainset agree on the data, that is, they all hold consistent information about the access control of the data (the first part of type `dlm` of the `ledgra` output (see Section 2.2)) and where the data resides: the set of nodes that are the second component of the `ledgra` output.

```
Global_consistency Il = (∀ I I'. inbc I Il → inbc I' Il →
          (∀ d. (ledgra (graphI I') d) = (ledgra (graphI I) d)))
```

The theorem shows that if global consistency holds, then a step of the state transition does preserve it.

```
theorem consistency_preservation:
global_consistency Il ⟹ (Il → Il') ⟹ global_consistency Il'
```

Preservation of global consistency guarantees that any transaction happening within IBC preserves one consistent view over all data, their access control, and residence. If initially data is not visible on all blockchains, not all ledgers are equal. However, if eventually data has travelled across, all ledgers become the same: the blockchainset becomes like one blockchain: a “blockchain of blockchains”.

4 Conclusions, related work, and outlook

In this paper, we have provided an abstract formal model of the Inter-blockchain protocol (IBC) [23] as an instantiation of the Isabelle Infrastructure framework. We have detailed the formal presentation in Isabelle and the extensions to the Isabelle Infrastructure framework, most notably by defining sets of (heterogeneous) blockchains including protocols and a distinguished relayer. The abstraction we conceived for this model has been first validated by a proof of concept by sketching how the abstract notion of Consensus can be instantiated by a locale (Section 2.5). Furthermore, we have defined a global consistency property over blockchainsets proving that our abstraction yields the desired expressivity (Section 3). We have proved a preservation theorem for global consistency in Isabelle. Summarising, our model allows to prove meta-theoretical results but is not too abstract to allow instantiation onto concrete blockchains and their Consensus algorithms. As a more general thought, the dealings with global consistency seem to suggest that IBC creates a blockchain of blockchains.

4.1 Related Work

Relevant examples for the investigation of formal support for blockchains and smart contracts can be found in abundance in the proceedings of the first FMBC workshop [2]. We only discuss the few most closely related ones from there since others are either focusing on specific blockchains (unlike the generic IBC we consider) or are differing in the formal approach (not using theorem provers and thus not addressing the same level of expressivity and assurance).

A range of works formalises smart contracts typical for the Ethereum virtual machine. For example, using the K framework [22], the Lem language [7], and F* [6]. We focus here on the work that has been performed in the K-framework [22]. The K-framework is a semantics framework enabling to produce executable operational semantics for programming languages. K also provides tools like parsers, interpreters, model-checkers and program verifiers. It has been applied to provide a verification environment for the Ethereum Virtual Machine EVM [19] which is useful for verifying programme modules within Ethereum's smart contract systems, for example, Ethereum's Name Service (ENS) [24].

In comparison to those dedicated verification environments for specific blockchains, like Ethereum, our formal model strongly abstracts from technical detail. This abstraction is necessary to accommodate a global view that allows to reason about the communication between a heterogeneous set of blockchains.

A few works use model checkers and SMT solvers, for example [4]. Deductive verification platforms like Why3 [11,13] have been also used for smart contracts. Interactive proof assistants (e.g. Isabelle/HOL or Coq) have been used before for modeling and proving properties about Ethereum and Tezos smart contracts [1].

Very related is the work by Nielsen and Spitters on Smart Contract Interactions in Coq [17]. The authors construct a model of smart contracts that allows for inter-contract communication generalising over depth-first execution blockchains like Ethereum and breadth-first execution blockchains like Tezos. They use Coq's functional language Galina to express smart contracts. Besides the obvious difference of being a Coq development rather than an Isabelle development, we address the high level protocol language IBC instead of focusing on generalised smart contracts.

Maybe even more closely related is the work on the specification of the dedicated security framework Cap9 in Isabelle [15]. Compared to us it focuses again on the expression of smart contracts and does not have the inter-blockchain aspect like our IBC.

4.2 Outlook

The global consistency preservation theorem proves the concept of the IBC specification and also shows that the formalisation in itself is a useful experiment: extracting a closed abstract model of the IBC from the technical specification [23] has immediately produced the consistency question. The abstraction allowed to define semantics in which a strong global consistency theorem could be proved within Isabelle in reasonably short time. It should be understood that these are first steps that mainly serve to prove the concept of using the Isabelle Infrastructure framework for advancing the IBC. A clear next step is to elaborate the sketched application example of Section 2.5 of a concrete blockchain and its consensus algorithm. A much more challenging next step is to refine the model by elaborating a more concrete IBC protocol example by instantiation of the `ipc_prot` component of the `blockchainset` type. This would be a fruitful future avenue for applied research in collaboration with the designers of IBC.

References

1 S. Amani, M. Bégel, M. Bortin, and M. Staples. Towards verifying ethereum smart contract bytecode in isabelle/hol. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 66–77. ACM, 2018.

2 N. Catano, D. Marmsoler, and B. Bernardo, editors. *Pre-proceedings of the First Workshop on Formal Methods for Blockchains, FMBC*, 2019. Selected papers to appear in Springer LNCS. URL: `https://sites.google.com/view/fmbc`.

3 CHIST-ERA. Success: Secure accessibility for the internet of things, 2016. http://www.chistera.eu/projects/success.

4 Sylvain Conchon, Alexandrina Korneva1, and Fatiha Zaidi. Verifying smart contracts with cubicle. In Catano et al. [2]. Selected papers to appear in Springer LNCS. URL: `https://sites.google.com/view/fmbc`.

5 Cosmos. Cosmos hub, 2020. accessed 23.1.2020. URL: `https://hub.cosmos.network/master/hub-overview/overview.html`.

6 I. Grishchenko, M. Maffei, and C. Schneidewind. A semantic framework for the security analysis of ethereum smart contracts. In L. Bauer and R. Ksters, editors, *Principles of Security and Trust*, Lecture Notes in Computer Science, pages 243–269. Springer, 2017.

7 Y. Hirai. Defining the ethereum virtual machine for interactive theorem provers. In M. Brenner, K. Rohloff, J. Bonneau, A. Miller, P. Y. Ryan, V. Teague, A. Braccialiand M. Sala, F. Pintore, and M. Jakobsson, editors, *Financial Cryptography and Data Security*, Lecture Notes in Computer Science, pages 520–535. Springer, 2017.

8 F. Kammüller. Formal modeling and analysis of data protection for gdpr compliance of iot healthcare systems. In *IEEE Systems, Man and Cybernetics, SMC2018*. IEEE, 2018.

9 F. Kammüller. Combining secure system design with risk assessment for iot healthcare systems. In *Workshop on Security, Privacy, and Trust in the IoT, SPTIoT'19, colocated with IEEE PerCom*. IEEE, 2019.

10 F. Kammüller. Qkd in isabelle – bayesian calculation. *arXiv*, cs.CR, 2019. URL: `https://arxiv.org/abs/1905.00325`.

11 F. Kammüller. Isabelle infrastructure framework for ibc, 2020. Isabelle sources for IBC formalisation. URL: `https://github.com/flokam/IsabelleSC`.

12 F. Kammüller, M. Kerber, and C.W. Probst. Towards formal analysis of insider threats for auctions. In *8th ACM CCS International Workshop on Managing Insider Security Threats, MIST'16*. ACM, 2016.

13 F. Kammüller, M. Wenzel, and L. C. Paulson. Locales – a sectioning concept for Isabelle. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99*, volume 1690 of *LNCS*. Springer, 1999.

14 Florian Kammüller. A formal development cycle for security engineering in isabelle, 2020. `arXiv:2001.08983`.

15 Mikhail Mandrykin1, Jake O'Shannessy, Jacob Payne, and Ilya Shchepetkov. Formal specification of a security framework for smart contracts. In Catano et al. [2]. Selected papers to appear in Springer LNCS. URL: `https://sites.google.com/view/fmbc`.

16 A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, 1999.

17 J. Botsch Nielsen and B. Spitters. Smart contract interactions in coq. In Catano et al. [2]. Selected papers to appear in Springer LNCS. URL: `https://sites.google.com/view/fmbc`.

18 T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.

19 Daejun Park, Yi Zhang, Manasvi Saxena, Philip Daian, and Grigore Roundefinedu. A formal verification tool for ethereum vm bytecode. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 912–915, New York, NY, USA, 2018. Association for Computing Machinery. URL: `https://doi.org/10.1145/3236024.3264591`, `doi:10.1145/3236024.3264591`.

20 Lawrence Paulson and Jasmin Blanchette. Three years of experience with sledgehammer, a practical link between automatic and interactive theorem provers. *Proceedings of the 8th International Workshop on the Implementation of Logics*, 02 2015. `doi:10.29007/tnfd`.

21 Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1-2):85–128, 1998.

22 Grigore Rosu. Specifying languages and verifying programs with k. *2013 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 28–31, 2013.

23 The IBC Specification Team. The interblockchain communication protocol. Technical report, 2020. 4th May 2020 — 1.0.0-rc5. URL: `https://github.com/cosmos/ics/blob/master/spec.pdf`.

24 Duy Minh Vo. Verification of smart contracts using the k-framework, 2018.

Part IV.

Lightning talks

Verifying, testing and running smart contracts in ConCert

Danil Annenkov, Mikkel Milo, Jakob Botsch Nielsen, and Bas Spitters

Concordium Blockchain Research Center, Aarhus University

Introduction Smart contracts are programs running on top of a blockchain. They often control big amounts of cryptocurrency and cannot be changed after deployment. Unfortunately, many vulnerabilities have been discovered in smart contracts and this has lead to huge financial losses (e.g. TheDAO, Parity's multi-signature wallet). So, smart contract verification is becoming increasingly important. Functional smart contract languages are becoming increasingly popular: e.g. Simplicity [13], Liquidity[1], Plutus [6], Scilla [14] and Midlang[2]. A contract in such a language is just a function from a message type and a current state to a new state and a list of actions (transfers, calls to other contracts), making smart contracts more amenable for formal verification. We build on the ConCert framework [3] for embedding smart contracts in Coq and the execution model introduced in [12]. In the present work, we extend ConCert with an extraction functionality, implement anonymous voting based on the Open Vote Network protocol and integrate property-based testing using QuickChick [9].[3]

Extraction The Coq proof assistant features the *extraction* functionality [10]. Extraction allows for generating a program in OCaml, Haskell or Scheme from a program in Coq. This functionality thus enables proving properties of functional programs in Coq and then automatically producing code in one of the supported languages that could be integrated with existing developments. Coq allows also for encoding properties of a program in the program's type using dependent types. This expressivity makes the extraction procedure non-trivial, requiring to distinguish the parts that are relevant for computation from the computationally irrelevant ones ("logical" parts). Recent projects such as MetaCoq [15] and CertiCoq [1] provide formal guarantees that the extraction procedure is correct, that is, the computational properties of the erased terms are preserved. We extend on the work on the certified erasure [15] and implement a simple optimisation procedure for removing some redundant constructs left after the erasure step. This procedure allows for easier integration with target language primitives. After the simplification step, the code is pretty-printed to a functional smart contract language. Currently, we support Liquidity and Midlang as target languages, but the technique applies to the other languages mentioned above.

As an example, let us consider a simple counter contract with the state being just an integer number (represented using the type `Z` from the standard library of Coq) and accepting increment and decrement messages: `counter : msg → Z → option (list action ∗ Z)`. The main functionality is given by the two functions `inc_counter` (see below) and `dec_counter` which are called from `counter` depending on a message.

```
Program Definition inc_counter (st : Z) (n : {z : Z | 0 < z}) : {new_st : Z | st < new_st} :=
  exist (st + proj1_sig n) _. Next Obligation. (* proof goes here *) Qed.
```

We use *refinement types* to encode some invariants of these functions. E.g. `inc_counter` takes the current state and a positive integer. It adds the number to the current state and returns the next state along with a proof that it is greater than the current state. This way, we capture a specification of `inc_counter` in its type (i.e. the function indeed increments the state). Refinement types are implemented in Coq using dependent pairs with the second component being a proof. The constructor `exist` is used to construct a value of such a dependent pair and `proj1_sig` to project the first component. We give the first component (a number) explicitly and leave a placeholder for the proof which we fill in using Coq's tactics.

```
let exist a = a
let inc_counter (st : storage) (n : int) =
  exist (addInt st ((fun x → x) n))
```

Listing 1: Liquidity

```
proj1_sig : Sig a → a
proj1_sig e = case e of Exist a → a
inc_counter : Z → Sig Z → Sig Z
inc_counter st n =
  Exist (add st (proj1_sig n))
```

Listing 2: Midlang

As one can see from the listings above, the extraction procedure removes all "logical" parts (proof terms) from the original Coq code. In the original Coq code, `inc_counter` is called from the `counter` function

[1]https://www.liquidity-lang.org/

[2]https://developers.concordium.com/midlang

[3]Our development is available at https://github.com/AU-COBRA/ConCert

(not shown here) which performs input validation and constructs the argument of type {z : Z | 0 < z}. In the extracted code, the only way of interacting with the contract is by calling `counter`. Therefore, it is safe to execute `inc_counter` without additional input validation.

We successfully applied the developed extraction to several variants of a counter contract, to the crowdfunding contract described in [3] and to an interpreter for a simple expression language. The latter example shows the possibility of extracting certified interpreters for domain-specific languages such as Marlowe [8] and CL [4, 2] representing an important step towards safe smart contract programming.

Boardroom Voting Hao, Ryan and Zielisky developed the Open Vote Network protocol [7], an e-voting protocol that allows a small number of parties ('a boardroom') to vote anonymously on a topic. Their protocol allows tallying the vote while still maintaining maximum voter privacy, meaning that each vote is kept private unless all other parties collude. Each party proves with zero-knowledge to all other parties that they are following the protocol correctly and that their votes are well-formed.

This protocol was implemented as an Ethereum smart contract by McCorry, Shahandashti and Hao [11]. In their implementation, the smart contract serves as the orchestrator of the vote by verifying the zero-knowledge proofs and computing the final tally.

We implement a similar contract in the ConCert framework. The original protocol works in three steps. First, there is a sign-up step where each party submits a public key and a zero-knowledge proof that they know the corresponding private key. After this, each party publishes a commitment to their upcoming vote. Finally, each party submits a computation representing their vote, but from which it is computationally intractable to obtain their actual private vote. Together with the vote, they also submit a zero-knowledge proof that this value is well-formed, i.e. it was computed from their private key and a private vote (either 'for' or 'against'). After all parties have submitted their public votes, the contract is able to tally the final result. For more details, see the original paper [7].

Listing 1 shows the message type used in the contract. Here, `A` is an element in an arbitrary finite field, `Z` is the type of integers and `positive` can be viewed as the type of finite bit strings.

```
Inductive Msg :=
| signup (pk : A) (proof : A * Z)
| commit_to_vote (hash : positive)
| submit_vote (v : A) (proof : VoteProof)
| tally_votes.
```

Listing 1: The message type for the boardroom voting contract.

The contract provides three functions `make_signup_msg`, `make_commit_msg` and `make_vote_msg` meant to be used off-chain by each party to create the messages that should be sent to the contract. We prove that our contract cannot compute the wrong tally under the assumption that all parties used these functions. Also, we prove that when these functions are used, the zero-knowledge proofs attached will be verified correctly by the contract. Note that, due to this verification done by the contract, the contract is able to detect if a party misbehaves. However, we do not prove formally that incorrect proofs do not verify since this is a probabilistic statement better suited for tools like EasyCrypt [5].

Since the tallying and the zero-knowledge proofs are based on finite field arithmetic we develop some required theory about $\mathbb{Z}_p$ including Fermat's theorem and the extended Euclidean algorithm. This allows us to instantiate the boardroom voting contract with $\mathbb{Z}_p$ and test it inside Coq using ConCert's executable specification. To make this efficient, we use the Bignums library of Coq to implement operations inside $\mathbb{Z}_p$ in an efficient way.

We are currently working on using the extraction mechanism described above to extract and run the boardroom voting contract on existing blockchains. One challenge is to make it efficient enough to run on blockchains without being prohibitively expensive. Indeed, the Ethereum version by McCorry, Shahandashti and Hao [11] uses elliptic curves instead of finite fields to achieve the same security guarantees with much smaller key sizes and therefore more efficient computation. In the future, we expect to make the same improvement for our version to be practically applicable on the blockchains we target.

Testing smart contracts With ConCert's executable specification our contracts are fully testable from within Coq. This enables us to integrate property-based testing into ConCert using QuickChick. This serves as a cost-effective, semi-formal, semi-automated approach to discover bugs and increases reliability that the implementation is correct. It may be used either as a preliminary step to support formal verification or as a complementary approach whenever the properties become too involved to prove.

The testing framework is semi-automated in the sense that the user must implement a *generator* function for the message type of the contract they want to test, i.e. a function which generates "arbitrary" messages to be sent to the contract. The framework then generates thousands of "arbitrary" blockchain execution traces and uses QuickChick to test if the supplied properties hold.

We demonstrate the usability of the framework by testing complex contracts such as the Congress contract (the essence of TheDAO), ERC-20 tokens, Tezos FA2 token standard,[4] and the UniSwap token exchange platform. The testing framework currently supports testing of functional properties, as well as temporal properties (i.e. involving reachability of states). With our development, we successfully discovered well-known vulnerabilities in ERC-20 compliant tokens, reentrancy in the Congress contract, and a recently discovered reentrancy vulnerability in the UniSwap protocol.

References

[1] Abhishek Anand et al. “CertiCoq: A verified compiler for Coq”. In: *CoqPL’2017*.

[2] Danil Annenkov and Martin Elsman. “Certified Compilation of Financial Contracts”. In: *PPDP’2018*.

[3] Danil Annenkov, Jakob Botsch Nielsen, and Bas Spitters. “ConCert: A Smart Contract Certification Framework in Coq”. In: *CPP’2020*.

[4] Patrick Bahr, Jost Berthold, and Martin Elsman. “Certified Symbolic Management of Financial Multi-Party Contracts”. In: *SIGPLAN Not.* (2015).

[5] Gilles Barthe et al. “EasyCrypt: A Tutorial”. In: *Foundations of Security Analysis and Design VII: FOSAD 2012/2013 Tutorial Lectures*. Ed. by Alessandro Aldini, Javier Lopez, and Fabio Martinelli. 2014.

[6] James Chapman et al. “System F in Agda, for fun and profit”. In: *MPC19*. 2019.

[7] Feng Hao, Peter YA Ryan, and Piotr Zieliński. “Anonymous voting by two-round public discussion”. In: *IET Information Security* 4.2 (2010).

[8] Pablo Lamela Seijas and Simon Thompson. “Marlowe: Financial Contracts on Blockchain”. In: *International Symposium on Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*. Ed. by Tiziana Margaria and Bernhard Steffen. 2018.

[9] Leonidas Lampropoulos and Benjamin C. Pierce. *QuickChick: Property-Based Testing in Coq*. Software Foundations series, volume 4. 2018.

[10] Pierre Letouzey. “Programmation fonctionnelle certifiée – L’extraction de programmes dans l’assistant Coq”. PhD thesis. Université Paris-Sud, 2004.

[11] Patrick McCorry, Siamak F Shahandashti, and Feng Hao. “A smart contract for boardroom voting with maximum voter privacy”. In: *FC 2017*.

[12] Jakob Botsch Nielsen and Bas Spitters. “Smart Contract Interactions in Coq”. In: *FMBC’2019*.

[13] Russell O’Connor. “Simplicity: A New Language for Blockchains”. In: PLAS17.

[14] Ilya Sergey et al. “Safer Smart Contract Programming with Scilla”. In: *OOPSLA19*. 2019.

[15] Matthieu Sozeau et al. “Coq Coq Correct! Verification of Type Checking and Erasure for Coq, in Coq”. In: *POPL’2019*.

[4]https://medium.com/@TQTezos/introducing-fa2-a-multi-asset-interface-for-tezos-55173d505e5f

Summary of Bitcoin Trace-Net: Contract Verification at Signing Time

James Chiang

DTU, Kongens Lyngby, Denmark
jchi@dtu.dk

Abstract

Bitcoin contracting protocols regulate the transfer of bitcoins amongst participants in a trustless manner. A safe and secure contract design should feature collaborative execution traces, but also allow the verifying actor to reach a safe terminal state despite any potentially adversarial actions by the counter-party. Trace-Net[1] is a proposed Petri Net formalism extended with a stateful actor knowledge model. A Trace-Net model is lifted from raw Bitcoin contract transactions, enabling the contract to be verified at the raw Bitcoin transaction level: Verification can therefore be performed at at run-time, and does not require manually generated contract specifications. This extended abstract summarizes the recently published Bitcoin contract verification method named Trace-Net, which is sufficiently expressive for the verification of complex contracts including off-chain state channels and contracts featuring cryptographic sub-protocols unobservable on the blockchain.

1 A Model of Contracting Protocols

Contracting protocol designs in Bitcoin such as payment channel networks [1], generalized state channels [2] or coin join [3] variants offer both collaborative or alternative, unilateral protocol execution paths. Trace-Net [4] semantics model such protocols as actor strategies which includes sending messages through direct channels to counter-parties or broadcasting contract transactions to the Bitcoin network. Each verifying actor assumes that all other actors are potentially colluding against it: Therefore, Trace-Net considers all contracting counter-parties as a single, external actor with consolidated knowledge. The modeled knowledge of both internal and external actors are updated with signatures, secret hash pre-images or other transaction attributes when direct messages are exchanged and new transactions are observed on the Bitcoin network or Bitcoin blockchain. Similar to the Dolev-Yao [5] model, actors in Trace-Net have access to the same set of public functions such as signing, hashing, and extraction of transaction attributes which defines their ability to deduce knowledge from new observations.

Bitcoin Network & Blockchain Model The internal actor can always propagate known, valid contract transactions to the Bitcoin network and adjust the transaction feerate to ensure it is amended to the blockchain within a maximum block interval, unless a conflicting contract transaction can be deduced from the observing, external actor's knowledge. In Trace-Net, such a transaction race is always decided by the external actor, who is presumed to outbid the original transaction feerate.

Trustless Execution Property The contract execution state-space is necessarily finite since initial and derived knowledge of the participating actors is bounded. Therefore, trace or temporal properties of the contract execution are decidable by the verifier. In particular, we propose a definition of *trustless execution*, which guarantees that the verifying actor can always safely

[1]Under submission. Earlier version presented at MIT Cryptoeconomic Systems 2020. Full paper can be found at https://arxiv.org/abs/2007.07528.

terminate the contract despite any adversarial action by the counter-party. In practice, this implies that a safe contract design will feature non-collaborative abort traces executable by the verifying actor if the adversarial counter-party refuses to execute the expected, collaborative execution trace.

2 Lifting a Symbolic Model from Raw Transactions

A contracting protocol is executed in two phases. In the first *setup* phase, key pairs and secret hash pre-images are generated by the participants, and unsigned, raw transaction templates negotiated. In the subsequent *execution* phase of the contract protocol, messages are exchanged between counter-parties and transaction templates are signed and completed for broadcast and appended to the Bitcoin blockchain.

In order to construct a protocol state machine from the set of raw transaction templates negotiated after the *setup phase*, the symbolic spending paths of every individual Bitcoin output script in the contract transactions must be deducible. This has been demonstrated with a constraint solving approach by Klomp & Bracciali [6] for a fragment of the Bitcoin script language. Alternatively, this can be achieved by implementing Bitcoin contracts in Miniscript [7] [8], a typed template language which expresses a commonly used subset of Bitcoin Script. Critically, Miniscript defines semantics for the symbolic execution of an output script composed from the composable Bitcoin Script templates supported by Miniscript.

Trace-Net Once the symbolic execution paths of all contract transaction outputs are deduced, an extended Time Petri net [9] representation of the contract protocol can be constructed. Trace-Net extends the Time Petri net with Bitcoin timelock semantics [10], which prevent Bitcoin transactions to be amended before a certain blockheight and is enforced by Bitcoin consensus rules. Timelock semantics are critical in enforcing an order of specific strategies in Bitcoin contract designs. Since Bitcoin transactions are signed, completed and broadcast by actors, the Trace-Net model must also feature actor knowledge states: A valid on-chain, contract transition can only be fired if an actor's knowledge allows the valid Bitcoin transaction with the required input signatures and hash pre-images to be deduced. The Trace-Net model is unfolded into a state graph representation to verify the underlying contracting protocol for the aforementioned *trustless execution* property and other trace properties of interest. Transitions in the state graph include direct messages between actors, the passing of time intervals and the broadcast and confirmation of Bitcoin contract transactions.

3 Related Work & Discussion

Trace-Net is proposed to verify UTXO-based smart contracts at the protocol transaction level. In contrast, BitML [11] [12] [13] by Bartoletti & Zunino is an executable contract specification language amenable to model checking [14] at the symbolic language level: Contract specification properties are guaranteed to translate to the raw Bitcoin transaction level when compiled. This approach, however, trades off control over the on-chain footprint for compiler-guaranteed security: Implementing contracts directly at the transaction-level can result in lower execution costs or optimized privacy. For such use-cases, Trace-Net provides an automatable verification framework for contract implementations. This would enable the enforcement of universal contract safety policies at run-time, for example.

References

[1] Joseph Poon and Thaddeus Dryja. The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments, 2016. https://lightning.network/lightning-network-paper.pdf.

[2] Lukas Aumayr, Oguzhan Ersoy, Andreas Erwig, Sebastian Faust, Kristina Hostakova, Matteo Maffei, Pedro Moreno-Sanchez, and Siavash Riahi. Generalized Bitcoin-Compatible Channels. Cryptology ePrint Archive, Report 2020/476, 2020. https://ia.cr/2020/476.

[3] Gregory Maxwell. CoinJoin: Bitcoin privacy for the real world. Post on Bitcoin Forum., August 2013. https://bitcointalk.org/index.php?topic=279249.

[4] James Chiang. Bitcoin Trace-Net: Formal Contract Verification at Signing Time. In *MIT Cryptoeconomic Systems Conference 2020*, March 2020. https://arxiv.org/abs/2007.07528.

[5] Danny Dolev and Andrew Yao. On the security of public key protocols. *IEEE Transactions on information theory*, 29(2):198–208, 1983. https://doi.org/10.1109/TIT.1983.1056650.

[6] Rick Klomp and Andrea Bracciali. On symbolic verification of Bitcoin's script language. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pages 38–56. Springer, 2018. https://doi.org/10.1007/978-3-030-00305-0_3.

[7] Pieter Wuille. Miniscript. http://bitcoin.sipa.be/miniscript. Accessed: 2020-05-15.

[8] Pieter Wuille and Andrew Poelstra. Miniscript: Streamlined Bitcoin Scripting. https://medium.com/blockstream/miniscript-bitcoin-scripting-3aeff3853620, September 2019.

[9] Louchka Popova-Zeugmann. *Time and Petri Nets*. Springer Verlag, 2013. https://doi.org/10.1007/978-3-642-41115-1_3.

[10] Timelock - Bitcoin Wiki. https://en.bitcoin.it/wiki/Timelock. Accessed: 2020-05-15.

[11] Massimo Bartoletti and Roberto Zunino. BitML: A Calculus for Bitcoin Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 83–100. ACM, 2018. https://doi.org/10.1145/3243734.3243795.

[12] Nicola Atzei, Massimo Bartoletti, Stefano Lande, Nobuko Yoshida, and Roberto Zunino. Developing secure Bitcoin contracts with BitML. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1124–1128, 2019. https://doi.org/10.1145/3338906.3341173.

[13] Massimo Bartoletti, Maurizio Murgia, and Roberto Zunino. Renegotiation and recursion in Bitcoin contracts, 2020. https://arxiv.org/abs/2003.00296.

[14] Massimo Bartoletti and Roberto Zunino. Verifying liquidity of Bitcoin contracts. In *Principles of Security and Trust*, pages 222–247. Springer International Publishing, 2019. https://doi.org/10.1007/978-3-030-17138-4_10.

High-assurance field inversion for pairing-friendly primes

Benjamin S. Hvass, Diego F. Aranha, Bas Spitters

Concordium Blockchain Research Center, Computer Science Aarhus University

Bilinear pairings have become popular for deploying privacy-preserving cryptocurrencies, as they represent a fundamental building block for the zero-knowledge proofs required for security. ZCash is a clear example of this trend, where pairing-based zero-knowledge Succint Non-Interactive Arguments of Knowledge (zk-SNARKs) underlie private shielded transactions [BCG+14]. Another example of application of pairing-based protocols comes from the Chia blockchain, where Boneh-Lynn-Shacham (BLS) [BLS04] signatures where adopted for improved smart transaction support.

There has been substantial progress in the past decade towards selecting parameters [BN05, AFK+12, BD17] and implementing pairing-based cryptography efficiently in software [ABLR13]. However, current record-setting implementations rely on hand-optimized architecture-specific Assembly code for the underlying field arithmetic and a great deal of manual tuning to unlock the best performance across a range of architectures. This introduces low-level code which is both hard to audit and to verify as correct, and a number of cryptographic libraries have suffered with simple bugs as a direct consequence [EPG+19]. Moreover, implementations need to be *constant-time*, in the sense that execution time does not depend on input. Otherwise, potentially confidential information may be leaked and compromise the security and privacy preserving properties of the code.

Due to its many optimizations efficient code can be hard to verify in a *post hoc* way. Recently, an alternative path for implementing cryptographic libraries was demonstrated as viable in the Fiat-Crypto framework [EPG+19]. By combining correct-by-design optimized low-level code with automatically generated and formally verified high-level code, it became possible to develop libraries which are both efficient and formally verified. The approach was illustrated through the implementation of field arithmetic for several standardized elliptic curves using an extensible code generation framework, capable of producing code competitive in performance with popular hand-optimized multi-precision libraries [EPG+19]. The verification steps are conducted using the Coq proof assistant, a state-of-the-art theorem prover [The19]. Such high assurance cryptographic implementations have recently been adopted by the industry: Google's BoringSSL and the WireGuard VPN relies on Fiat-Crypto [EPG+19]. Firefox and the WireGuard VPN depend on Evercrypt [PPF+19]. A SHA-3 implementation was made using Jasmin [ABRB+19].

Contributions We implement a verified and improved version of the constant-time algorithm from [BY19] in the Coq proof assistant and use Fiat-Crypto to generate an efficient C-implementation.

The inversion algorithm The inversion algorithm [BY19] we have implemented is a constant-time variant of the Euclidean Algorithm. It consists of a constant amount of iterations of a so-called *division step*. Each of these division steps consists of a conditional swap and a few arithmetical operations including a shift, a negation and an addition.

In [BY19], two main variants are presented: One which always operates on full-precision integers (i.e., of the same size as the prime modulus) and one which exploits that the conditional swaps in each division step only depends on the lower bits of the inputs and only does arithmetic on word-sized integers. The latter is a lot more efficient because it avoids the expensive large-number arithmetic.

We have verified all methods required for both the full- and variable precision version (i.e., the division step).

Verified code and the Coq proof assistant Fiat Cryptography is implemented and verified in the Coq proof assistant [The19]. Coq has a small trusted code base consisting of a kernel which checks all the proofs about programs written in Coq's functional language (Gallina). The kernel has been checked by logicians, the implementation is being computer verified [SBF+19] and is used by thousands of developers. Moreover, it is the only proof assistant that has been used at the highest EAL7 level of common criteria. One can manually inspect the *specification* of an algorithm and trust the Coq kernel to have checked that the proof that an algorithm satisfies it.

Fiat-Crypto embeds a small C-like language in the general logical framework of Coq. This is the language in which we implement the algorithm. Fiat then makes a provable connection between mathematical definitions and the generated efficient code in C or rust.

E.g., consider the following Coq definition of the divstep method in [BY19]:

```
Definition divstep_spec_full m d f g v r :=
  if (0 <? d) && Z.odd g
  then (1 - d, g, (g - f) / 2,
        2 * r mod m, (r - v) mod m)
  else (1 + d, f, (g + (g mod 2) * f) / 2,
        2 * v mod m, (r + (g mod 2) * v) mod m).
```

The specification of our Fiat-Crypto implementation of divstep is then that it should behave in the same way, i.e., compute the same result modulo the different representations as infinite precision and finite precision integers. This is asserted in the theorems `divstep_correct_full` and `twos_complement_word_full_divstep_correct` for respectively multi-limb and wordsize integers.

These theorems reside in `src/Arithmetic/Inv.v` and `src/Arithmetic/JumpDivstep.v` in the source code.

Implementation We implement the full- and variable precision versions of the constant-time algorithm from [BY19] in Fiat and use the framework to generate verified and constant-time field inversion. To do this, we extended the library with a few methods including shifting of large integers and signed arithmetic for large integers (in twos complement).

The implementation allows for inversion modulo any prime, in particular for primes used in pairing-based cryptography. We illustrate our approach with the BLS12-381 curve used in ZCash as an efficient instantiation for pairings at the 128-bit security level[1].

We have proved functional correctness of all necessary subprocedures of the inversion algorithm from [BY19] (including the divstep method). In future work, we will prove that when iterated this division step actually computes the field inverse. This, however, requires reasoning about real and 2-adic numbers, which needs several additional libraries. The loop which iterates the division step is also not generated by Fiat, since this is not yet supported; at the moment one has to write the loop around the generated C-code oneself. We are working on automating this.

[1]https://electriccoin.co/blog/new-snark-curve/

The code is available at `https://github.com/bshvass/fiat-crypto` (our contribution is approx. 1500 lines of code).

Benchmarks The generated code was integrated in the RELIC toolkit [AG], a cryptographic library containing a state-of-the-art implementation of pairings. RELIC uses a combination of hand-written Assembly with higher-level C-code, and has been used by cryptocurrency and blockchain projects. For example, it has been adopted by the Chia project for BLS signatures and used for generating test vectors for the ZCash implementation of pairings in Rust.

Integrating the code with RELIC allowed convenient benchmarking to compare the efficiency of our approach with other field inversion algorithms already implemented in the library. We compare our work to the following implementations (all using the finite field arithmetic generated by Fiat Crypto): EEA, an implementation of the Extended Euclidean Algorithm; FLT, an implementation which uses modular exponentiation by $(p-2)$ using arithmetic generated by Fiat; and a hand-optimized implementation of the algorithm from [BY19] which is not auto-generated by Fiat. The results are summarized in the following table (where the second and last entries corresponds to this work):

Algorithm	Verified	Auto generated	Leaks	Cycles
Bernstein-Yang (fast) [BY19]	No	No	length of p	32,384
Bernstein-Yang (fast) [BY19]	**Yes**	**Yes**	**length of** p	**87,733**
Extended Euclidean	No	No	p and input	157,870
Fermat's Little Theorem	No	Partially	p	296,302
Bernstein-Yang [BY19]	No	Partially	length of p	305,924
Bernstein-Yang [BY19]	**Yes**	**Yes**	**length of** p	**309,150**

Table 1: Cycle counts for field inversion measured on an Intel Core i7-8650U CPU running at 1.90GHz with HyperThreading and TurboBoost disabled.

In the tests, p was always chosen to be the BLS12-381 prime, but Fiat can generate the algorithm for any desired prime. Thus, our verified and constant-time auto-generated implementation is approximately only half as fast as an insecure competitive implementation. This is reasonable for uses where security and correctness are indispensable.

Future work At the moment only the divstep method is proven correct and we are working on proving the entire algorithm correct (i.e., that iterating the division step yields the field inverse). This has proven more difficult than anticipated, due to the proof requiring

To fully generate the implementations, one would need to extend Fiat with loops. Furthermore, one could extend the supported language of Fiat to be able to use more efficient C primitives which would speed up all implementations generated by Fiat.

To obtain a fully verified, and constant time, compilation to Assembly, we would like to use the CompCert [BBG+20] verified C-compiler, but at the moment this does not support some GCC extensions which Fiat-Crypto relies on (128-bit integer types in particular).

References

[ABLR13] Diego F. Aranha, Paulo S. L. M. Barreto, Patrick Longa, and Jefferson E. Ricardini. The Realm of the Pairings. In *Selected Areas in Cryptography*, volume 8282 of *LNCS*, pages 3–25. Springer, 2013.

[ABRB$^+$19] José Bacelar Almeida, Cécile Baritel-Ruet, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Alley Stoughton, and Pierre-Yves Strub. Machine-checked proofs for cryptographic standards: Indifferentiability of sponge and secure high-assurance implementations of SHA-3. In *CCS*, pages 1607–1622, 2019.

[AFK$^+$12] Diego F. Aranha, Laura Fuentes-Castañeda, Edward Knapp, Alfred Menezes, and Francisco Rodríguez-Henríquez. Implementing Pairings at the 192-Bit Security Level. In *Pairing*, volume 7708 of *LNCS*, pages 177–195. Springer, 2012.

[AG] D. F. Aranha and C. P. L. Gouvêa. RELIC is an Efficient LIbrary for Cryptography. `https://github.com/relic-toolkit/relic`.

[BBG$^+$20] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. Formal verification of a constant-time preserving C compiler. *PACMPL*, 4(POPL):7:1–7:30, 2020. doi:10.1145/3371075.

[BCG$^+$14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *S&P'14*, pages 459–474. IEEE Computer Society, 2014.

[BD17] Razvan Barbulescu and Sylvain Duquesne. Updating key size estimations for pairings. *IACR Cryptology ePrint Archive*, 2017:334, 2017.

[BLS04] Dan Boneh, Ben Lynn, and Hovav Shacham. Short Signatures from the Weil Pairing. *J. Cryptology*, 17(4):297–319, 2004.

[BN05] Paulo S. L. M. Barreto and Michael Naehrig. Pairing-Friendly Elliptic Curves of Prime Order. In *Selected Areas in Cryptography*, volume 3897 of *LNCS*, pages 319–331. Springer, 2005.

[BY19] Daniel J. Bernstein and Bo-Yin Yang. Fast constant-time gcd computation and modular inversion. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(3):340–398, May 2019. doi:10.13154/tches.v2019.i3.340-398.

[EPG$^+$19] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple High-Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises. In *S&P'19*. IEEE Computer Society, 2019. `https://github.com/mit-plv/fiat-crypto`.

[PPF$^+$19] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, et al. Evercrypt: A fast, verified, cross-platform cryptographic provider. In *S&P'19*, pages 634–653, 2019.

[SBF$^+$19] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. Coq Coq Correct! Verification of Type Checking and Erasure for Coq, in Coq. *POPL*, 4, 2019. doi:10.1145/3371076.

[The19] The Coq Development Team. The coq proof assistant, version 8.10.0, October 2019. doi:10.5281/zenodo.3476303.

Albert, an intermediate smart-contract language for the Tezos blockchain

Bruno Bernardo, Raphaël Cauderlier, Arvid Jakobsson, Basile Pesin, and Julien Tesson

Nomadic Labs, Paris, France
{first_name.last_name}@nomadic-labs.com

Abstract

Tezos is a smart-contract blockchain. Tezos smart contracts are written in a low-level stack-based language called Michelson. In this article we present Albert, an intermediate language for Tezos smart contracts which abstracts Michelson stacks as linearly typed records. We also describe its compiler to Michelson, written in Coq, that targets Mi-Cho-Coq, a formal specification of Michelson implemented in Coq.

Introduction Tezos is a public blockchain launched in June 2018 with smart-contracts capabilities. An open-source implementation of a Tezos node in OCaml is available [10].

The language of the smart contracts stored in the Tezos blockchain is Michelson [1]. It is a stack-based Turing-complete domain-specific language with a mix of low-level and high-level features. Low-level features include stack manipulation instructions. High-level features are high-level data types (option types, sum types, product types, lists, sets, maps, and anonymous functions) as well as corresponding instructions. Michelson is strongly typed: data, stacks and instructions have a type. Intuitively the type of a stack is a list of the types of its values, and the type of an instruction is a function type from the input stack to the output stack. The combination of high and low level features is the result of a trade-off between the need to meter resource consumption (computation *gas* and storage costs) and the willingness to have strong guarantees on the Michelson programs.

Michelson has been designed with formal verification in mind: its strong type system guarantees that there can be no runtime error apart from explicit failure and *gas* or token exhaustion. Its OCaml implementation uses GADTs to ensure subject reduction. Furthermore, there is a Coq implementation of a Michelson interpreter, called Mi-Cho-Coq [5] and the functional correctness of some contracts have been formally verified using this framework [8, 7].

Because of its low-level aspects, it is hard to write Michelson programs, and indeed higher-level languages compiling to Michelson, such as LIGO [3] or SmartPy [4] have been developed in the Tezos ecosystem. Ideally, there would be certified compilers from these high-level languages to Michelson, and formal proofs of smart-contracts would be done directly at the higher level and not at the Michelson level, as it is being done now with Mi-Cho-Coq.

In this context, the goal of Albert is to be an intermediate language with a certified compiler to Michelson that could be used as a target for certified compilers from high-level languages.

Design overview The key aspect of Albert's design is the abstraction of Michelson stacks by records with named fields. This gives two practical benefits: unlike in Michelson, in Albert we do not need to care about the order of the values and we can bind variables to names. Also, unlike Michelson where contracts can only contain one sequence of instructions, it is possible in Albert to define multiple functions, thus giving the possibility to implement libraries. An important limitation of Albert is that resources are still being tracked: variables are typed by a linear type system that enforces that each value cannot be consumed twice. A **dup** operation duplicates resources that need to be consumed multiple times. A next step would be to generate these operations in order to abstract data consumption.

A more detailed presentation of Albert's syntax, typing rules and semantics can be found in [6]. In this extended paper, we restrict ourselves to a global overview of Albert's design and features.

In a nutshell, each expression or instruction is typed by a pair of record types whose labels are the variables touched by the instruction or expression. The first record type describes the consumed values and the second record type describes the produced values. Thanks to the unification of variable names and record labels, records in Albert generalise both the Michelson stack types and the Michelson pair type.

Albert offers slightly higher-level types than Michelson: records generalise Michelson's pairs and non-recursive variants generalise Michelson's binary sum types as well as booleans and option types. Variants offer two main operations to the user: constructing a variant value using a constructor, and pattern-matching on a variant value.

The semantics of the Albert base language is defined in big-step style.

An example of a simple voting contract written in Albert is available here [2]. The user of the contract can only vote for a pre-defined set of options and must pay at least a certain amount of tokens for its vote to be considered. The storage of the contract is a record with two fields: a `threshold` that represents the minimum amout that must be transferred, and an associative map, `votes`, with strings as keys (the options of the vote) and integers as values (the number of votes for each associated key). The contract contains a `vote` function that checks that the parameter sent is one of the available options, fails if not and otherwise updates the vote count. The main function `guarded_vote` verifies that the amount of tokens sent is high enough and if so, calls `vote`.

Implementation overview Albert is formally specified with the Ott tool [9] in a modular way (one `.ott` file per fragment of the language). From the Ott specification the Albert lexer and parser as well as typing and semantic rules are generated in Coq. The type checker is a Coq function that uses an error monad to deal with ill-typed programs. There is no type inference, which should not be a problem since Albert is supposed to be used as a compilation target.

The Albert compiler is written in Coq, as a function from the generated Albert grammar to the Michelson syntax defined in Mi-Cho-Coq. The compiler is extracted to OCaml code, which is more efficient and easier to use as a library. Compilation of types, data and instructions are mostly straightforward, apart from things related to records or variants. Records are translated into nested pairs of values, variants into a nesting of sum types. Projections of record fields are translated into a sequence of projections over the relevant components of a pair. Pattern matching over variants are translated into a nesting of `IF_LEFT` branchings. A mapping from variable names to their positions in the stack exists at every point in the program. This mapping is currently naive, variables are ordered by the lexicographic order of their names. This mapping is used in the translation of assignment instructions.

Future Work Albert is very much a work in progress. Next steps would be to have a smarter implementation of the compiler that would produce optimised code, as well as to prove the compiler correctness and meta-properties of the Albert language. Longer term, we would like to implement a certified decompiler from Michelson to Albert as well as a weakest-precondition calculus to Albert in order to reason about Albert programs.

References

[1] Michelson: the language of Smart Contracts in Tezos. https://tezos.gitlab.io/whitedoc/michelson.html.

[2] Albert. Implementation of a voting contract. https://gitlab.com/nomadic-labs/albert/-/blob/FMBC2020/examples/vote_wth_fun.alb. Accessed: 2020-05-22.

[3] Gabriel Alfour. LIGO: a friendly smart-contract language for Tezos. https://ligolang.org. Accessed: 2020-05-12.

[4] Smart Chain Arena. SmartPy. https://smartpy.io. Accessed: 2019-12-12.

[5] Bruno Bernardo, Raphaël Cauderlier, Zhenlei Hu, Basile Pesin, and Julien Tesson. Mi-Cho-Coq, a framework for certifying Tezos Smart Contracts. In *Proceedings of the First Workshop on Formal Methods for Blockchains (to be published)*, FMBC 2019, 2019.

[6] Bruno Bernardo, Raphaël Cauderlier, Basile Pesin, and Julien Tesson. Albert, an intermediate smart-contract language for the Tezos blockchain. In *Proceedings of the 4^{th} Workshop on Trusted Smart Contracts (to be published)*, 2020.

[7] Arthur Breitman. Multisig contract in Michelson. https://github.com/murbard/smart-contracts/blob/master/multisig/michelson/generic_multisig.tz.

[8] Raphaël Cauderlier. Certification of Breitman's Multisig implementation in Mi-Cho-Coq. https://gitlab.com/nomadic-labs/mi-cho-coq/blob/master/src/contracts_coq/multisig.v.

[9] Peter Sewell, Francesco Zappa Nardelli, and Scott Owens. Ott. https://github.com/ott-lang/ott.

[10] Tezos. An open-source implementation in OCaml. https://gitlab.com/tezos/tezos/. Accessed: 2020-05-12.

WhylSon: Proving your Michelson Smart Contracts in Why3*

Luís Pedro Arrojado da Horta[1,2], João Santos Reis[1,2], Mário Pereira[2,4], and Simão Melo de Sousa[1,2,3]

[1] Release Lab., Universidade da Beira Interior,Portugal
[2] NOVA LINCS
[3] Cloud Computing Competence Center (C4)
[4] DI, FCT, Universidade Nova de Lisboa, Portugal
{luis.horta | joao.reis | desousa}@ubi.pt, mjp.pereira@fct.unl.pt

Abstract

This paper introduces WhylSon, a deductive verification tool for smart contracts written in Michelson, which is the low-level language of the Tezos blockchain. WhylSon accepts a formally specified Michelson contract and automatically translates it to an equivalent program written in WhyML, the programming and specification language of the Why3 framework. Smart contract instructions are mapped into a corresponding WhyML shallow-embedding of the their axiomatic semantics, which we also developed in the context of this work. One major advantage of this approach is that it allows an out-of-the-box integration with the Why3 framework, namely its VCGen and the backend support for several automated theorem provers. We also discuss the use of WhylSon to automatically prove the correctness of diverse annotated smart contracts.

1 Introduction

Smart contracts are reactive programs that perform general-purpose computations within a blockchain and have been used to encode arbitrarily complex business logic of digital transactions. Since the use of smart contracts has been increasing significantly, and also since smart contracts cannot be changed once uploaded into a blockchain, it is of paramount importance to tackle the challenge of formally verifying their safety and correctness. The main focus of our work is the formal verification of smart contracts for the Tezos blockchain [15]. Moreover, we will lean towards the Michelson language and its formal specification [24].

Our approach is to make the verification process as automatic as possible. In order to do that, we chose the deductive program verification platform Why3 [14] as the underlying proof framework tool in our smart contract verification tool. Why3 is a framework aimed at automatic theorem proving through the use of external provers such as Alt-ergo [8], Z3 [12] or CVC4 [2]. Additionally, when a proof obligation can not be automatically discharged, Why3 allows the user to call interactive theorem provers such as Coq or Isabelle.

This document is organised as follows: Section 2 discusses how we specified Michelson language in the Why3 platform. Our axiomatic semantic will be described in Section 3. Section 4 explains how we generate WhyML code from Michelson. On Section 5 we will details two case studies, and Section 6 contains a critical analysis over the work developed. Finally, Sections 7 and 8 discuss some of the related work in the field of formal verification of smart contracts and the main conclusions we gathered throughout the development of this work.

*Research Supported by the Tezos Foundation through the project FRESCO - FoRmal vErification of Smart COntracts

2 Michelson Specification in Why3

Michelson is a stack rewriting language for writing smart contracts for the Tezos blockchain. For a complete explanations of the Michelson language, we refer the reader to [24]. The relevant details of this language for this work will be introduced when needed.

In the Michelson language there are four primitive data types for constants, that we can name: `nat`, `int`, `string` and `bytes`. Additionally we have type `bool` for booleans and the optional type `Option` τ of type τ, (similar to the option type in OCaml). Some of these types are not primitive in Why3 and thus, we had to make some choices on how to represent them. In order to ease the correspondence between types in each language, we present the reader with Table 1.

Michelson Primitive Type	**Corresponding Why3 type (v 1.3.0)**
string	string
nat	nat
int	int
bytes	seq bv.BV8
bool	bool
option τ	option τ
unit	unit

Table 1: Correspondence between Michelson and Why3 types.

In Michelson, both `int` (for integer constants) and `nat` (for natural number constants) have arbitrary precision, which means that computations with such constants are only limited by the Gas one is willing to pay. When it comes to Why3, type `int` already has arbitrary precision, but we had to manually define type `nat` as shown in Figure 1. Namely, we model `nat` as a record type with a single field `value` add an invariant.

```
type nat = { value: int }
  invariant { value ≥0 }
```

Figure 1: Definition of type `nat` in Why3.

Why3 supports type `string` as built-in since Version 1.3.0. Given that a byte is a set of 8 bits, we chose to use BV8 (short for BitVector of size 8). In Michelson all data structures are immutable, and that property is still maintained with the corresponding types in Why3.

In Michelson, comparisons between constants of the same type are possible. Figure 2 shows the definition of those comparable types in Why3.

Type `Mutez` represents *micro-tez* which is in fact the smallest unit of the Tezos blockchain token. Every operation involving Mutez is mandatory checked for over/underflows. Moreover this is one of the cases where the type system really helps, because it can assure us that we do not confuse *Mutez* for another numerical constant. The `Key_hash` type represents the hash value of a public key. Additionally, type `Timestamp` represents a date that can be written in a readable format according to RFC3339 [20], or in an optimised format, being the number of seconds since *Epoch*.

According to the specification in [24], comparison functions in Michelson for two given

```
type comparable =
  | Int int
  | Nat Natural.nat
  | String string
  | Bytes (seq Bytes.t)
  | Mutez int
  | Bool bool
  | Key_hash string
  | Timestamp string
  | Address string
```

Figure 2: Definition of type `comparable` in Why3.

constants K_1 and K_2 must return a integer value as shown in equation 1.

$$compare\ K_1\ K_2 = \begin{cases} -1 & \text{if } K_1 < K_2 \\ 0 & \text{if } K_1 = K_2 \\ 1 & \text{if } K_1 > K_2 \end{cases} \tag{1}$$

In order to abide by the given specification we had to implement our own versions for said functions. As an example, we present the reader with our implementation of the comparison function for boolean constants (see Figure 3).

```
let compare_bool (a b: bool) : int =
  match a, b with
  | False,True → (-1)
  | True,False → 1
  | _,_ → 0
  end
```

Figure 3: Comparison function for type `bool`.

Michelson's execution stack contains only data or instructions, thus the type `data` is defined as depicted in Figure 4.

In order to ensure that all data is properly constructed, the predicate `well_formed_data` is defined as shown in Figure 5.

For the WhyML representation of the Michelson execution stack, we chose an immutable sequence (type `stack_t`) defined as follows:
`type stack_t = seq well_formed_data`.

Additionally we defined a function named `typ_infer` for determining the type of a specific element in the stack. This function gives us an extra assurance that the stack is well formed and well typed.

3 Axiomatic Semantics in Why3

In this section we present the reader with some of the more important details of our axiomatic semantics of Michelson in WhyML. Our approach is a shallow embedding of the Michelson language in WhyML. Furthermore opcodes such as `SEQ` do not need to be directly encoded given that one can take advantage of the WhyML language constructs e.g. `let ... in ...` or the sequence operator ';'.

Every Michelson opcode results in an abstract function in WhyML containing a set of annotations (i.e. rules). Moreover this set of rules defines the expected behaviour of that opcode

```
type data =
  | Comparable comparable
  | Key
  | Unit
  | Some_data data
  | None_data typ
  | List (list data) typ
  | Pair data data
  | Left data typ
  | Right data typ
  | Set SetApp.set comparable_t
  | Map (my_map data) comparable_t typ
  | Big_map (my_map data) comparable_t typ
  ...
  | Mutez_Const
  | Chain_ID_Const
  | PACK_Const
  | Create_Contract_OP
  | Transfer_Tokens_OP
  | Set_Delegate_OP
  | Create_Account_OP
 with instruction = (* For convenience, all CAPITAL types are Michelson native instructions *)
  | SEQ_I instruction instruction
  ...
```

Figure 4: Definition of type `data` in WhyML.

```
predicate well_formed_data (d: data) =
    match d with
        | Map m _ _
        | Big_map m _ _ → well_formed_map m
        | Left d _
        | Right d _
        | Some_data d → well_formed_data d
        | Pair d1 d2 → well_formed_data d1 ∧ well_formed_data d2
        | List lst t → well_formed_data_list lst t
        | _ → true
    end
with well_formed_data_list (l: list data) (t: typ) =
    match l with
        | Nil → true
        | Cons hd tl → well_formed_data hd ∧ well_formed_data_list tl t ∧ typ_infer hd = t
    end
```

Figure 5: Definition of predicate `well_formed_data` in WhyML.

and the effect it produces on the stack. All the opcodes take as input (at least) the stack and return a new stack.

As an example of such abstract function take the opcode `ADD` defined in [24] as the sum of the top two elements in the input stack, figure 6 depicts the corresponding WhyML code.

Lines 2-9 in figure 6 define the pre conditions and lines 10-32 define the post conditions for this particular instruction. In particular, lines 4-6 are related with the contents of the stack whereas lines 7 and 8 concern the type of elements in the stack.

The limit of our formalisation. In the present version of the axiomatic semantics, we have not formalised the internal details of the cryptographic operations. We have instead defined these instructions as abstract operations that follow the expected pre and post conditions. For instance, the definition of the `sha512` instruction is shown on figure 7.

Because the semantics of serialisation operations is not clear from the reference documen-

```
val add (s: stack_t) (fuel: int) : stack_t
    requires { fuel > 0 }
    requires { length s ≥ 2 }
    requires { match typ_infer s[0].d, typ_infer s[1].d with
                | Comparable_t Int_t, Comparable_t Int_t
                | Comparable_t Int_t, Comparable_t Nat_t
                | Comparable_t Nat_t, Comparable_t Int_t
                | Comparable_t Nat_t, Comparable_t Nat_t → true
                | _ → false end }
    ensures  { length result = length s - 1 }
    ensures { match typ_infer s[0].d, typ_infer s[1].d with
                | Comparable_t Int_t, Comparable_t Int_t
                | Comparable_t Int_t, Comparable_t Nat_t
                | Comparable_t Nat_t, Comparable_t Int_t → typ_infer result[0].d = Comparable_t Int_t
                | Comparable_t Nat_t, Comparable_t Nat_t → typ_infer result[0].d = Comparable_t Nat_t
                | _ → false end }
    ensures  { forall i: int. 1 ≤ i < length result → result[i] = s[i+1] }
    ensures  { forall i: int. 1 ≤ i < length result → typ_infer result[i].d = typ_infer s[i+1].d }
    ensures  { match s[0].d, s[1].d with
                | Comparable (Int x), Comparable (Int y) →
                  let res = x + y in
                  result = (mk_wf_data res) :: s[2 ..]
                | Comparable (Int x), Comparable (Nat y) →
                  let res = Comparable (Int (x + (eval_nat y))) in
                  result = (mk_wf_data res) :: s[2 ..]
                | Comparable (Nat x), Comparable (Int y) →
                  let res = Comparable (Int ((eval_nat x) + y)) in
                  result = (mk_wf_data res) :: s[2 ..]
                | Comparable (Nat x), Comparable (Nat y) →
                  let res = Comparable (Nat (add_nat x y)) in
                  result = (mk_wf_data res) :: s[2 ..]
                | _ → false end }
```

Figure 6: Definition of ADD in WhyML.

tation, we also choose to abstract these operation the same way we handle cryptographic operations.

```
 val sha512_op (s: stack_t) (fuel: int) : stack_t
    requires { fuel > 0 }
    requires { length s ≥ 1 }
    requires { typ_infer s[0].d = Comparable_t Bytes_t }
    ensures  { length result = length s }
    ensures  { forall i: int. 1 ≤ i < length result → result[i] = s[i] }
    ensures  { typ_infer result[0].d = Comparable_t Bytes_t }
    ensures  { forall i: int. 1 ≤ i < length result → typ_infer result[i].d = typ_infer s[i].d }
    ensures  { result = (mk_wf_data Crypto_Hash_Const) :: s[1..] }
```

Figure 7: Definition of sha512 in WhyML.

4 Automated Translation

In this section we present some of the most important details about the automatic translation from Michelson to WhyML. For a visual representation of the WhylSon plugin structure, we refer the reader to figure 8. In order to obtain an abstract-syntax tree of a Michelson smart contract we implemented a parser in OCaml and Menhir. This parser respects the syntax described on the Tezos documentation [24]. It allows us to obtain a data type that fully abstracts the syntax (with the exception of annotations) which we can then manipulate in order to generate WhyML. The automated translation to WhyML using the Why3 API is explained in subsection 4.1. Additionally, a small example of a translated Michelson contract will be given in subsection 4.2.

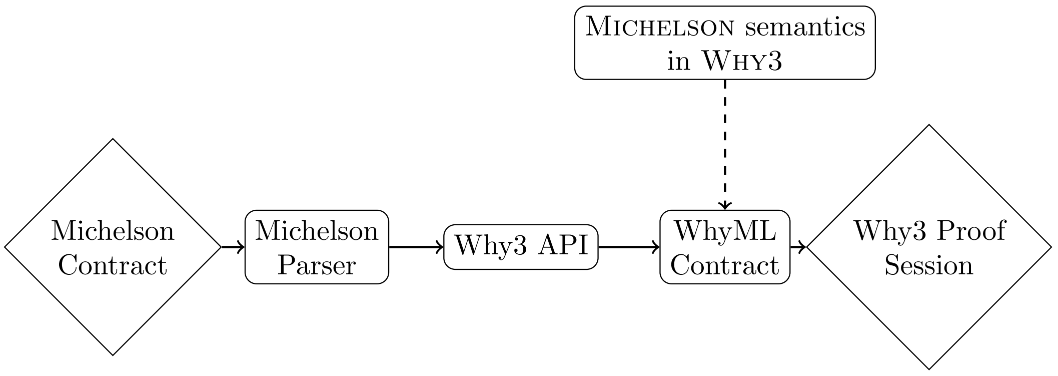


Figure 8: Visual Structure of the Implementation.

4.1 Why3 API

The core of our development is the translation of a Michelson contract into an equivalent WhyML program. Our purpose is to be able to feed the generated program to the Why3 proof engine, in order to conduct formal verification on the original contract[1]. It is worth highlighting that our translation is completely done in-memory, *i.e.*, Why3 *reads* the Michelson file and no intermediate Why3 file is generated in order to contain the result of translation. This leads to a very smooth integration with the Why3 framework.

The key insight of our translation mechanism is that we take the AST representation issued by the Michelson parser and, using the Why3 source code as an OCaml library, we generate an AST of the WhyML language. We organise our translation code into several mutually-recursive functions, each one dealing with the translation of a different syntactic element of the Michelson language. Consider, for instance, the Michelson instruction `ADD`. For this instruction, our parser emits an AST containing the node `I_add`. To translate this `add` statement into this WhyML counterpart, we build the following homomorphic translation

```
let rec inst = function
  | I_add -> mk_expr (Eidapp (Qident (mk_id "add"), stack_fuel_args))
  ...
```

where `mk_expr` and `mk_id` are simply smart constructs for WhyML expressions and identifiers, respectively. The above OCaml code creates an application expression to our axiomatized `add` operation of Figure 6, where the arguments are the current stack and fuel amount. A more interesting example, and one that shows how we take advantage of underlying translation to

[1]The correctness of the generated WhyML program implies the correctness of the original Michelson contract. At the moment, such an argument is based on the informal reasoning that the semantics of Michelson is captured by the our axiomatic semantics developed in Why3. A more rigorous rationale, which we plan to develop as future work, must provide mathematical and/or formal evidence that Michelson operational semantics conforms to our axiomatic encoding

WhyML, is the Michelson `SEQ` operation. For such Michelson statement, our parser issues a node of the form `I_seq (i1, i2)`, where `i1` and `i2` are the two instructions composing the sequence. Our translation engines features the following code for this case:

```
| I_seq (i1, i2) ->
    mk_expr (Elet (mk_id "__stack__", false, Expr.RKnone, inst i1, inst i2))
```

which builds the expression `let __stack__ = i1 in i2`. The Boolean constant `false` above tells Why3 that this is a non-ghost expression, while the `Expr.RKnone` indicates that this is a simple locally-defined symbol, with no direct translation to a purely-logical symbol. Let us note that the tree-like data type produced by our translation corresponds to the AST issued by the WhyML parser, hence no typing or name resolution information is present at this point.

Having defined our Michelson to WhyML transformation function, we want to integrate it with the Why3 framework in a completely transparent fashion for the end user. This means that we want to use the Why3 proof engine over a Michelson contract, as if this was the native language of the framework. One can easily extend the Why3 framework with the support for new input languages via its plugin capabilities. This is as simple as providing a parser and a translation function from the source language into one of the WhyML internal AST. Finally, in order to register the newly-developed plugin into the Why3 configuration base, one simply states the extension of files that should be processed by the devised translation. In our particular case, we write the following:

```
let () =
  Env.register_format mlw_language "michelson" ["tz"] read_channel
    ~desc:"Michelson format"
```

Here, `mlw_language` indicates that the target of our translation is a WhyML program and `read_channel` is the function that calls the Michelson parser and feeds the produced AST to our transformation mechanism. With all this machinery in place, one can then call Why3 directly on a `.tz` file. For instance, if one wishes to formally verify the contract contained in file `foo.tz`, using our plugin, the command line would be

```
$ why3 ide foo.tz
```

which opens the Why3 graphical Integrated Development Environment over the result of the Michelson contract translation.

4.2 A Trivial Example

For a better understanding of this automated translation, we present the reader with a visual toy example. Consider the Michelson contract shown in figure 9.

```
parameter nat;
storage nat;
code { UNPAIR; ADD;
       NIL operation; PAIR };
```

Figure 9: Toy example of a Michelson contract.

This is a very simple contract, in fact it takes the `nat` it received as parameter and adds it to the `nat` in the storage. Basically it just adds two natural numbers. The Michelson contract does not contain any pre or post conditions, but WhylSon is able to directly infer four safety conditions, namely about the length and type of both the input and the output stacks. The WhyML code generated by WhylSon is shown in figure 10.

```
use axiomatic.AxiomaticSem
use dataTypes.DataTypes
use seq.Seq
use int.Int
let test (__stack__: stack_t) (__fuel__: int) : stack_t
  requires { (length __stack__) = 1 }
  requires { __fuel__ > 0 }
  requires { (typ_infer (d (__stack__[0])))
             = (Pair_t (Comparable_t Nat_t ) (Comparable_t Nat_t )) }
  ensures { (length result) = 1 }
  ensures { (typ_infer (d (result[0])))
            = (Pair_t (List_t Operation_t ) (Comparable_t Nat_t )) } =
  let __stack__ =
    let __stack__ = unpair __stack__ __fuel__ in
    (let __stack__ = add __stack__ __fuel__ in
     (let __stack__ = nil_op __stack__ __fuel__ Operation_t  in
      (pair __stack__ __fuel__))) in
  __stack__
```

Figure 10: The WhyML generated code for the toy example.

Using only the split_vc transformation this example generates 26 verification conditions that are quickly proven by Alt-ergo[8].

5 Case Studies

In this section we discuss two case studies, namely the multisig and factorial smart contracts and explain how safety and functional correctness can be proved within WhylSon. For the sake of brevity we will elaborate the proof of safety for the multisig smart contract and on the functional correctness of the factorial smart contracts. We will detail the proof of correctness of the factorial smart contract since this contract highlight particularly well our purpose to show the advantages but also the drawbacks of our approach.

Both of these contracts were manually translated to WhyML. The complete details of the formalisation and the proof of these smart contracts can be found at https://gitlab.com/releaselab/fresco/whylson

5.1 Multisig

There are several versions of the multisig contract, and the one we used can be found in [25]. We separated the multisig contract into three parts. The first one is the majority of the contract, the second one is the loop which iterates over the list of keys and optional signatures (iter_multisig) and finally, the third part (outer_if_left) is where the operation requested by the signers is produced. For the sake of brevity, these fuctions are not depicted here.

Figure 11 contains the part of the multisig function that represents the contract. We ask the reader to notice that it has only two pre conditions and two post conditions regarding the size and the type of the stack. In order to prove the last post condition, we had to equip some of the code in the contract with some additional typing information. An example of such typing information is the one below the iter_multisig line. Furthermore, this complement

```
let multisig_contract (in_stack: stack_t) (fuel: int) : stack_t
    requires { fuel > 0 }
    requires { length in_stack = 1 }
    requires { typ_infer in_stack[0].d = Pair_t parameter storage }
    ensures  { length result = 1 }
    ensures  { typ_infer result[0].d = Pair_t (List_t Operation_t) storage }
    raises { Failing }
  =
    let s = unpair in_stack fuel in
    let s = swap s fuel in
    let s = dup s fuel in
    ...
    let s = iter_multisig s fuel
        ensures {
            typ_infer result[0].d = Comparable_t Nat_t ∧ (* @ valid *)
            typ_infer result[1].d = List_t (Option_t Signature_t) ∧
            typ_infer result[2].d = Comparable_t Bytes_t ∧
            typ_infer result[3].d = Or_t
                        (Pair_t (Comparable_t Mutez_t) (Contract_t Unit_t))
                        (Or_t
                            (Option_t (Comparable_t Key_hash_t))
                            (Pair_t (Comparable_t Nat_t) (List_t Key_t))) ∧
            typ_infer result[4].d = storage
        } in
    ...
```

Figure 11: Part of the multisig contract in WhyML.

was necessary to help the SMTs check some of the pre conditions needed for the instructions in the middle.

This code generated a total of 758 VCs, 750 of which were proven by Alt-ergo [8], Z3 [12] and CVC4[2] proved 4 verification conditions each.

5.2 Factorial

The contract depicted in figure 12 is the Michelson version of the factorial calculation. This contract calculates the factorial of a given natural number interactively. The contract receives as parameter the number whose factorial is going to be calculated and stores the result in the storage. It starts by dropping the previous storage and pushes an initial accumulator and iterator as the value 1. Then it compares the parameter value with 0 and if it's different, it enters the loop to calculate the factorial.

```
parameter nat;
storage nat;
code {  CAR; PUSH @index nat 1; DUP @acc;
        DIP 2 { DUP; PUSH nat 0; COMPARE; NEQ };
        DIG 2;
        LOOP { DIP { DUP;
                     DIP { PUSH nat 1; ADD @ipp } };
               MUL;
               DIP { DIP { DUP };
                     DUP;
                     DIP { SWAP };
                     COMPARE; LE };
               SWAP };
        DIP { DROP; DROP };
        NIL operation; PAIR };
```

Figure 12: Factorial Michelson contract.

Inside the loop *body*, the stack has size three, where the top element is the temporary result, the middle element is the index of the iteration and the bottom element is the input parameter. Since the *body* of the loop is where the computation actually happens, we will focus on the respective portion of WhyML code depicted in figure 13. For shortness we omitted typing information in between instructions as well as size and length pre and post conditions. The only specification that we left was the one regarding functional correctness.

```
let loop_body (s: stack_t) (fuel: int) : stack_t
    requires { match s[0].d,s[1].d with
                | Comparable(Nat res),Comparable(Nat n) → fact (n.value - 1) = res.value
                | _ → false end }
    ensures  { match s[0].d,s[1].d with
                | Comparable (Nat res_old), Comparable (Nat n_old)→
                    fact n_old.value = n_old.value  * res_old.value
                | _ → false
             end }
    ensures  { match s[1].d, result[1].d with
                | Comparable (Nat i), Comparable (Nat b) →  fact i.value  =  b.value
                | _ → false end }
=
 let s =
    let top = s[0] in let s = s[1..] in (* DIP *)
    let s = dup s fuel in
    let s =
      let top = s[0] in let s = s[1..] in (* DIP *)
      let s = push s fuel (mk_wf_data (Comparable (Nat (to_nat 1)))) in
      let s = add s fuel in
    push s fuel top in
  push s fuel top in
  let s = mul s fuel in
  let s =
    let top = s[0] in let s = s[1..] in (* DIP *)
    let s =
      let top = s[0] in let s = s[1..] in (* DIP *)
      let s = dup s fuel in
    push s fuel top in
    let s = dup s fuel in
    let s =
      let top = s[0] in let s = s[1..] in  (* DIP *)
      let s = swap s fuel in
    push s fuel top in
    let s = compare_op s fuel in
    let s = le s fuel in
  push s fuel top in
  swap s fuel
```

Figure 13: Factorial WhyML contract.

The first pre condition assures us that the value stored at the top of the input stack is in fact the value of factorial up to the previous iterations. The last post condition ensures that the value stored at the top of the result stack is the value of factorial up to the current iteration. This code generated 2890 VCs, of which 2671 were proven by Alt-ergo[8], and the remaining 219 by Z3[12].

6 Critical Analysis and Future Work

As stated in the previous sections, we chose Why3 as the main tool for our approach at verifying Tezos smart contracts based on one simple goal, that was to automate as much as possible the verification effort on the user side. Despite this being a clear and well defined objective, we came across some adversities which will be explained in the remainder of this section.

As shown in subsection 2 we defined numerous algebraic data types to reflect the grammar of the Michelson language in a direct correspondence. This decision has proven itself to have consequences, since SMTs find them very hard to work with. One possible solution is to go even further in our shallow embedding and try to map as much as possible every Michelson type directly into WhyML native types. Moreover this would allow us to remove some algebraic constructors from our definition. We first came across this issue when trying to prove safety properties of longer contracts, as in the case of multisig. Somewhere in the middle of the contract one could notice that the SMTs were struggling to prove some pre and post conditions regarding the type that the instructions were expecting. In an attempt to minimise their effort, we decided to propagate typing information throughout the contract. This was a very time consuming process, because, even if the task is systematic, we had to do it manually. This indicates that there is a clear room for automation here. As a future improvement one could write an interpreter that would work alongside the translation mechanism and automatically propagate such conditions throughout the generated WhyML code.

When it comes to functional correctness of a Michelson written contract, it is far from simple for one to infer what a contract does just by looking at it. Since WhylSon is not parsing specification from Michelson contracts yet, we had to add it manually to the contracts we tested. When proving the functional correctness of the factorial contract, we noticed that almost every proof needed numerous assertions in the middle of the WhyML code. Additionally, the proof trees for these goals were far too long, but were almost entirely based on hypothesis rewriting. Going forward we think that we might adopt some sort of proof by reflection mechanism [6, 18] so that this proving process becomes less tedious.

Finally the fact that Michelson is a stack rewriting language makes us operate solely over one data structure with no clear separation between values and instructions. This also increases the struggle that SMTs have when it comes to guaranteeing some frame conditions. With this thought in mind, we are considering adopting a higher level language such as Albert [5] or an intermediate representation Tezla [22]. On one hand, if we choose to go with Albert, we would use it as the input language to WhylSon and then using the Why3 code extraction mechanism described in [21] one could extract the Michelson certified code. Furthermore this last effort only amounts to writing a new printer that translates the internal Why3 AST into compilable Michelson code. On the other hand if we decide to go with Tezla the input language stays the same (i.e. Michelson) but the WhyML generated code would be based on Tezla which we think would facilitate some of the proofs.

7 Related Work

When it comes to formal verification of smart contracts, there are some efforts towards the design of verification platforms for said contracts. For instance, the work of *Nehai* and *Bobot* presented in [19] where they use Why3 to write smart contracts for the Ethereum blockchain [9]. Also *Bhargavan, K., et al.* developed a framework for analysis and verification of functional correctness of Ethereum (ETH) smart contracts by translation into F^* [7]. Moreover, for the same blockchain, in [3], *Abdellatif* and *Brousmiche* used the BIP framework for modeling and verifying said contracts using statistical model checking. Using the Coq Proof Assistant, *Zheng Yang* and *Hang Lei* combined symbolic execution with higher order theorem proving into a tool called FEther aimed at verifying Ethereum smart contracts [26]. The CertiK company has developed a commercial framework for formally verifying smart contracts and blockchain ecosystems [10]. *Marvidou* and *Laska* presented FSolidM in [17], a framework that allows its users to write more secure contracts for ETH using a graphical interface for designing finite state machines that will then be automatically translated into ETH smart contracts. In [23], *Sergey, I., et al.* describe SCILLA, an intermediate language for ETH smart contracts that is amenable to formal verification.

When it comes to Michelson formalisation, *Bernardo, B., et al.* specified a big-step semantic for Michelson using the Coq proof assistant [4] that serves as a base for a verification framework. This work differs from ours because we focus on the automation of the verification process. This fact relies on Why3 where the proof obligations are dispatched to external provers, where as in Coq the proof is made manually. The Archetype language [13] is a domain specific language that allows for formal specification of Tezos smart contracts, which in turn are translated to WhyML for use in Why3, as a back-end. In this work, the contracts to be verified are Archetype contracts. Moreover, we chose Michelson as the object of our verification process, thus mitigating the need for the smart contract writer to learn yet another language for smart contract development.

8 Conclusions

In this paper we presented WhylSon, a tool for automated formal verification of Michelson smart contracts. Moreover, WhylSon is the result of several implementations also described in this document, namely a Michelson parser, an axiomatic semantics and a translation function, all leading to a shallow embedding of Michelson in WhyML.

The first steps to the automatic proof in Why3 of manually annotated Michelson smart contracts were done, since we were able to use our axiomatic semantics and our Why3 plugin to successfully write and prove several Michelson contracts. Furthermore, the plugin development proved itself simple, due to the fact that the Why3 platform exposes its API as an OCaml library.

In practice we found that some of the proof trees were bigger than expected and required user intervention, thus threatening our main purpose of automation. We are aware that this is a consequence of our encoding of Michelson types as tree-like data structures. Our perspective is that using one or more of the solutions discussed in 6 we can mitigate this issue, leading us to a platform that allows the user to conduct formal verification of Michelson written smart contract with an elevated degree of automation.

As a final thought, we think that proving Michelson contracts has a certain advantage over proving some other formulation, since what is effectively executed is the Michelson smart contract and also because this approach can be used a back-end in developing reliable smart contract in any higher level language such a LIGO[1] or SmartPy[16].

Nevertheless, smart contracts developers will implement their smart contracts in a higher level language than Michelson. In this setting, it is also relevant to be able to formally prove these smart contracts at a level that programmers understand and be involved with. So an interesting long-term line of work to explore is to connect WhylSon with certifying-certified compilation techniques and platforms. For instance, we should evaluate how the integration of WhylSon with, *e.g.*, the Archetype platform [13], that also makes use of Why3, can benefit the automatic proof of Michelson smart contracts. We should also evaluate how WhylSon could benefit from rigorously designed compilers as the one designed for Albert [5] to Mi-cho-coq [4]. For the Why3 platform, such an endeavour could make use of the techniques introduced by *Clochard, M., et al.* in [11].

References

[1] G. Alfour. LIGO. URL: https://ligolang.org/.

[2] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. Cvc4. In *International Conference on Computer Aided Verification*, pages 171–177. Springer, 2011.

[3] A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in bip. In *Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM'06)*, pages 3–12. Ieee, 2006.

[4] B. Bernardo, R. Cauderlier, Z. Hu, B. Pesin, and J. Tesson. Mi-cho-coq, a framework for certifying tezos smart contracts. In *FMBC'19: Workshop on Formal Methods for Blockchains*, 2019.

[5] B. Bernardo, R. Cauderlier, B. Pesin, and J. Tesson. Albert, an intermediate smart-contract language for the Tezos blockchain. *arXiv:2001.02630 [cs]*, Jan. 2020. arXiv:2001.02630.

[6] Y. Bertot and P. Castéran. Proof by Reflection. In Y. Bertot and P. Castéran, editors, *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*, Texts in Theoretical Computer Science An EATCS Series, pages 433–448. Springer, Berlin, Heidelberg, 2004. doi:10.1007/978-3-662-07964-5_16.

[7] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Béguelin. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, PLAS '16, pages 91–96, New York, NY, USA, 2016. ACM. URL: http://doi.acm.org/10.1145/2993600.2993611, doi:10.1145/2993600.2993611.

[8] F. Bobot, S. Conchon, E. Contejean, M. Iguernelala, S. Lescuyer, and A. Mebsout. The alt-ergo automated theorem prover, 2008, 2013.

[9] V. Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 2014.

[10] CertiK. : Building fully trustworthy smart contracts and blockchain ecosystems, 2019. [Online; https://certik.org/docs/white_paper.pdf, accessed April-2020].

[11] M. Clochard, C. Marché, and A. Paskevich. Deductive verification with ghost monitors. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–26, Jan. 2020. doi:10.1145/3371070.

[12] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 337–340, Berlin, Heidelberg, 2008. Springer. doi:10.1007/978-3-540-78800-3_24.

[13] R. B. S. P. e. Duhamel, G. Archetype: a tezos smart contract development solution dedicated to contract quality insurance, 2019. [Online; https://docs.archetype-lang.org/ accessed 14-April-2020].

[14] J.-C. Filliâtre and A. Paskevich. Why3—where programs meet provers. In *European Symposium on Programming*, pages 125–128. Springer, 2013.

[15] L. Goodman. Tezos: A self-amending crypto-ledger position paper, 2014.

[16] F. Maurel and S. C. Arena. SmartPy. URL: https://smartpy.io/.

[17] A. Mavridou and A. Laszka. Designing secure ethereum smart contracts: A finite state machine based approach, 2017. arXiv:1711.09327.

[18] G. Melquiond and R. Rieu-Helft. A Why3 Framework for Reflection Proofs and its Application to GMP's Algorithms. In D. Galmiche, S. Schulz, and R. Sebastiani, editors, *9th International Joint Conference on Automated Reasoning*, number 10900 in Lecture Notes in Computer Science, pages 178–193, Oxford, United Kingdom, July 2018. URL: https://hal.inria.fr/hal-01699754, doi:10.1007/978-3-319-94205-6_13.

[19] Z. Nehai and F. Bobot. Deductive proof of ethereum smart contracts using why3. In *FMBC'19: Workshop on Formal Methods for Blockchains*, 2019.

[20] C. Newman and G. Klyne. Date and Time on the Internet: Timestamps. RFC 3339, July 2002. URL: https://rfc-editor.org/rfc/rfc3339.txt, doi:10.17487/RFC3339.

[21] M. J. Parreira Pereira. *Tools and Techniques for the Verification of Modular Stateful Code*. Theses, Université Paris-Saclay, Dec. 2018. URL: https://tel.archives-ouvertes.fr/tel-01980343.

[22] J. Santos Reis, P. Crocker, and S. Melo de Sousa. Tezla, an intermediate representation for static analysis of michelson smart contracts, 2020. ArXiV n. submit/3191624.

[23] I. Sergey, A. Kumar, and A. Hobor. Scilla: a smart contract intermediate-level language, 2018. arXiv:1801.00687.

[24] Tezos Foundation. Michelson: the language of smart contracts in tezos, 2020. [Online; https://tezos.gitlab.io/whitedoc/michelson.html#language-semantics last accessed 15-April-2020].

[25] Tezos Foundation. Multisig conctract, 2020. [Online; https://tezos.gitlab.io/whitedoc/michelson.html#multisig-contract last accessed 15-April-2020].

[26] Z. Yang and H. Lei. Fether: An extensible definitional interpreter for smart-contract verifications in coq. *IEEE Access*, 7:37770–37791, 2018.