4th International Workshop on Formal Methods for Blockchains

FMBC 2022, August 11, 2022, Haifa, Israel

Edited by

Zaynah Dargaye
Clara Schneidewind

Editors

Zaynah Dargaye
Nomadic Labs, Paris, France
Zaynah.Dargaye@nomadic-labs.com

Clara Schneidewind
MPI-SP, Bochum, Germany
clara.schneidewind@mpi-sp.org

ACM Classification 2012
Security and privacy → Formal methods and theory of security; Security and privacy → Logic and verification; Theory of computation → Program verification; Software and its engineering → Formal software verification; Security and privacy → Distributed systems security; Computer systems organization → Peer-to-peer architectures

ISBN 978-3-95977-250-1

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-250-1.

Publication date
September, 2022

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at https://portal.dnb.de.

License
This work is licensed under a Creative Commons Attribution 4.0 International license (CC-BY 4.0): https://creativecommons.org/licenses/by/4.0/legalcode.

In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work under the following conditions, without impairing or restricting the authors' moral rights:
- Attribution: The work must be attributed to its authors.

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/OASIcs.FMBC.2022.0

OASIcs – OpenAccess Series in Informatics

OASIcs is a series of high-quality conference proceedings across all fields in informatics. OASIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

Editorial Board

- Daniel Cremers (TU München, Germany)
- Barbara Hammer (Universität Bielefeld, Germany)
- Marc Langheinrich (Università della Svizzera Italiana – Lugano, Switzerland)
- Dorothea Wagner (*Editor-in-Chief*, Karlsruher Institut für Technologie, Germany)

ISSN 1868-8969

https://www.dagstuhl.de/oasics

Contents

Preface
Zaynah Dargaye and Clara Schneidewind 0:vii

Invited Talk

MEV-Freedom, in DeFi and Beyond
Massimo Bartoletti 1:1–1:1

Regular Papers

Finding Smart Contract Vulnerabilities with ConCert's Property-Based Testing Framework
Mikkel Milo, Eske Hoy Nielsen, Danil Annenkov, and Bas Spitters 2:1–2:13

Automatic Generation of Attacker Contracts in Solidity
Ignacio Ballesteros, Clara Benac-Earle, Luis Eduardo Bueso de Barrio, Lars-Åke Fredlund, Ángel Herranz, and Julio Mariño 3:1–3:14

Proofgold: Blockchain for Formal Methods
Chad E. Brown, Cezary Kaliszyk, Thibault Gauthier, and Josef Urban 4:1–4:15

Multi: A Formal Playground for Multi-Smart Contract Interaction
Martín Ceresa and César Sánchez 5:1–5:16

4th International Workshop on Formal Methods for Blockchains (FMBC 2022).
Editors: Zaynah Dargaye and Clara Schneidewind
OASICS OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

The 4th International Workshop on Formal Methods for Blockchains (FMBC) took place on August 11, 2022, as part of CAV 2022, the 34th International Conference on Computer Aided Verification, and FLoC 2022, the 8th Federated Logic Conference. FMBC's purpose is to be a forum to identify theoretical and practical approaches that apply formal methods to blockchain technology.

This fourth edition of FMBC attracted 12 submissions on topics such as verification and testing of smart contracts or analysis of blockchain protocols. Each paper was reviewed by at least three program committee members or appointed external reviewers. This led to a selection of 4 (long) papers that were presented at the workshop as regular talks, as well as 6 works that were presented as lightning talks. Additionally, we were very pleased to have an invited keynote by Massimo Bartoletti (Università degli Studi di Cagliari, Italy).

This volume contains the papers selected for regular 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 Bruno Bernado and Diego Marmsoler, the Program Committee Chairs of the last editions of FMBC, for their constant support. Further, we would like to thank Shaull Almagor and Guillermo A. Pérez, Workshop Chairs of FLoC 2022, for their guidance. Finally, we would like to express our gratitude to our sponsors Algorand, Cluster of Excellence CASA – Cyber Security in the Age of Large-Scale Adversaries, IOHK, and Nomadic Labs for their generous support.

September 2022

Zaynah Dargaye
Clara Schneidewind

4th International Workshop on Formal Methods for Blockchains (FMBC 2022).
Editors: Zaynah Dargaye and Clara Schneidewind
OASICS OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Program Committee

Wolfgang Ahrendt
Chalmers University of Technology, Sweden

Leonardo Alt
Ethereum Foundation, Germany

Lacramioara Astefanoaei
Nomadic Labs, France

Roberto Blanco
MPI-SP, Germany

Joachim Breitner
Germany

Achim Brucker
University of Exeter, UK

Ethan Cecchetti
University of Maryland, USA

Manuel Chakravarty
IOHK & Tweag, Netherlands

Jing Chen
Algorand Inc, USA

Zaynah Dargaye
Nomadic Labs, France

Jérémie Decouchant
TU Delft, Netherlands

Antonella Del Pozzo
Université Paris-Saclay & CEA & List,
France

Dana Drachsler Cohen
Technion, Israel

Cezara Dragoi
INRIA & ENS & CNRS & PSL, France

Ansgar Fehnker
Twente, Netherlands

Dominik Harz
Interlay & Imperial College London, UK

Lars Hupel
INNOQ, Germany

Igor Konnov
Informal Systems, Austria

Paul Laforgue
Nomadic Labs, France

Julian Nagele
Bank of America, USA

Russel O'Connor
Blockstream

Maria Potop-Butucaru
LIP6, France

Albert Rubio
Complutense University of Madrid, Spain

César Sanchez
IMDEA, Spain

Clara Schneidewind
MPI-SP, Germany

Sun Meng
Peking University, China

Simon Thompson
IO Global, UK

Josef Widder
Informal Systems, Austria

4th International Workshop on Formal Methods for Blockchains (FMBC 2022).
Editors: Zaynah Dargaye and Clara Schneidewind
OASICS OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Supporting Reviewers

Mojtaba Eshghie

4th International Workshop on Formal Methods for Blockchains (FMBC 2022).
Editors: Zaynah Dargaye and Clara Schneidewind
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

MEV-Freedom, in DeFi and Beyond

Massimo Bartoletti
University of Cagliari, Italy

Abstract

Maximal Extractable Value (MEV) refers to a class of recent attacks on public blockchains, where adversaries with the power to reorder, drop or insert transactions in a block can "extract" value from user transactions in the mempool. Empirical research has shown that mainstream DeFi protocols, like e.g. Automated Market Makers and Lending Pools, are massively targeted by MEV attacks. This has detrimental effects on their users, on transaction fees, and on the congestion of blockchain networks. Despite the growing knowledge on MEV attacks on blockchain protocols, an exact definition is still missing. Indeed, formally defining these attacks is an essential prerequisite to the design of provably secure, MEV-free blockchain protocols. In this talk, we propose a formal definition of MEV, based on a general, abstract model of blockchains and smart contracts. We then introduce MEV-freedom, a property enjoyed by contracts resistant to MEV attacks. We validate this notion by rigorously proving that Automated Market Makers and Lending Pools are not MEV-free. We finally discuss how to design MEV-free contracts.

2012 ACM Subject Classification Security and privacy → Formal methods and theory of security

Keywords and phrases Blockchain, Smart Contracts, Formal Security Notion

Digital Object Identifier 10.4230/OASIcs.FMBC.2022.1

Category Invited Talk

Bio

Massimo Bartoletti is Associate Professor at the Department of Mathematics and Computer Science of the University of Cagliari. His research activity concerns the development of tools and techniques for the specification, analysis and verification of software, with a special emphasis on security. Massimo Bartoletti is founder of the laboratory "BlockchainUnica" (`http://blockchain.unica.it`), one of the largest academic research group on blockchain technologies in Italy, director of the node of the Cyber Security National Lab for the University of Cagliari, and core member of the CINI working group on Blockchain. The laboratory is currently investigating several aspects of blockchain technologies, among which custom Domain-Specific Languages for smart contracts. He is principal investigator of several R&D projects on blockchain technologies, and member of the scientific board of several workshops on blockchain technologies. He is also the organisation chair of the first International School on Algorand Smart Contracts, funded by a grant of the Algorand Foundation.

© Massimo Bartoletti;
licensed under Creative Commons License CC-BY 4.0
4th International Workshop on Formal Methods for Blockchains (FMBC 2022).
Editors: Zaynah Dargaye and Clara Schneidewind; Article No. 1; pp. 1:1–1:1
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Finding Smart Contract Vulnerabilities with ConCert's Property-Based Testing Framework

Mikkel Milo
Department of Computer Science, Aarhus University, Denmark

Eske Hoy Nielsen
Department of Computer Science, Aarhus University, Denmark

Danil Annenkov
Department of Computer Science, Aarhus University, Denmark

Bas Spitters
Department of Computer Science, Aarhus University, Denmark

Abstract

We provide three detailed case studies of vulnerabilities in smart contracts, and show how property based testing would have found them: 1. the Dexter1 token exchange; 2. the iToken; 3. the ICO of Brave's BAT token. The last example is, in fact, new, and was missed in the auditing process.

We have implemented this testing in ConCert, a general executable model/specification of smart contract execution in the Coq proof assistant. ConCert contracts can be used to generate verified smart contracts in Tezos' LIGO and Concordium's rust language. We thus show the effectiveness of combining formal verification and property-based testing of smart contracts.

2012 ACM Subject Classification Software and its engineering → Formal methods; Software and its engineering → Software verification and validation

Keywords and phrases Smart Contracts, Formal Verification, Property-Based Testing, Coq

Digital Object Identifier 10.4230/OASIcs.FMBC.2022.2

Supplementary Material *Software (The ConCert Framework)*: `https://github.com/AU-COBRA/ConCert/tree/fmbc2022`, archived at `swh:1:dir:00e8602bf86a672643073ed9b89a9de8436247a6`

Funding This research was partially supported by a grant from Nomadic Labs and by the Concordium Blockchain Research Center.

Acknowledgements We would like to thank the LIGO team and in particular Tom Jack, Raphaël Cauderlier, Exequiel Rivas, Rémi Lesénéchal, Gabriel Alfour, Thomas Letan and Arvid Jakobsson for the discussions about testing for LIGO.

1 Introduction

Blockchain-based technologies have seen rising interest in recent years. This can be attributed to their ability to sustain a public distributed ledger with a high degree of reliability, integrity, and transparency, without requiring a trusted third party. Smart contracts are distributed applications deployed on a blockchain. They are typically used for sensitive transactions, for example, carrying large amounts of money or other valuable assets, but in principle, they can perform any computation. Once a smart contract is deployed on the blockchain, it is impossible to change its source code. The blockchain ensures that contracts are executed correctly according to the execution model. However, it gives no guarantee that the smart contract's code is correct. Like other programs, smart contracts are susceptible to bugs.

© Mikkel Milo, Eske Hoy Nielsen, Danil Annenkov, and Bas Spitters;
licensed under Creative Commons License CC-BY 4.0

Some attacks on smart contracts have resulted in substantial losses. For example, the "DAO attack" on Ethereum, where $50 million worth of cryptocurrency was stolen due to a re-entrancy vulnerability[1]. In April 2020, an attacker exploited a re-entrancy bug in the Lendf.me platform, resulting in a loss of about 99.5% of the platform's funds (∼$25 million). In 2021 cryptocurrency-related crimes including smart contract attacks resulted in losses of approximately $14 billion [6]. Hence, having a high assurance that a smart contract implementation is free of bugs is imperative. Unit testing is often used in the process of smart contract development. However, subtle bugs related to smart contract state evolution over a series of calls, or interaction with other contracts often cannot be captured by conventional unit testing. Moreover, even proving functional correctness properties is not sufficient, as it was exemplified by the Dexter contract considered in Section 3. To address such issues, we are using the ConCert framework in the Coq proof assistant which facilitates formal verification and property-based testing of smart contracts.

Contributions

We present the details of the property-based testing functionality of the ConCert framework. The testing functionality was presented briefly in earlier works on ConCert [3, 2]. This paper contributes to the property-based testing functionality of ConCert and presents three case studies demonstrating how ConCert can be used to find real-world bugs in smart contracts. Contributions to the testing framework include counterexample shrinking, negative testing capabilities, improved customisation and usability improvements.

The first two case studies show how ConCert could have been used to find bugs that were found in smart contracts by auditors and attackers. The last case study shows how we used ConCert to find new bugs which could have led to upwards of $8 million being stolen or frozen.

2 ConCert Overview

In this section, we give a brief overview of the ConCert framework, focusing on the smart contract execution layer and property-based testing. ConCert is open-source, and available at `https://github.com/AU-COBRA/ConCert/`.

2.1 Pipeline

The pipeline overview is presented in Figure 1. We start by developing a smart contract in Coq using the ConCert infrastructure. That is, smart contracts are written in Gallina, a functional language of Coq that shares similarities with other functional languages. They are just ordinary functions that use some pre-defined blockchain primitives provided by the ConCert infrastructure. This facilitates porting smart contracts written in functional smart contract languages to Coq.[2] Even for a language like Solidity, this is fairly straightforward. We then can write a specification and test the smart contact function semi-automatically against it, using the integration with QuickChick [8]. With more effort, we can also prove the properties of smart contracts using the ConCert infrastructure. Proofs and tests crucially use the execution layer to reason about interacting contracts (see more details in Section 2.2), which enables us to capture properties beyond the mere functional correctness of a single contract invocation (see Section 3).

[1] `https://www.wired.com/2016/06/50-million-hack-just-showed-dao-human/`
[2] E.g. LIGO, Liquidity, Sophia

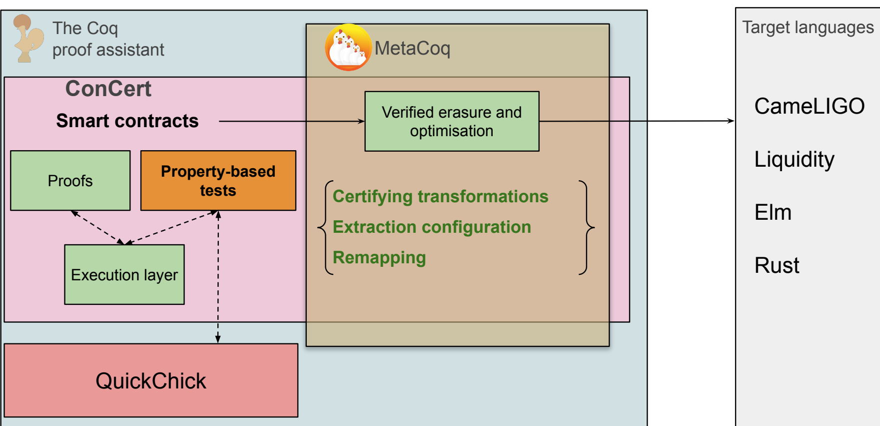


Figure 1 The pipeline.

After testing and verification, one can obtain an executable implementation in one of the supported smart contract languages through *code extraction*. Our development uses the verified erasure procedure of MetaCoq [9] with verified optimisations and certifying pre-processing of ConCert. This gives us a code-generation procedure with strong correctness guarantees and a small trusted computing base consisting of MetaCoq's *quote* functionality, the pretty-printers into the target languages and the extraction configuration. Note that ConCert's extraction does not use unsafe coercions, like `Obj.magic` in OCaml. Therefore, the resulting code is type-checked as a regular user-defined contract. Additionally, extraction configuration involves mappings from ConCert's primitives to specific primitives for each supported target blockchain. These mappings contribute to the TCB and are carefully defined together with experts for a particular target blockchain. Outside of the ConCert pipeline, the compilers used to produce low-level code (e.g. Michelson) from extracted contracts are blockchain-specific and also contribute to the overall TCB.

2.2 Smart Contract Execution Layer

The execution layer provides a model which facilitates reasoning about contract execution traces. This makes it possible to state and prove temporal properties of interacting smart contracts. Smart contracts in ConCert are modelled by abstracting a number of blockchains.[3] These blockchains can be characterised as variants of a message-passing model. ConCert models core behaviour for such models. Each blockchain can have some specific features not present in the ConCert execution model directly (e.g. Tezos' views), but similar behaviour can be expressed through message passing. Contracts which use such features are not directly expressable in ConCert. Some contracts might not be extractable to some targets if they use concepts that cannot be mapped to the target blockchain.

A contract consists of two functions:

- `init : Chain → ContractCallContext → Setup → option State`
  The initialisation function is called after the contract is deployed on the blockchain. The first parameter of type `Chain` gives access to data about the blockchain (e.g. current chain height). The `ContractCallContext` parameter provides data about the current call (e.g. caller address, amount sent to the contract). `Setup` represents initialisation parameters.

[3] E.g. Concordium, Tezos, Dune, Æternity

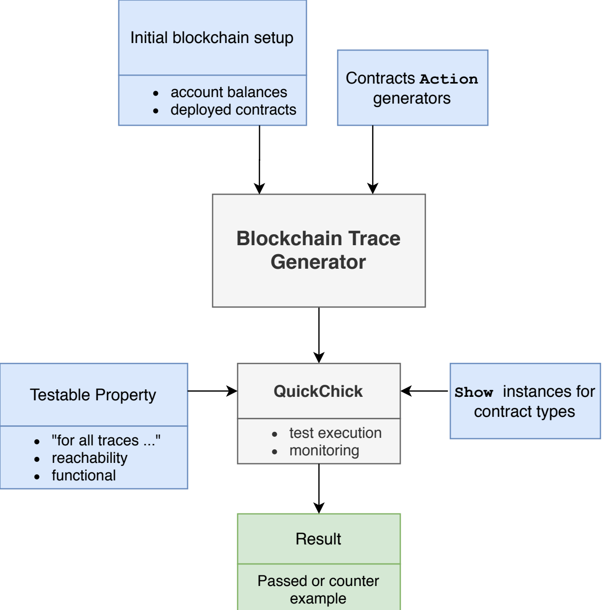


Figure 2 Property-based Testing in ConCert.

- `receive : Chain → ContractCallContext → State → option Msg → option (State * list ActionBody)` This function represents the main functionality of the contract that is executed for each call to the contract. `Chain` and `ContractCallContext` are the same as for `init`. The parameter of type `State` is the current state of the contract; `Msg` is a user-defined type of messages that contract accepts (the *entrypoints* of the contract). The result of a successful execution is a new state and a list of *actions* represented with `ActionBody`. The actions can be transfers, calls to other contracts (including itself), and contract deployments.

Both `receive` and `init` are ordinary Coq functions, making them convenient to reason about. However, reasoning about the contract functions in isolation is not sufficient, as many deployed contracts actually consist of a collection of interacting contracts, for example for the sake of modularity. One call to `receive` potentially emits more calls, which can create complex call graphs between deployed contracts. Therefore, it is necessary to consider execution traces to prove some safety properties of smart contracts. An execution trace `ChainTrace` is the reflexive, transitive closure of a proof-relevant `ChainStep` relation, which essentially captures the addition of a block to the blockchain. In this step, any *actions* (such as contract calls and transfers) coming from external users are executed.

`ChainTrace` gives a relational operational semantics for the executions process. The semantics is non-deterministic since it allows for arbitrary execution order for the actions emitted by contract calls. Thus, ConCert provides two executable implementations: one follows depth-first and the other follows breadth-first order. It also provides proof that if running `add_block` succeeds, it results in a valid instance of `ChainTrace`. Having an executable implementation is crucial for property-based testing.

2.3 Property-based Testing framework

Property-based testing (henceforth abbreviated *PBT*), also known as *random-property testing*, is a technique for testing where test data is generated pseudo-randomly and tested in large quantities against some decidable property. We integrate the PBT library *QuickChick* [8] with

the execution framework to obtain a method for testing contract executions. In particular, we support testing the functional correctness of contracts but also testing (decidable) properties of entire execution traces. The overview of the testing framework is given in Figure 2.

In brief, the PBT framework works by having the user provide *generators* for the `Msg` type of the contract(s) tested. In this context, generators are functions that produce pseudo-random values of the given type. These generators are used to populate randomly generated execution traces with pseudo-random contract calls during testing with QuickChick. The user also configures the initial blockchain setup consisting of account balances and contracts that are currently available for interaction (deployed contracts). QuickChick also uses `Show` type class instances to print test results (e.g. counterexamples).

For example, consider how to test a token contract whose `Msg` type is

```
Inductive Msg :=
   transfer of (address * address * nat)
 | approve of (address * address * nat).
```

That is, it has two entrypoints: one for transferring tokens between the two given addresses and one for approving an address to spend a given number of tokens on behalf of another address. Generating pseudo-random values of `Msg` then amounts to either generating a `transfer` or an `approve`, and populating it with parameters by using the generators for `address` and `nat`. We can either implement this manually or have QuickChick automatically derive such a generator[4]. Note that we might prefer to implement this manually since we might want to ensure that the number of tokens to be transferred in `transfer` is never larger than the balance of the sender. We provide various combinators to make it easy and convenient to implement complex generators.

Suppose we want to test that `transfer` updates the internal balances correctly. In ConCert, this functional correctness property is specified by using pre- and post-conditions. Testing such a property with QuickChick could look like

```
QuickChick ({{msg_is_transfer}} Token.receive {{transfer_correct}}).
```

The code above states that if the incoming message is a transfer, then after executing the token contract's `receive` function, its state should be consistent with a predicate `transfer_correct`. By default, QuickChick will generate 10.000 inputs and test that the property is satisfied in all of them, or otherwise report a counterexample. The counterexamples reported are automatically minimized by the PBT framework to produce smaller counterexamples that are easy to understand. From our experience, these tests typically take less than a minute (see Section 7).

One can also test whether some state is reachable from the given state. For example, the following test

```
QuickChick (token_cb ~~> (person_has_tokens person_3 42)).
```

shows that from the state `token_cb` with three addresses participating in the token there is a state where `person_3` has 42 tokens. The corresponding trace is reported to the user.

[4] Due to limitations of QuickChick, the `Derive` command fails for some parameterised inductive types, e.g. `Msg` type in implicitly parameterised with some blockchain configuration. We have reported this issue: `https://github.com/QuickChick/QuickChick/issues/286`

3 Dexter decentralized exchange

In this section, we consider a bug in (an earlier version of) Dexter, a decentralized token exchange contract on the Tezos blockchain. The bug would have allowed an attacker to manipulate exchange rates to obtain unintended profit through a simple attack. The contract had previously been formally verified for functional correctness[5]. However, this bug can only be discovered when considering *execution traces* - that is, sequences of contract calls. We demonstrate how this bug can be found by testing a *natural* specification on traces. So, we argue that this bug would likely have been discovered when using ConCert as part of the specification process.

The Dexter exchange smart contract is used for exchanging tokens and tez (the on-chain currency of Tezos), it implements a so-called *constant-product market*, which means that the total value of the contract never decreases. A property of such markets is that the exchange rate cannot be significantly manipulated unless a party owns most of the market's assets [1]. The rate at which tokens and tez can be exchanged is calculated dynamically at each trade according to the function

$$getInputPrice(Ts, Ts_{reserve}, Tez_{reserve}) = \frac{Ts \cdot 997 \cdot Tez_{reserve}}{Ts_{reserve} \cdot 1000 + Ts \cdot 997}$$

where Ts are the tokens being exchanged, $Ts_{reserve}$ is the reserve of tokens held by the Dexter contract, and $Tez_{reserve}$ is the contracts tez reserve.

One key property of constant-product markets, that cannot be verified from functional correctness alone, is that splitting trades is never profitable. Specifically, suppose a user trades N tokens for Z tez. Suppose this trade is split into $k > 1$ trades, totalling N tokens for a total of Z' tez. Then it should be the case that $Z' \leq Z$.

In ConCert, we can state this property by asserting that for each block added to generated traces, the total amount of tez gained from trades does not exceed what the user would have gained from trading the same amount of tokens in a single exchange. The full Coq definition can be found in `examples/dexter/DexterTests.v`

With this test, our PBT framework automatically finds a counterexample that violates the property. The counterexample show two consecutive exchanges; first trading 14 tokens for 5 tez, then 16 tokens for an additional 5 tez. However, the payout for a single trade of 30 tokens would have been 9 tez, netting the user an extra one tez from splitting the trade. The vulnerability is due to a combination of Tezos' breadth-first execution model[6] and the way the contract tracks its asset reserves. Concretely the problem is that in breadth-first both trades are executed before the actions emitted by the trades are executed, meaning that the second trade will start before the tez and tokens from the first trade have finished being transferred. The contract accounts for this by manually tracking the number of tokens, but fails to do the same for the tez reserve. Thus when the second trade starts the contract uses the wrong tez reserve for calculating the exchange rate. A strength of ConCert is that it allows testing in both depth-first and breadth-first execution order, running the same test with depth-first shows no vulnerability.

The bug was fixed prior to the deployment of Dexter.

---

[5] `https://research-development.nomadic-labs.com/dexter-decentralized-exchange-for-tezos-formal-verification-work-by-nomadic-labs.html`

[6] Tezos moved to depth-first execution order after Dexter was developed

4 iToken

In this section, we show how the bZx iToken smart contract was compromised and how ConCert could have discovered this vulnerability. The iToken smart contract is an interest accumulating ERC20 token used as part of the bZx decentralized finance platform. In September 2020 an attacker stole $8 million worth of cryptocurrency by exploiting a vulnerability in the iToken contract caused by a misplaced line of code[7]. This vulnerability was missed by two audits of the platform. The vulnerability was in the tokens `transferFrom`, which is used to transfer tokens between users. The transfer logic was implemented in the following way:

```
uint256 balanceFrom = balances[from];
uint256 balanceTo = balances[to];
balances[from] = balanceFrom.sub(amount);
balances[to] = balanceTo.add(amount);
```

This logic would have been safe had lines 2 and 3 been swapped. To see where this goes wrong, consider the case where `from = to`. In this case, the transferred amount would be subtracted from the sender's balance in line 3. However, in line 4 the original balance of the sender is used to add the transferred amount to the sender's balance, resulting in the sender ending gaining tokens through the self-transfer.

This bug could be found using the PBT framework by writing a test checking that the balance remains the same after a self-transfer. However, such a test would require knowledge of the possibility of a bug in this edge case. Instead, we formulate the property that *the sum of all balances should remain unchanged after a call*, with the exception of minting and burning calls. In ConCert testing such a property looks like:

```
Definition msg_is_not_mint_or_burn state msg :=
  match msg with
  | mint _ | burn _ ⇒ false
  | _ ⇒ true
  end.
Definition sum_balances_unchanged chain cctx (old_state : State) (msg : Msg)
                                     (result : option (State * list ActionBody)) : bool :=
  let balances_sum state := sum s.(balances) in
  match result with
  | Some (new_state, _) ⇒ balances_sum old_state =? balances_sum new_state
  | None ⇒ true (* Return true in the case that nothing changed *)
  end.

QuickChick ({{msg_is_not_mint_or_burn}} iTokenContract {{sum_balances_unchanged}})
```

examples/iTokenBuggy/iTokenBuggyTests.v:sum_balances_unchanged

By running the test, we indeed obtain a minimal counterexample showing that self-transfers violate the property.

5 Basic Attention Token

In this section, we show how ConCert was used to find new bugs, that were missed by several audits, in the Basic Attention Token (BAT) smart contract. BAT is an Ethereum initial coin offering smart contract developed by Brave. It is a combination of an ERC-20 token and a crowdsale contract, where users can fund ether to Braves' project in return for BAT tokens. The crowdsale runs for a fixed amount of blocks, after which the funding either succeeds or

[7] https://bzx.network/blog/incident

fails. If funding succeeds, Brave receives all the ether raised. If it fails, all users can claim a refund of their ether by burning their tokens. As the contract owners, Brave get a fixed amount of free tokens to spend.

We test functional correctness using a similar Hoare triple test as shown in Section 2.3. In addition, we formulated five key safety properties.

1. **Funding is final:** Once the contract enters its funded state it cannot leave it again.
2. **Funding possible:** If there is enough ETH in the blockchain to reach the funding goal, then it should be possible to reach a state in which the funding succeded.
3. **No refunding for owners:** The free tokens given to the owners should not be refundable.
4. **Refund guarantee:** There should always be enough ETH in the contract balance to refund all funded tokens. Unless funding succeded.
5. **No frozen funds:** It should always be possible to completely drain the contract balance, so no ETH gets permanently frozen.

Through testing, we found that only the first property holds. Most of the bugs occur from combining token and crowdsale functionality and both parts behave safely on their own. *This highlights that composing contracts is nontrivial and can easily introduce subtle bugs.*

5.1 Test Setup

In Sections 3 and 4 we showed that ConCert could find known bugs. For those, it was not so important whether the generators would cover the entire input space. However, when testing a complex contract with the purpose of finding potentially unknown bugs, it is crucial to have good generators. A good quality generator should be able to cover the entire input space of the smart contract and have a good balance between generating calls that succeed and calls that fail. Using automatically derived generators will often result in too many failing calls for complex smart contracts. For testing BAT we take the approach of combining manually written generators designed to only produce valid calls with generators that are likely to produce invalid calls. That is, for each entrypoint, we define two generators. This is illustrated in Figure 3. The `finalize` entrypoint is an entrypoint that transitions the contract from funding to the funded state. It can only be called by the owner after funding succeeds. The first generator `gFinalize` only produces calls that we expect to succeed, while the `gFinalizeinvalid` generator will generate calls with an arbitrary sender, which is unlikely to be valid. We use the `x` ← `e1` ;; `e2` monadic bind notation to bind generated values. The generators for potentially invalid calls can be automatically derived using QuickChick. All the generators are combined into a single call generator.

This approach gives us a generator that can cover the entire input space while still allowing us to tune the distribution of valid and invalid calls to different entrypoints. Using the PBT framework we can measure statistics about the generator and use that to tune the distribution.

5.2 Finding Vulnerabilities

We test each of the five safety properties for the BAT contract defined in Section 5. Here we detail a few of the tests.

A key property is that the contract doesn't deadlock, i.e. with enough user support it should always be possible to reach the funded state. Since ConCert can test reachability of states we can easily state this property by combining the reachability checker with a deployment configuration generator. The following test states that for any BAT deployment

```
Definition gFinalize env contract_state : G (option (Address * Msg)) :=
  if (isFullyFunded env contract_state) (* Check if funding succeded *)
  then returnGen (Some (fund_addr, finalize)) (* Call finalize from owner address *)
  else returnGen None. (* Don't return call if not funded *)
Definition gFinalizeInvalid env contract_state : G (Address * Msg) :=
  sender ← gAddress ;; (* Generate arbitrary address *)
  returnGen (sender, finalize).
```

examples/bat/BATGens.v:gFinalize

Figure 3 Generators for the `finalize` entrypoint.

configuration there should exist a trace from the state where BAT is deployed with that configuration to a state where the contract is funded.

```
QuickChick (forAll gBATSetup (build_init_cb (fun cb ⇒ cb ~~> is_finalized))).
```

examples/bat/BATTests.v

Here `gBATSetup` is the configuration generator, `build_init_cb` builds an inital state with the contract deployed, and `is_finalized` checks for a given blockchain state if the contract is funded. By running the test, we obtain counterexamples showing four classes of configurations where the contract cannot be fully funded. One of them is the case where the funding period is empty or already over at the time of deployment. Ideally, the contract should have included a check at deployment preventing such configurations.

A crucial safety property is that any user who donated should be guaranteed their money back in case of failed funding. By testing the functional correctness of entrypoints, we already know that the contract will always refund the correct amount and will always succeed, given that the contract has enough funds. Therefore, testing refund guarantee reduces to checking that there is always enough funds to refund all tokens held by “real” users. Here we distinguish between real users of the contract and the owner, because the owner’s free tokens should not be counted. That is, we want to test that the following is always true.

$$\texttt{contractBalance} \geq \frac{totalTokenSupply - ownersTokens}{tokenExchangeRate}$$

In ConCert a test of this looks like:

```
Definition contract_balance_lower_bound (cs : ChainState) :=
  let contract_balance := env_account_balances cs contract_base_addr in
  (* Get BAT contract state *)
  match get_contract_state State cs contract_base_addr with
  | Some cstate ⇒
    (* Get token balance of owner *)
    let bat_fund_balance := with_default 0 (FMap.find owner (balances cstate)) in
      if cstate.(isFinalized)
      then checker true (* Case where refunds are not permitted *)
      (* Assert that there is enough ETH to refund all tokens held by "real" users *)
      else checker (Z.geb contract_balance
        (Z.of_N (((total_supply cstate) - bat_fund_balance) / cstate.(tokenExchangeRate))))
  | None ⇒ checker true (* Case where contract isn't deployed *)
  end.
QuickChick (forAllChainState contract_balance_lower_bound)
```

examples/bat/BATTests.v:contract_balance_lower_bound

Running the test we get the following minimized counterexample from the testing framework.

```
Chain{|
  Block 1 [Action{act_from: 10, act_body: (act_deploy 0, Setup{owner:=17;...})}];
  Block 2 [Action{act_from: 17, act_body: (act_call 128, 0, transfer 16 14)}]
|}
```

This counterexample shows a trace where the BAT contract is deployed in the first block, after which the owner (address 17) immediately transfers some of its free tokens to another user. This is possible because the contract combines crowdsale and token contract behaviour. This violates two of the safety properties because nothing is preventing the second user from refunding the transferred tokens. Thus it is possible for the free tokens given to the owner to be refunded by first transferring them. This also breaks the property that all real users should be guaranteed a refund because if the owner refunds some of the free tokens then there is no longer enough ETH to refund all tokens held by real users.

The remaining safety properties were tested using similar methods.

6 Related Work

Various testing approaches have been applied to smart contracts. Tools like Truffle[8] for Ethereum or SmartPy[9] for Tezos mostly cover conventional unit testing that can be insufficient. The testing framework for LIGO[10] supports unit testing and mutation testing. However, none of the conventional testing frameworks offers a possibility for generating random traces and testing properties of interacting contracts. We will now focus on works using fuzzing/property-based testing techniques.

The closest to our work is the property-based testing framework for the Tezos' Michelson language. The framework utilises QCheck, a QuickCheck-inspired property-based testing framework for OCaml. QCheck was extended by Nomadic Labs with the ability to generate arbitrary sequences of Liquidity Baking contract calls. The contract is manually reimplemented in OCaml and serves as a model for the original contract. The model implementation is then validated against the original contract through the actual Tezos execution model. The development is tailored to the Liquidity Baking contract and is not connected to the Michelson formalisation in Coq Mi-Cho-Coq [4]. We are currently collaborating with the Mi-Cho-Coq team on integrating ConCert with the formalisation of Michelson.

For the Ethereum blockchain, several works are using randomised testing techniques. Echidna [7] and Brownie[11] use fuzzing-like techniques for testing smart contracts. The common challenge for this approach is that randomly generated transaction data might not be enough to ensure good coverage. This is especially problematic in the case of smart contract interactions, since the whole sequence (trace) of actions must be generated. Echidna uses coverage-driven feedback to automatically tune the testing parameters. Brownie uses unit-test like tests with user-defined generators for randomising inputs to contract calls in the tests. Brownie does not generate calls or execution traces, which limits the types of bugs that it can find. In our approach, instead of tuning pre-defined parameters, we allow users to define generators that produce random data with fewer discarded tests. For simple cases, data generators can be derived automatically using the QuickChick infrastructure.

---

[8] `https://trufflesuite.com/`
[9] `https://smartpy.io/docs/scenarios/testing/`
[10] `https://ligolang.org/docs/advanced/testing`
[11] Property-based testing framework for EVM: `https://github.com/eth-brownie/brownie`

The EthPloit project [10] generates possible exploits using fuzzing techniques. The exploits are split into three categories. For each of these categories, a special exploit detector oracle is used to report an exploit. For example, the Balance Increment oracle compares the overall initial balance of attackers' accounts with the current balance after a series of transfers and reports, if the balance of the attackers' accounts increases. EthPloit utilises static analysis to focus attention on particular variables and functions. The input for selected functions is generated randomly, or chosen using a seed set. The seed sets are used to provide runtime feedback. This improves the fuzzing efficiency by exploiting the results of previous runs. In our approach, the users specify the properties to test, instead of searching for particular categories of exploits. Violation of such properties is reported as a counterexample, which points to vulnerabilities. The pure/functional nature of our smart contracts avoids many pitfalls and simplifies reasoning about smart contracts. When compared to effectfull languages, such as Solidity, static analysis is less urgent.

Finally, the `cooked-validators` library[12] for the Plutus smart contract language [5] facilitates property-based testing with arbitrary transaction sequences. Note, however, that the execution model for Plutus does not involve on-chain inter-contract communication. Plutus itself supports property-based testing at the contract endpoint level using QuickCheck.[13]

7 Evaluation

We evaluate our framework in terms of usability, specifically regarding bug-finding capabilities. We demonstrated the testing framework on three concrete examples in the previous section, showing that it can find different types of real-world bugs. The vulnerabilities had a wide range of causes: the execution order, complex contract-to-contract interactions and the evolution of the contract state. Such bugs would not have been detected in other tools considering only functional correctness. This highlights ConCert's unique capability of modelling and testing complex contract interactions.

We have tested various other smart contracts, such as a reference implementation of the ERC-20 Token[14], and re-discovered known bugs, thus supporting the claim that our framework is effective at finding bugs. Since we have the full power of Coq at our disposal, we can effectively test any *decidable* property on the `Chain` type. Hence, there are few limitations in terms of expressiveness. While ConCert can find many common bug types, some bugs, such as vulnerabilities related to gas, remain out of reach of ConCert.

We also emphasise that once contracts are implemented (in ConCert) and the executable specifications are written (i.e. the decidable properties to be proven or tested), the only prerequisite for automatically testing the specifications is to implement the action generators and show instances, as discussed in Section 2.3. Implementing these requires only some expertise with Gallina and QuickChick, and can in some cases be derived automatically. Hence, the setup is relatively simple, only requiring moderate extra effort compared to writing traditional tests for users already familiar with property-based testing and Gallina or similar functional languages.

Since the contracts tested were ported to ConCert there is the risk that bugs were introduced in this process. However, since the framework gives full counterexamples it is easy to verify that bugs found are also present in the original contract, this part could

[12] https://iohk.io/en/blog/posts/2022/01/27/simple-property-based-tests-for-plutus-validators/
[13] https://plutus-pioneer-program.readthedocs.io/en/latest/pioneer/week8.html#using-quickcheck-with-plutus
[14] https://github.com/AU-COBRA/ConCert/tree/master/examples/eip20

also be automated. Another worry could be that the implementations of the contracts or the generators were tailored to finding known bugs. We took extra care implementing the generators for all three contracts, making sure that no knowledge of known bugs was used. Moreover, we did not test for a specific bug but for natural properties that would be part of any reasonable specification. For the Dexter and iToken contracts, we only implemented the entrypoints related to the known bugs. This slightly sped up finding the bugs, but adding the other entrypoints would only slow this down by a small constant factor. For the BAT contract the full contract was ported and it was not tailored towards any specific properties.

Additionally, the feedback loop from executing tests is fast, making it easy to use during the contract development process. In our experience, QuickChick will usually report counterexamples, if they exist, within 1-2 seconds and otherwise report that all inputs (by default 10.000 traces) passed – usually in 5-10 seconds (for traces of 14 calls). Of course, the time depends on many factors, most importantly, the length of traces and the complexity of generators and contracts. Heuristically, we limit ourselves to 10.000 tests, based on the extensive experience from QuickChick. Naturally, tests cannot fully guarantee that there is no bug, we use proofs for that goal.

8 Conclusions

We have presented the ConCert Coq framework for testing, verifying and extracting smart contracts. We have demonstrated the framework for property-based testing on three smart contracts using it to discover vulnerabilities used in previous attacks and new bugs that could have led to millions of dollars stolen or frozen. As stated in the previous section, the vulnerabilities had a wide range of causes covering the most common causes of flaws in smart contracts.

We have re-discovered several bugs in real-world contracts (not presented in this paper), such as the $50 million “DAO attack” on Ethereum, and tested reference implementations of ERC-20 and FA2 Token Standards, common standards for tokens used in several blockchains[15].

Hence, our approach to testing smart contracts scales to real-world contracts and is capable of finding significant bugs. Contracts in ConCert are extractable to Concordium’s Rust framework, Liquidity, and CameLIGO. Thus in total, we have a toolchain for producing executable code for smart contracts that are tested and verified. The importance of combined auditing, testing and verification is also starting to be recognized by the industry.[16]

References

1 Guillermo Angeris, Hsien-Tang Kao, Rei Chiang, Charlie Noyes, and Tarun Chitra. An Analysis of Uniswap markets. *Cryptoeconomic Systems*, 1(1), 2021. doi:10.21428/58320208.c9738e64.

2 Danil Annenkov, Mikkel Milo, Jakob Botsch Nielsen, and Bas Spitters. Extracting Smart Contracts Tested and Verified in Coq. In *CPP'2020*. Association for Computing Machinery, 2021. doi:10.1145/3437992.3439934.

3 Danil Annenkov, Jakob Botsch Nielsen, and Bas Spitters. ConCert: A Smart Contract Certification Framework in Coq. In *CPP'2020*, 2020. doi:10.1145/3372885.3373829.

4 Bruno Bernardo, Raphaël Cauderlier, Zhenlei Hu, Zhenlei Pesin, and Julien Tesson. Mi-Cho-Coq, a framework for certifying Tezos Smart Contracts. In *FMBC19*, 2019.

[15] https://github.com/AU-COBRA/ConCert

[16] e.g. https://forum.cardano.org/t/cip-proposal-cardano-audit-best-practice-guidelines/100022

5. James Chapman, Roman Kireev, Chad Nester, and Philip Wadler. System F in Agda, for fun and profit. In *MPC'19*, 2019. doi:10.1007/978-3-030-33636-3_10.
6. Kim Grauer, Will Kueshner, and Henry Updegrave. Chainalysis 2022 Crypto Crime Report. *Chainalysis 2022*, 2022. URL: https://go.chainalysis.com/2022-Crypto-Crime-Report.html.
7. Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. Echidna: effective, usable, and fast fuzzing for smart contracts. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 557–560, 2020. doi:10.1145/3395363.3404366.
8. Zoe Paraskevopoulou, Catalin Hritcu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin C. Pierce. Foundational property-based testing. In Christian Urban and Xingyuan Zhang, editors, *6th International Conference on Interactive Theorem Proving (ITP)*, volume 9236 of *Lecture Notes in Computer Science*, pages 325–343. Springer, 2015. doi:10.1007/978-3-319-22102-1_22.
9. Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq Project. *Journal of Automated Reasoning*, February 2020. doi:10.1007/s10817-019-09540-0.
10. Qingzhao Zhang, Yizhuo Wang, Juanru Li, and Siqi Ma. EthPloit: From Fuzzing to Efficient Exploit Generation against Smart Contracts. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020. doi:10.1109/SANER48275.2020.9054822.

Automatic Generation of Attacker Contracts in Solidity

Ignacio Ballesteros
Polytechnic University of Madrid, Spain

Clara Benac-Earle
Polytechnic University of Madrid, Spain

Luis Eduardo Bueso de Barrio
Polytechnic University of Madrid, Spain

Lars-Åke Fredlund
Polytechnic University of Madrid, Spain

Ángel Herranz
Polytechnic University of Madrid, Spain

Julio Mariño
Polytechnic University of Madrid, Spain

Abstract

Smart contracts on the Ethereum blockchain continue to suffer from well-published problems. A particular example is the well-known smart contract reentrancy vulnerability, which continues to be exploited. In this article, we present preliminary work on a method which, given a smart contract that may be vulnerable to such a reentrancy attack, proceeds to attempt to automatically derive an "attacker" contract which can be used to successfully attack the vulnerable contract. The method uses property-based testing to generate, semi-randomly, large numbers of potential attacker contracts, and then proceeds to check whether any of them is a successful attacker. The method is illustrated using a case study where an attack is derived for a vulnerable contract.

2012 ACM Subject Classification Software and its engineering → Software testing and debugging; Software and its engineering → Dynamic analysis; Software and its engineering → Empirical software validation

Keywords and phrases Property-Based Testing, Smart Contracts, Reentrancy Attack

Digital Object Identifier 10.4230/OASIcs.FMBC.2022.3

Funding This work has been partly funded under the grant S2018/TCS-4339 (BLOQUES-CM) co-funded by EIE Funds of the European Union and Comunidad de Madrid and by the Spanish MCI/AEI under grant ref. PID2019-104735RB-C44.

1 Introduction

The support of Smart Contracts introduced a key development in the Ethereum [3] blockchain platform since the first blockchain, Bitcoin [10], was originally proposed for cryptocurrency transfers. Smart contracts provided the opportunity to study the properties and security of code executed in blockchain platforms.

In the last few years, a variety of tools and frameworks to analyze and find vulnerabilities in blockchain smart contracts have been developed based on static and dynamic analysis. These tools are based on popular program testing techniques such as fuzz testing [9, 15], symbolic execution, taint tracking, and static analysis.

© Ignacio Ballesteros, Clara Benac-Earle, Luis Eduardo Bueso de Barrio, Lars-Åke Fredlund, Ángel Herranz, and Julio Mariño;
licensed under Creative Commons License CC-BY 4.0

A complementary testing technique is property-based testing (PBT) [4], a model-based testing technique. In PBT, tests are automatically generated from a model to cover a multitude of scenarios that a human tester may not have considered.

In [6], we presented Makina, a library and a domain specific language for writing PBT models for *stateful* programs. Models written in the new domain specific language are, using Elixir macros, rewritten into PBT state machines [1, 11]. Our main goal with Makina is to ease the task of developing correct and maintainable models, and to encourage model reuse. To meet these goals, Makina provides a declarative syntax for defining model states and model commands. In particular, Makina encourages the typing of specifications, and ensures through its rewrite rules that such type information can be used to effectively typecheck models. Moreover, to promote model reuse, the domain specific language provides constructs to permit models to be defined in terms of collections of previously defined models.

In (still unpublished) previous work, we have proposed a PBT model to test smart contracts. Such a model consists of a generic part, i.e., modelling concerns common to all smart contracts, and a specific part which is tailored to the specific behavior of each contract. As a case study, in this paper we extend the proposed PBT model and introduce automatic code generators to test a contract. The objective is obtaining a successful attacker contract on a contract vulnerable to the reentrancy attack.

A reentrancy attack involves two smart contracts: a victim contract and an untrusted attacker contract. A reentrancy attack occurs when a function from the victim contract makes a call to the attacker. In the Ethereum Virtual Machine (EVM), a reentrancy attack can happen also when a transfer to a contract is made. This transfer may end up in the execution attacker's code. The attacker takes advantage of this and tries to drain the victim's funds. One of the first (known) examples was the DAO attack which caused a loss of 60 million US dollars in June 2016.

The reentrancy attack is still an issue for Solidity smart contracts. Recent examples of reentrancy attacks are the 7.2 million dollar BurgerSwap hack (May 2021) caused by a fake token contract and a reentrancy exploit, and the 18.8 million dollar Cream Finance hack (August 2021) where the reentrancy vulnerability allowed the exploiter for a second borrow [12]. The reentrancy attack has been found in token standards as the ERC777 in the exploit of Uniswap[1]. This attack has been widely studied [2], classified[2] and is described in the official documentation of Solidity[3]. Some design patterns have been proposed to prevent this vulnerability, but the source of errors is tied to the language design.

To prevent a reentrancy attack in a Solidity smart contract one should (i) ensure all state changes happen before calling external contracts, i.e., update balances or code internally before calling external code, or (ii) use function modifiers that prevent reentrancy.

A way of testing that a contract is vulnerable to a reentrancy attack involves creating one attacker contract that exposes the problem. In this work, we use PBT to automatically generate such attacker contracts and test them against the given contract.

Additionally, many development and testing tools in the Ethereum platform offer ways of detecting contracts vulnerable to the reentrancy attack [8, 14, 16]. These tools are effective finding the vulnerability, but they are limited when providing an external test, for example in the form of an attacker contract, to independently check the vulnerability.

---

[1] `https://blog.openzeppelin.com/exploiting-uniswap-from-reentrancy-to-actual-profit/`
[2] `https://swcregistry.io/docs/SWC-107`
[3] `https://docs.soliditylang.org/en/v0.8.11/security-considerations.html#re-entrancy`

The rest of the paper is organized as follows. In Section 2 we present a smart contract which is vulnerable to the reentrancy attack and that will be used as a case study. The ideas behind the generation of attacker contracts are outlined in Section 3. In Section 4 we explain in detail how property based testing works and how it is used to test stateful systems. Experimental results are presented in Section 5. Some related work is discussed in Section 6. Finally, conclusions, limitations and further work are discussed in Section 7.

2 Case study

In this section, we present a running example based on the well known reentrancy attack. The goal is to test whether a contract is vulnerable to such an attack by creating an attacker contract, that tries the attack on the given contract. As we shall see, the attacker contract can be generalized and, thus, reused for testing other contracts against the reentrancy attack.

2.1 The victim contract

The following contract represents a wallet vulnerable to reentrancy attacks. In this wallet multiple accounts can deposit and withdraw *ether*. The funds are private and one can only operate on its own balance.

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity ^0.8.15;
3
4  contract Wallet {
5    mapping(address => uint) private credit;
6
7    function donate() external payable {
8      require(msg.value > 0 wei);
9      credit[msg.sender] += msg.value;
10   }
11
12   function withdraw() external {
13     uint bal = credit[msg.sender];
14     require(bal > 0);
15
16     (bool sent, ) =
17       msg.sender.call{value: bal}("");
18     require(sent, "Failed to send Ether");
19
20     credit[msg.sender] = 0;
21   }
22 }
```

Listing 1 Wallet contract (victim)

The `Wallet` contract has a `private` attribute named `credit`, which is a table storing the balance for each account. The contract exposes two public methods: `donate` and `withdraw`, and they can be called by any account or any other contract.

The method `donate` is labeled as `payable`, indicating that when this method is called, it can be done with some *ether* value attached. The *ether* value can be seen in the variable `msg.value`. In this case, the `Wallet` contract requires it to be greater than 0. The variable `msg.sender` has the address of who (account or contract) called to this method. The `Wallet` contract proceeds to register and link the `ether` amount and the sender into the `credit` variable.

When the withdraw method is called, the Wallet contract transfers back the registered balance of the caller (msg.sender). The transfer is done using the call function with the corresponding *ether* attached. Finally, it updates the msg.sender balance to 0.

The reentrancy attack is based on the exploit of a particular behavior of EVM smart contracts, affecting to languages like Solidity. When a transfer is done to a contract, the receiver can execute some code during the transaction. In the case of a transfer between two contracts, this behavior is giving the control to the receiver during a function call.

Here, the goal of the reentrancy attack is to steal money from a victim contract by draining its funds. In the Wallet contract the reentrancy attack can be exploited because in the withdraw method, the sender balance is updated after the amount has been transferred (lines 20 and 17, respectively). During this transfer, the attacker calls again to withdraw. The balance has not been updated yet (line 20), so the conditions to make a new transfer are still met (lines 13 and 14). This loop of re-entrant calls could be executed until the Wallet transfers all of its funds.

In the following section, we explain the attacker contract and how it is able to exploit the attack on this Wallet contract.

2.2 The attacker contract

Solidity smart contracts can define a function to be executed when they receive a transfer. This function is named receive, and it is fundamental for the reentrancy attack. In this section we explain a contract exploiting the reentrancy attack in the Wallet contract of the previous section.

In the following Solidity code, a contract attacker is presented. The entry point is the attack function. This function makes a first payment to the victim contract and then calls the withdraw function which will transfer ether back to the attacker. This transfer will trigger the receive function and before the receive terminates, the attacker calls again the withdraw function. That is, the attacker contract consists of two phases interacting with the victim contract, first the trigger phase, and second, the receive phase to continue draining the victims funds.

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity >=0.4.22 <0.9.0;
3
4  contract Attacker {
5    address private victim;
6
7    constructor(address victim_) {
8      victim = victim_;
9    }
10
11   receive() external payable {
12     uint victimBalance =
13       address(victim).balance;
14     bool has_funds =
15       victimBalance >= msg.value;
16
17     if (has_funds) {
18       try victim.withdraw() {} catch {}
19     }
20   }
21
22   function attack() external payable {
```

```
23      require(msg.value >= 1 wei);
24      uint init = address(this).balance;
25
26      victim.donate{value: msg.value}();
27      victim.withdraw();
28
29      uint end = address(this).balance;
30      emit SuccessfullAttack(end > init);
31    }
32  }
```

Listing 2 Attacker contract

The `attack` function emits an event `SuccessfulAttack(true)` with the result of the attack. We will use this event to test whether the attack was successful or not.

3 Automatic generation of attacker contracts

The goal is to automate the generation of a successful attacker contract. Our approach uses Property-based testing (PBT) to automatically generate random calls to the public functions obtained from the Application Binary Interface (ABI) of the victim contract with the goal of detecting a reentrancy vulnerability. If a vulnerability is found, the result is an attacker contract that can exploit the aforesaid vulnerability.

As we have seen in the previous section, in the case of the reentrancy attack, there are two phases in the interactions between the attacker and the victim: (i) the `trigger` phase that prepares the attack, and (ii) the `receive` phase where the attack takes place. These two phases correspond to two functions in the generated attacker contract: the `trigger` function and the `receive` function.

The trigger function contains a sequence of calls to the victim contract which ends with a call that *triggers* a transfer from the victim to the attacker. Then, this transfer invokes the `receive` function. In the `receive` function, there is a sequence of at least one call to reenter into the victim contract. Our approach is to automatically find such sequences, if they exist for the victim contract, using PBT.

The following code is the template of the attacker contract where the sequence of calls of the trigger and receive functions are generated using PBT. In the next section, we explain how they are automatically generated.

```
1  function trigger_sequence()
2    { /* To be generated using PBT */ }
3  function receive_sequence()
4    { /* To be generated using PBT */ }
5
6  function attack() external payable {
7    ...
8    trigger_sequence();
9    ...
10 }
11
12 receive() {
13   ...
14   if (has_funds) {
15     try receive_sequence() {} catch {}
16   }
17 }
```

Listing 3 Attacker contract template

For clarity, only the relevant code is displayed. In this code, line 8 corresponds to lines 26 and 27 of the attacker contract shown in Section 2.2. Note that this template contains the code that is common to many attacker contracts for the reentrancy attack, while the generated code is specific to each victim contract.

4 Property-based testing

Property-based testing (PBT) is a testing methodology which focuses on generating, automatically, test cases from a formal description of the behavior of the system under test. That is, PBT can be considered a form of model-based testing.

Formally, the model is an extended finite-state machine where one has to define the *state*, *commands*, *preconditions* and *postconditions*. PBT tools, for example, [1], will generate execution scenarios that will be run on the model and on the system under test to prove that the actual system behaves like the model. To generate the execution scenarios, *generators* are used.

4.1 Generators

A generator is capable of generating an infinite number of values of some data type, according to a probability distribution. PBT tools come equipped with a library of standard generators, for example, a generator for integer numbers.

In this work, we have defined a generator for non-negative integers to represent the ether used in payments. This generator is used as an argument for all transactions, for example in the calls to the *payable* functions or when deploying a contract.

A user of a PBT tool can also implement custom generators. In this work, we have defined a generator for attacker contracts.

4.1.1 Attacker contract generator

The attacker contract generator uses the template described in Section 3 and the ABI of the victim contract to generate a Solidity contract. The ABI is generated when a contract is compiled and contains the specification of the contract's functions. From the specification, one can extract the name, arguments and modifiers.

The attacker contract generator uses generators of values and function calls. These generators are fed with the information extracted from the ABI to obtain targeted function calls with valid values.

To generate an attacker contract, two sequence of calls are generated and inserted in the template: the trigger and the receive sequence. These sequences contain calls to the victim contract. A call is generated from the ABI, based on its specification. When a call to a contract needs to be generated, it chooses one of the defined function in the ABI. If the function has any arguments or modifiers, for example: if it is `payable`, an *ether* amount is generated for the call. Figure 1 is a diagram of this described process.

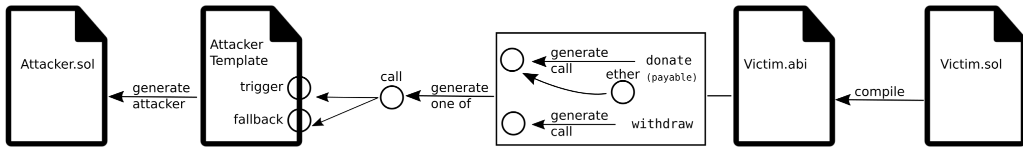


Figure 1 Attacker contract generator.

The following represents an example of generated code in the `Attacker` for the trigger and receive sequences in the `Wallet` example.

```
function trigger_sequence() payable {
  victim.donate{value: 3}();
  victim.withdraw();
  victim.donate{value: 1}();
}

function receive_sequence() payable {
  victim.withdraw();
  victim.donate{value: 1}();
  victim.withdraw();
  victim.donate{value: 2}();
  victim.withdraw();
  victim.withdraw();
}
```

Listing 4 Generated attacker contract sequences.

In this example, the generator has filled the two sequences: the `trigger_sequence` and the `receive_sequence`. This code fragment belongs to a generated contract based on the template in Listing 3. For example, the `trigger_sequence` includes 3 generated function calls: 2 `donate` calls and 1 `withdraw`. The function `donate` is `payable`, which means that it can receive `ether` attached, and the calls to this function include a generated `ether` value as argument.

4.1.2 Function call generator

The generation of function calls is driven by the specification found in the ABI. In the ABI, a function definition contains the name of the function, the number of arguments and their types and any other modifier. In the example of the `donate` function, Listing 5 represents the object definition extracted from the ABI.

```
{
  function: "donate",
  method_id: "ed88c68e",
  input_names: [],
  types: [],
  returns: [],
  type: :function,
  state_mutability: :payable,
  inputs_indexed: []
}
```

Listing 5 ABI definition of function `donate`.

Given the ABI definition of a function, the contract call generator outputs a new object with generated values. As the `donate` call is `payable`, an *ether* value is generated. Listing 6 contains the generated object for a single function call. Then, this generated call is encoded as Solidity code: `donate{value: 3}();`

```
{
  name: "donate",
  args: [],
  state_mutability: :payable,
  value: 3
}
```

Listing 6 ABI definition of function `donate`.

The contract call generator is able to generate values for each given argument of a function. The `donate` function does not have any additional argument, but given list of types for each argument, it generates a list of values for each corresponding argument. This generator is explained in the next subsection.

4.1.3 Value generator with context awareness

A valid Solidity value can be generated given a type. Given an `int`, a number in the range of the integers is generated. For `uint`, only non-negative integers are possible. This mapping between types and possible values is done for each Solidity type except `function`, `fixedNxM` and `ufixedNxM`.

There are limitations with random generated values. A relevant case is `address` values. A generated reference for an address will be syntactically valid, but it is unlikely to represent a populated address in the blockchain. This behavior is desired to test some cases, but the generation will be biased towards testing non-existent addresses. To fix this limitation, one could include in the address generator the possibility of generating addresses from a set of already known values.

A context in the generation of values carries extra information about values to be generated, for example: set of possible addresses, limits on the amount of *ether*.

4.2 Commands

In most cases, a test case is not a simple call to a function, but rather a sequence of calls to operations or methods interacting with a system. In our case, the blockchain and the smart contracts it contains. That is, a test case generator returns sequences of API calls, and the test property checks whether the execution of such a sequence of API calls is correct. To interact with the Ethereum blockchain we have defined commands that will be executed during the tests, checking the expected result after each call. In the following subsections, we describe the specific commands interacting with the blockchain that are relevant to understand how the test for reentrancy attack is executed.

4.2.1 Register Attacker

The Register command is responsible for compiling the generated attacker. The input of this command is the source code of an attacker contract, generated by the attacker contract generator, described in Section 4.1.1 using the known victim's ABI. The command compiles the generated attacker source code and registers the contract. This registration is required to later deploy instances of the attacker with the *Deploy* command. After the execution of this command we will have a registered attacker.

4.2.2 Deploy

This command will deploy a given contract with the arguments of the contract's constructor. We use this command with the victim contract and with the attacker. Once the contract is registered with the contract manager library, the command deploy can be executed. The result of this command is a transaction hash returned by the blockchain, ensuring that the request was received but might not be processed yet. The address where the contract was deployed is known once the transaction is completed.

The `constructor` of the attacker contract requires the address of the victim as an argument (see line 7 of Listing 2). If the victim contract has already been deployed, the tool is able to select the address as the argument for the `constructor`.

4.2.3 Attack

The Attack command triggers the execution of the attack. It calls to the method `attack` in the attacker contract. The condition to be able to execute this command is having an attacker deployed and its address known. We expect a transaction hash returned, to later check the transaction completion status. As the `attack` function is `payable`, this command has the possibility to call the method with an *ether* amount. The PBT tool generates this value with an *ether* value generator. The `attack` function in the smart contract emits an event with the result of the test. This Attack command also specifies that we expect that event with a result of *"not succeeding"*. This event is checked later with the *Get Events* commands.

4.2.4 Get Events

Given a transaction hash and if the transaction is complete, this command fetches and decodes any known events emitted. We know what events we expect from the transaction because any transaction command can add the expected result. For example, the *Attack* command included the result event as the one expected. At this point, we can have all the information to test the success of the attack

4.3 Shrinking

As a final ingredient in PBT implementations, there is an attempt to derive an easy-to-understand counter example through a procedure called *shrinking* which systematically tries to simplify counter examples, in order to ease the (manual) analysis required to discover the cause of the detected error. Shrinking is linked to generators. For instance, shrinking uses 0 as a simpler value for the int generator.

In our approach, the attacker contract is automatically obtained from shrinking, as it is explained in the following section.

5 Results

To test the Wallet contract presented in Section 2, we have used *Makina* [5], an Elixir DSL for writing stateful PBT models compatible with Quviq's Erlang QuickCheck [1] and PropEr/PropCheck [11]. To replicate the blockchain ecosystem, we use `Ganache`[4] as a local development and testing environment.

[4] `https://trufflesuite.com/ganache/`

The objective of the test is finding a successful attacker contract. To do this, the test deploys a private blockchain environment in which the commands are executed. The test fails when the expected result after the execution of the command is different from the behavior of the blockchain system. That is, we are comparing the results of executing commands in the model of the blockchain with those in the real system. Concretely, if the `attack` function successfully emits an event `SuccessfulAttack(true)`, then the test has found an attacker exploiting the reentrancy vulnerability in the victim contract.

In the case of the Wallet example, when the test is run, we obtain the following output.

```
Failed!

expected: SuccessfulAttack(false)
obtained: SuccessfulAttack(true)
```

That is, a test has found a successful attack event after the execution of an attack. The result of the test includes a counterexample, which can be used to reproduce the test case to manually check the error and study the source of the vulnerability. The provided counterexample includes: (i) an attacker contract that is capable of exploiting the reentrancy attack vulnerability, and (ii) a sequence of interactions with the blockchain to make the attack possible.

PBT tools try to shrink the counterexample to produce a simpler and smaller attacker contract, and to reduce the number of interactions with the blockchain to reproduce the attack. Note that the shrinking process does not guarantee that the reduced counterexample is the minimal case. Therefore, different runs of the test usually lead to different counterexamples.

In the following, we present a reduced counterexample of the generated attacker contract sequences i.e., after shrinking. The original counterexample had 22 calls to the victim contract in the trigger sequence, 10 to `withdraw` and 12 to `donate`, and 32 calls in the attack sequence, 13 to `withdraw` and 19 to `donate`.

```
1 function trigger_seq() public payable {
2   target.donate{value: 1}();
3   target.withdraw();
4 }
5
6 function receive_seq() public payable {
7   target.withdraw();
8 }
```

Listing 7 Generated and reduced attacker contract sequences.

The reduced counterexample contains the same calls that were manually written in the first Attacker contract shown as an attacker example (Listing 2).

The second part of the counterexample is the sequence of commands interacting with the blockchain. Initially, the sequence of commands had 113 steps. This does not necessary mean that all those commands are executed, on the contrary, it means that an error was found at some point while executing those commands. The shrinking is applied here to find a smaller counterexample. The PBT tool manages to shrink from 113 commands to a sequence of 15 commands. From this, the relevant commands that are required to prepare the blockchain into a state where the attack can be made are the following:

```
1 victim_address = deploy(Victim, value: 0)
2 register_attacker(Attacker, source_code)
3 deploy(Attacker, victim_address, value: 0)
4
5 call(victim, :donate, value: 1)
6 attack_hash = call(attacker, :attack, value: 2)
7 get_events(attack_hash)
```

Listing 8 Generated and reduced commands sequences.

The first three commands deploy the contracts required for the test. The `value` attribute represents the amount of initial *ether* that is transfer to the contract. The register command compiles the source code generated with the attacker contract generator and registers into the tool with the name `Attacker`. The deployment of the attacker command uses the address of the victim (`var_victim`) as argument to the constructor of the `Attacker`.

The next three lines contain calls generated interacting with the contracts. Notice how there is a command calling the method `donate` of the victim contract (`var_victim`) with an amount of `value` of 1 *ether*. This donation is fundamental to perform the attack, as the attacker contract tries to steal preexisting funds. These funds come from a known account that we have omitted for simplicity.

The time spent to obtain this result was 2 minutes and 30 seconds until finding a successful attacker contract. Then, it took 1 minute to shrink the counterexample. More than a hundred tests where generated, each one of them with a fresh Ganache environment. The tests ran on a laptop with an 11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz and 16GB of RAM.

6 Related Work

There has been a great interest on detection and prevention of reentrancy attacks after the DAO incident. The DAO attack itself and reentrancy vulnerabilities are documented in [2], which presents a whole catalog of vulnerabilities for Ethereum contracts. A survey of tools for analyzing Ethereum smart contracts is presented in [7]. Specifically for reentrancy vulnerabilities, [13] proposes to take advantage of the similar code structures found in these attacks. Structural analysis is performed in order to detect suspicious code patterns. After that, a dynamic analysis is performed in order to rule out false positives.

Several testing tools and frameworks for smart contracts have also been proposed, some of them combining techniques such as *fuzzing* or *genetic algorithms*. Dapp tools[5] is a kit including `hevm`[6], a fuzzing tool which also uses symbolic execution to falsify assertions in contracts. Foundry is an alternative collection of development tools, providing a testing framework, `forge`[7], with fuzzing capabilities. Some testing tools worth mentioning are CONTRACTFUZZER [8] and ECHIDNA [14]. Each of these tools require different information from the users – models, invariants, etc. – and also provide diverse outputs – some of them produce complete test suites, other just diagnostics on existing suites, etc. Our approach is original in the sense that is mostly automatic and provides a reusable, somehow minimal attacker contract.

Our work shares with CONTRACTFUZZER [8] a similar approach reading the Application Binary Inteface (ABI) to extract all the functions calls to a contract. This is a key component for the automation of the tools. The CONTRACTFUZZER tool has an `Agent` contract, which

[5] https://dapp.tools/
[6] https://github.com/dapphub/dapptools/tree/master/src/hevm
[7] https://github.com/foundry-rs/foundry/tree/master/forge

plays a similar role as our template `Attacker` contract when checking for reentrancy attacks. The existence of this contract is required by the nature of the attack, because it depends on the interaction of two contracts. The `Agent` auxiliary contract is designed as a link to execute calls to any contract. Our template is oriented to code generation, with the objective of building a self-contained contract. While CONTRACTFUZZER is oriented to be an analysis tool, we present a testing approach to be integrated with the development of smart contracts. We aim to provide useful and reduced counterexamples to understand the origin of the error and how to reproduce it, not only the diagnosis of the error.

We pursue similar objective as the ECHIDNA [14] tool. We both want to provide a tool where a minimal interaction with the user is possible. This requires analysis of the ABI for the automatic generation of calls. ECHIDNA focuses on detecting assertion violations and custom properties written in Solidity. To test them, ECHIDNA generates a sequence of calls to a contract to reach those conditions, shrinking them when an error is found. The main difference with our tool is that we can automatically test interactions between contracts.

Our proposal fits in the space between some functionalities of CONTRACTFUZZER and ECHIDNA, where we are able to automatically generate calls to a given contract but also provide meaningful counterexamples. Another advantage of our approach is that PBT is a general testing technique, that is, PBT testers do not need to learn a specific tool for testing smart contracts.

7 Conclusions and future work

In this paper, we present an approach to automatically detect vulnerabilities in Solidity smart contracts. Given a potential victim, the result of our approach is an attacker contract that exposes a vulnerability. We illustrate the approach in the case of the well known reentrancy vulnerability but our method that can be applied to other vulnerabilities in smart contracts.

Our method consists of identifying the common structure of an attacker. This structure is prepared as a template for automatically generating attackers given a victim contract. This generation is made using PBT tools, in which we have defined a custom generator for contracts of this type of attack. Using a model of a blockhain, we check the execution correctness by comparing it with the behavior of a running blockchain system.

The method proposed is able to identify the vulnerability in a given vulnerable contract. We provide as a result an instance of an attacker contract, along with all the steps required to replicate the attack.

The scope of this paper is limited to the identification of the reentrancy vulnerability in the `Wallet` example. The future work will include doing a study on the effectiveness of this method with a set of real world contracts. With this study, we will be able to compare the success of this method with similar tools detecting the reentrancy attack vulnerability.

The method explained it is not only aimed to identify the reentrancy attack vulnerability. In future work, we plan to apply this method on a broader range of attacks and study the effectiveness to detect other properties of smart contracts.

In this work, we have illustrated the approach with a simple example of a contract, the `Wallet` example. The public interface of the `Wallet` contract is fairly simple, and we did not have to elaborate complex interactions in the generated code. We plan to extend the context aware generator to, for example, use the return value from generated calls as arguments in the following calls. By doing that, we will be able to study the capability of our approach to generate more complex contracts and attacks.

The reentrancy attack is present in a wide variety of contracts. Token standards as the ERC777, mentioned in Section 1, can be susceptible to reentrancy attacks. As future work, the presented method could be studied to be adapted to this common interface contracts. The definition of specific PBT commands following the properties of the standard could be checked with generated contracts trying to break these conditions. The reentrancy condition arises as another property to be checked for these contracts. The general effectiveness of the vulnerability identification is future work with a corpus of contracts to test.

We have presented a testing approach that tests a pair of contracts: a victim and an attacker. Some interesting behaviors and bugs in our victim contracts may arise when multiple contracts operates and interact between them. In the future work, we plan to test models of more than two contracts interacting.

An additional detail is the refinement of the success condition in a reentrancy attack. Using the condition of a detected repeated call hook in the attacker contract along with detecting profits in an exploit.

References

1. Thomas Arts, John Hughes, Joakim Johansson, and Ulf T. Wiger. Testing telecoms software with Quviq QuickCheck. In Marc Feeley and Philip W. Trinder, editors, *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang, Portland, Oregon, USA, September 16, 2006*, pages 2–10. ACM, 2006. doi:10.1145/1159789.1159792.
2. Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on Ethereum smart contracts (SoK). In *Proceedings of the 6th International Conference on Principles of Security and Trust – Volume 10204*, pages 164–186, Berlin, Heidelberg, 2017. Springer-Verlag. doi:10.1007/978-3-662-54455-6_8.
3. Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform. *white paper*, 2013. URL: http://ethereum.org/ethereum.html.
4. Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM. doi:10.1145/351240.351266.
5. Luis Eduardo Bueso de Barrio, Lars-Ake Fredlund, Ángel Herranz, Clara Benac Earle, and Julio Mariño. Makina: A new quickcheck state machine library. In *Proceedings of the 20th ACM SIGPLAN International Workshop on Erlang*, Erlang 2021, pages 41–53, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3471871.3472964.
6. Luis Eduardo Bueso de Barrio, Lars-Åke Fredlund, Ángel Herranz, Clara Benac Earle, and Julio Mariño. Makina: a new quickcheck state machine library. In *Proceedings of the 20th ACM SIGPLAN International Workshop on Erlang*, pages 41–53, 2021.
7. Monika di Angelo and Gernot Salzer. A survey of tools for analyzing Ethereum smart contracts. *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*, pages 69–78, 2019.
8. Bo Jiang, Ye Liu, and W. K. Chan. ContractFuzzer: fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, September 2018. doi:10.1145/3238147.3238177.
9. Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33(12):32–44, 1990. doi:10.1145/96267.96279.
10. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *white paper*, May 2009. URL: https://bitcoin.org/bitcoin.pdf.
11. Manolis Papadakis and Konstantinos Sagonas. A proper integration of types and function specifications with property-based testing. In *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang*, Erlang '11, pages 39–50, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/2034654.2034663.
12. Kamil Polak. https://hackernoon.com/hack-solidity-reentrancy-attack, January 2022.

13 Noama Fatima Samreen and Manar H. Alalfi. Reentrancy vulnerability identification in Ethereum smart contracts. In *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, February 2020. doi:10.1109/iwbose50093.2020.9050260.

14 J. Smith. Echidna, a smart fuzzer for Ethereum, 2018.

15 Ari Takanen, Jared D. Demott, and Charles Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, Inc., Norwood, MA, USA, 2nd edition, 2018.

16 Valentin Wüstholz and Maria Christakis. *Harvey: A Greybox Fuzzer for Smart Contracts*, pages 1398–1409. Association for Computing Machinery, New York, NY, USA, 2020. doi:10.1145/3368089.3417064.

Proofgold: Blockchain for Formal Methods

Chad E. Brown
Czech Technical University in Prague, Czech Republic

Cezary Kaliszyk
Universität Innsbruck, Austria

Thibault Gauthier
Czech Technical University in Prague, Czech Republic

Josef Urban
Czech Technical University in Prague, Czech Republic

Abstract

Proofgold is a peer to peer cryptocurrency making use of formal logic. Users can publish theories and then develop a theory by publishing documents with definitions, conjectures and proofs. The blockchain records the theories and their state of development (e.g., which theorems have been proven and when). Two of the main theories are a form of classical set theory (for formalizing mathematics) and an intuitionistic theory of higher-order abstract syntax (for reasoning about syntax with binders). We have also significantly modified the open source Proofgold Core client software to create a faster, more stable and more efficient client, Proofgold Lava. Two important changes are the cryptography code and the database code, and we discuss these improvements. We also discuss how the Proofgold network can be used to support large formalization efforts.

2012 ACM Subject Classification Theory of computation → Automated reasoning

Keywords and phrases Formal logic, Blockchain, Proofgold

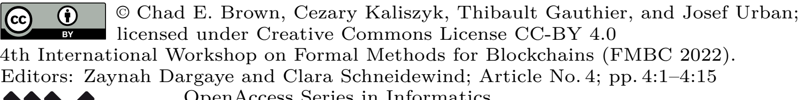
Digital Object Identifier 10.4230/OASIcs.FMBC.2022.4

Funding The results were supported by the Ministry of Education, Youth and Sports within the dedicated program ERC CZ under the project POSTMAN no. LL1902, the ERC starting grant no. 714034 SMART, the Czech Technical University Global Postdoc Fellowship, the European Regional Development Fund under the Czech project AI&Reasoning no. CZ.02.1.01/0.0/0.0/15_003/0000466, and Amazon Research Awards.

1 Introduction

Proofgold is a cryptocurrency network with support for formal logic and mathematics. An initial version of the Proofgold Core software was anonymously announced on June 8, 2020, via a memo.cash account.[1] The software and a discussion forum was available at proofgold.org until December 2021, at which time proofgold.org became unreachable.[2] During these first 18 months of Proofgold's existence the authors of the present paper experimented with the system, including publishing a number of formal theories and developments into Proofgold's blockchain. In the course of these experiments it became clear that the Proofgold Core software was slow and unstable. As a consequence the authors have created an alternative client.[3] Since Proofgold is relatively new and not well known we will need to describe Proofgold in general in order to put our work into an understandable context. We will make explicit what is our work and what is preexisting work.

[1] `https://memo.cash/profile/1NzEUQWpb5Mze9REfkVAZ8wDcxzqpZFNJ8`
[2] An archive of proofgold.org from December 2021, including the last release of the Proofgold Core software, is available at `https://prfgld.github.io/`.
[3] `http://proofgold.net/`



© Chad E. Brown, Cezary Kaliszyk, Thibault Gauthier, and Josef Urban;
licensed under Creative Commons License CC-BY 4.0

4th International Workshop on Formal Methods for Blockchains (FMBC 2022).
Editors: Zaynah Dargaye and Clara Schneidewind; Article No. 4; pp. 4:1–4:15
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

We give an introduction of Proofgold in Section 2. We present the intuitionistic higher-order logic at the kernel of Proofgold in Section 3. Several Proofgold theories are described in Section 4, all but the first of which was published into the Proofgold chain by one of the present authors. We describe the Proofgold Lava Client in Section 5, a significantly faster and more stable client primarily developed by one of the present authors. A HOL4 interface developed by one of the present authors for proving theorems with bounties is described in Section 6. A description of the bounty system, current bounties and possible future use of bounties is explored in Section 7. We conclude by considering related work in Section 8.

2 Introduction to Proofgold

At the core of Proofgold is a proof checker for intuitionistic higher-order logic with functional extensionality. On top of this framework users can publish *theories*. A theory consists of a finite number of primitive constants along with their types and a finite number of sentences as axioms. A theory is uniquely identified by its 256-bit identifier given by the Merkle root of the theory (seen as a tree). After a theory has been published, *documents* can be published in the theory. Documents can define new objects (using primitives or previously defined objects), prove new theorems and make new conjectures. When a theory is published, the axioms are associated with public keys which are marked as the *owners* of the propositions. Likewise, when a document proves a theorem within a theory, a public key (associated with the publisher of the document) is associated with the proven proposition. These are the only ways propositions can have declared owners. As a consequence, it is possible to determine if a proposition is known (either as an axiom or as a previously proven theorem) by checking if it has an owner. Ownership of propositions also gives a way of redeeming bounties by proving conjectures. A bounty can be placed on an unproven proposition where this bounty can only be spent by the owner of a proposition (or the owner of the negated proposition). By publishing a document resolving the conjecture, the bounty proposition (or its negation) will become owned by public keys associated with the publisher of the document. After this the bounty can be claimed. In order to prevent network participants from frontrunning proofs (i.e., "stealing" proofs from unconfirmed documents to unfairly claim bounties), a document can only be published (at which point proofs in the document are revealed) after a commitment has been published and sufficiently confirmed. All of the concepts above (theories, documents, owners, bounties and commitments) are inherited from the Qeditas code base and are described either in the Qeditas white paper [25] or Qeditas technical documentation [6].[4]

A major difference between Proofgold and Qeditas is the consensus mechanism. Qeditas had planned to be proof-of-stake, with the initial stake determined by a snapshot of the Bitcoin blockchain (i.e., an "airdrop"). Proofgold is a combination of proof-of-stake and proof-of-burn, where the proof-of-burn element involves burning small amounts of Litecoin. During the initial month of Proofgold (mostly in June 2020) before any participant had a stake, blocks could be created using proof-of-burn alone. Proofgold nodes must be run in combination with Litecoin nodes in order to verify proofs-of-burn. The Litecoin burn transactions also contain data committing to the previous Litecoin burn transaction and the id of new Proofgold block. Due to this, an outline of the Proofgold blockchain can be viewed from Litecoin. A benefit of this combination is that Proofgold's security model is able to

[4] A large part of Proofgold's code was inherited from the open source Qeditas project. More information about Qeditas is at `https://qeditas.org`. Qeditas appears to have never launched.

reuse the proof-of-work used to secure Litecoin. In particular, Litecoin's proof-of-work (as reflected by the block ids of Litecoin blocks) is used to determine the next staking modifier for a Proofgold block. The staking modifier is a large unpredictable number that is used to determine when the next opportunity each qualified Proofgold asset will have to stake. The connection with Litecoin reduces the risk of some well-known problems with proof-of-stake consensus mechanisms [17].

Another difference from Qeditas is that the first 5000 Proofgold blocks automatically put half the block reward as a bounty on pseudorandom propositions. This has the effect that as new people with low stake (very few Proofgold bars [5]) enter the system, they can quickly obtain a higher stake by taking the time to prove theorems with these bounties. The automatic placement of bounties has since ended, so that each block generates only a block reward of 25 bars to the staker of the block and all bounties are placed intentionally by a network participant.

The primary logical difference between Qeditas and Proofgold is that the logic underlying Qeditas included type variables, where the logic underlying Proofgold does not.

The elements of Proofgold described above are the work of Proofgold Core developers who were previously accessible via the proofgold.org forum, with some of the present authors sometimes giving feedback via the forum.

3 Intuitionistic Higher Order Logic

We briefly describe a formulation of intuitionistic higher order logic (IHOL). We begin with a set $\mathcal{T}$ of *simple types*. One base type o is the type of propositions. In general Proofgold allows finitely many other base types, but we will only consider cases with one other base type ι. All other types are $\alpha\beta$, meaning the type of functions from α to β. Some authors write this as $\beta\alpha$ (following Church [4]) or $\alpha \to \beta$ (especially in the presence of other type constructors).

We next define a family of simply typed terms. For each type $\alpha \in \mathcal{T}$, let $\mathcal{V}_\alpha$ be a countably infinite set of variables of type α. Let $\mathcal{C}$ be a finite set of typed constants. We define a set Λ_α of *terms of type* α as follows: For each variable x of type α, $x \in \Lambda_\alpha$. For each constant c of type α, $c \in \Lambda_\alpha$. If $s \in \Lambda_{\alpha\beta}$ and $t \in \Lambda_\alpha$, then $(s\ t) \in \Lambda_\beta$. If x is a variable of type α and $s \in \Lambda_\beta$, then $(\lambda x.s) \in \Lambda_{\alpha\beta}$. If $s, t \in \Lambda_o$, then $(s \to t) \in \Lambda_o$. If x is a variable of type α and $s \in \Lambda_o$, then $(\forall x.s) \in \Lambda_o$. Note that Λ_α also depends on the set $\mathcal{C}$, but this set will be fixed in each theory.

We use common conventions to omit parentheses. We sometimes include annotations on λ and $\forall$ bound variables (e.g., $\lambda x : \alpha.s$ and $\forall y : \beta.s$) to indicate the type of the variable. We define $\mathcal{F}s$ to be the free variables of s and for sets A of terms we define $\mathcal{F}A$ to be $\bigcup_{s\in A} \mathcal{F}s$. We assume a capture avoiding substitution s^x_t is defined. Terms of type o are called *propositions*. A *sentence* is a proposition with no free variables.

The only built-in logical connective is implication ($\to$) and the only built-in quantifier is the universal quantifier ($\forall$). In the context of higher-order logic it is well-known how to define the remaining logical constructs in a way that respects their intuitionistic meaning. In each case we use an impredicative definition that traces its roots to Russell [20] and Prawitz [18]. We define $\bot$ to be the proposition $\forall p : o.p$ where x is a variable of type o. We write $\neg s$ for $s \to \bot$. We define $\wedge$ to be $\lambda qr : o.\forall p : o.(q \to r \to p) \to p$ and write $s \wedge t$ for $(\wedge s)t$. We

[5] The common Proofgold currency token is called "bars" which are made up of 100 billion Proofgold "atoms."

$$\frac{}{\Gamma \vdash s}\; s \in \mathcal{A} \qquad \frac{}{\Gamma \vdash s}\; s \in \Gamma \qquad \frac{\Gamma \vdash s}{\Gamma \vdash t}\; s{\approx}t \qquad \frac{\Gamma, s \vdash t}{\Gamma \vdash s \to t} \qquad \frac{\Gamma \vdash s \to t \qquad \Gamma \vdash s}{\Gamma \vdash t}$$

$$\frac{\Gamma \vdash s}{\Gamma \vdash \forall x.s}\; x \in \mathcal{V}_\alpha \setminus \mathcal{F}\Gamma \qquad \frac{\Gamma \vdash \forall x.s}{\Gamma \vdash s^x_t}\; x \in \mathcal{V}_\alpha, t \in \Lambda_\alpha$$

$$\frac{\Gamma \vdash sx = tx}{\Gamma \vdash s = t}\; x \in \mathcal{V}_\alpha \setminus (\mathcal{F}\Gamma \cup \mathcal{F}s \cup \mathcal{F}t) \text{ AND } s,t \in \Lambda_{\alpha\beta}$$

Figure 1 Proof Calculus for Intuitionistic HOL.

define $\vee$ to be $\lambda qr : o.\forall p : o.(q \to p) \to (r \to p) \to p$ and write $s \vee t$ for $(\vee s)t$. For each type α we use $\exists x : \alpha.s$ as notation for $\forall p : o.(\forall x : \alpha.s \to p) \to p$ where p is not x and is not free in s. For equality we write $s = t$ (where s and t are type α) as notation for $\forall p : \alpha\alpha o.pst \to pts$ where p is neither free in s nor t. This is a modification of Leibniz equality which we will call *symmetric Leibniz equality*.[6] We write $s \neq t$ to mean $(s = t) \to \bot$. The $\beta\eta$-conversion relation $s{\approx}t$ is defined in the usual way.

Let $\mathcal{A}$ be a set of sentences intended to be axioms of a theory. A natural deduction system for intuitionistic higher-order logic with functional extensionality and axioms $\mathcal{A}$ is given by Figure 1. In particular the rules define when $\Gamma \vdash s$ holds where Γ is a finite set of propositions and s is a proposition. Aside from the treatment of functional extensionality, this is the same as the natural deduction calculus described in [3].

Adding Curry-Howard style checkable proof terms to such a calculus is well-understood and we do not dwell on this here [22]. Proofs published in Proofgold documents are given by such proof terms. There are two practical restrictions Proofgold places on proofs. One restriction is that proofs cannot be too big. A proof is part of a document, a document is published in a transaction and a transaction is published in a block. Proofgold has a block size limit of 500KB, so that proofs larger than 500KB (measured in Proofgold's binary format) cannot be published. If one has a proof larger than 500KB, then either one must find a smaller proof or separate the result into lemmas with smaller proofs published in separate documents (in separate blocks). Another restriction is that checking a proof is not allowed to be too hard. In an extreme case, checking a proof could require β-normalizing a term of size m to obtain a term of size $2^{2^{2^m}}$ (or even much larger). The Proofgold Core checker avoids such "poison proofs" by maintaining a counter that increments while a document is being checked. Each step of the computation increments the counter. For example, substituting t for a de Bruijn index x in a term $r\ x\ x$ increments the counter at least 5 times since there are two applications, two occurrences of x and one occurrence of r. Depending on the structure of r the counter may be incremented more. Also, in practice the substitution may be beneath a binder so that de Bruijn indices in t may need to be shifted. Such shifting increments the counter in a similar way. If the counter reaches a certain bound (150 million), then an exception is raised and the document is considered to be incorrect.

[6] This variant of equality was the choice of the initial Proofgold developers.

4 Proofgold Theories

A theory is determined by a finite number of typed primitives and axioms. Each theory is isolated from other theories. A drawback of this approach is there is no way to directly use results from one theory in another. A benefit is that if one theory turns out to be inconsistent, this will have no affect on other theories. Below we describe a number of Proofgold theories, where the authors are responsible for all but the first.

4.1 HF Theory

Proofgold has one built-in theory: a theory of hereditarily finite sets (HF). This theory was included by the initial Proofgold developers in order to have a language for generating potentially meaningful pseudorandom propositions for bounties given as half the block reward. There are many primitive constants, but only six do not have a defining equation: $\varepsilon : (\iota o)\iota$ (a "choice" operator), $\in: \iota\iota o$ (set membership), $\emptyset : \iota$ (the empty set, also the ordinal 0), $\bigcup : \iota\iota$ (the union operator), $\wp : \iota\iota$ (the power set operator) and $\mathsf{r} : \iota(\iota\iota)\iota$ (the replacement operator). For each of the above constants, there is at least one axiom giving a property the constant must satisfy. Additional axioms are a classical principle ($\forall p.\neg\neg p \to p$), set extensionality, an $\in$-induction principle and an induction principle implying all sets are hereditarily finite.

The theory additionally includes 97 constants with axioms giving a definitional equation for each constant. Examples include a constant indicating a set has exactly 5 elements, a constant indicating that an algebraic structure is a loop and a constant indicating that two untyped combinators (represented as sets) are equivalent under conversion. The HF theory was used to generate pseudorandom bounties for the first 5000 Proofgold blocks. These extra constants make it possible to easily generate sentences targeting certain classes.[7] The last of the pseudorandom bounties was automatically placed in December 2020. As of May 2022, 38% of the conjectures have been resolved and the bounties collected. The fact that 62% are still outstanding after 17 months is an indication of the difficulty of the problems.

4.2 Two HOTG Theories

There are two theories axiomatizing higher-order Tarski Grothendieck set theory (HOTG). These were both published into the Proofgold blockchain by the first author. The two theories follow the two formulations described in [3]. One is based on the Mizar formulation[8] and the other is based on the Egal formulation. Both theories are classical via the Diaconescu proof of excluded middle from choice (at ι) and set extensionality [19]. Most of the documents published into the Proofgold blockchain have been published in the HOTG-Egal theory. These documents target formalization of mathematics. A highlight is the construction of the real numbers via a representation of Conway's surreal numbers [5].

4.3 A Theory for HOAS

A different kind of theory published into the Proofgold blockchain is a theory for reasoning about syntax. Unlike the theories above, this theory does not imply classical principles. This theory was also published into the Proofgold blockchain by the first author. We describe it in

[7] More information can be found in http://grid01.ciirc.cvut.cz/~chad/pfghf.pdf.
[8] More information about the Mizar formulation of HOTG can be found in http://grid01.ciirc.cvut.cz/~chad/pfgmizar.pdf.

more detail here in order to make it clear Proofgold can be used for more than formalization of mathematics. In particular, Proofgold can be used to prove properties of programs with bound variables.

For our theory of syntax, we include one base type ι, two primitive constants $\mathsf{P} : \iota\iota\iota$ and $\mathsf{B} : (\iota\iota)\iota$ and four axioms:

- Pairing is injective: $\forall xyzw : \iota.\mathsf{P}xy = \mathsf{P}zw \to x = z \wedge y = w$.
- Binding is injective: $\forall fg : \iota\iota.\mathsf{B}f = \mathsf{B}g \to f = g$.
- Binding and pairing give distinct values: $\forall xy : \iota.\forall f : \iota\iota.\mathsf{P}xy \neq \mathsf{B}f$.
- Propositional extensionality: $\forall pq : o.(p \to q) \to (q \to p) \to p = q$.

The constant P is a generic pairing operation on syntax and B is a generic binding operation, allowing representation by higher-order abstract syntax (HOAS) [16].

We can embed many syntactic constructs into the theory by building on top of the basic pairing and binding operators. For example, we could embed untyped λ-calculus by taking P to represent application and B to represent λ-abstraction. Instead of adopting this simple approach, we will use tagged pairs when representing application and λ-abstraction, so that there will still be infinitely many pieces of syntax that do not represent untyped λ-terms. To do this we will need one tag, so let us define nil to be $\mathsf{B}(\lambda x.x)$. Now we can define $\mathsf{A} : \iota\iota\iota$ to be $\lambda xy : \iota.\mathsf{P}\ \mathsf{nil}\ (\mathsf{P}\ x\ y)$ and define $\mathsf{L} : (\iota\iota)\iota$ to be $\lambda f : \iota\iota.\mathsf{P}\ \mathsf{nil}\ (\mathsf{B}\ f)$. It is easy to prove A and L are both injective and give distinct values.

We can now impredicatively define the set of untyped λ-terms relative to a set $\mathcal{G}$ (intended to be the set of possible free variables) as follows. Let us write $(\mathcal{G}, x)$ for the term $\lambda y : \iota.\mathcal{G}\ y \vee y = x$. Here $\mathcal{G}$ has type ιo while x and y have type ι (and are different). We will define $\mathsf{Ter} : (\iota o)\iota o$ so that Ter is the least relation satisfying three conditions:

- $\forall \mathcal{G} : \iota o.\forall y : \iota.\mathcal{G}y \to \mathsf{Ter}\ \mathcal{G}\ y$,
- $\forall \mathcal{G} : \iota o.\forall f : \iota\iota.(\forall x : \iota.\mathsf{Ter}\ (\mathcal{G}, x)\ (fx)) \to \mathsf{Ter}\ \mathcal{G}\ (\mathsf{L}\ f)$ and
- $\forall \mathcal{G} : \iota o.\forall yz : \iota.\mathsf{Ter}\ \mathcal{G}\ y \to \mathsf{Ter}\ \mathcal{G}\ z \to \mathsf{Ter}\ \mathcal{G}\ (\mathsf{A}\ y\ z)$.

Technically, the impredicative definition of Ter is given as

$$\begin{array}{lll} \mathsf{Ter} & := & \lambda \mathcal{G} : \iota o.\lambda x : \iota.\forall p : (\iota o)\iota o.(\forall \mathcal{G} : \iota o.\forall y : \iota.\mathcal{G}y \to p\ \mathcal{G}\ y) \\ & & \to (\forall \mathcal{G} : \iota o.\forall f : \iota\iota.(\forall x : \iota.p\ (\mathcal{G}, x)\ (fx)) \to p\ \mathcal{G}\ (\mathsf{L}\ f)) \\ & & \to (\forall \mathcal{G} : \iota o.\forall yz : \iota.p\ \mathcal{G}\ y \to p\ \mathcal{G}\ z \to p\ \mathcal{G}\ (\mathsf{A}\ y\ z)) \to p\ \mathcal{G}\ x. \end{array}$$

We can similarly define one-step β-reduction (relative to a set of variables) as follows:

$$\begin{array}{lll} \mathsf{Beta}_1 & := & \lambda \mathcal{G} : \iota o.\lambda xy : \iota.\forall r : (\iota o)\iota\iota o. \\ & & (\forall \mathcal{G} : \iota o.\forall f : \iota\iota.\forall z.(\forall x.\mathsf{Ter}\ (\mathcal{G}, x)\ (fx)) \to \mathsf{Ter}\ \mathcal{G}z \to r\ \mathcal{G}\ (\mathsf{A}\ (\mathsf{L}\ f)\ z)\ (fz)) \\ & & \to (\forall \mathcal{G} : \iota o.\forall fg : \iota\iota.(\forall z.r\ (\mathcal{G}, z)\ (fz)(gz)) \to r\ \mathcal{G}\ (\mathsf{L}\ f)\ (\mathsf{L}\ g)) \\ & & \to (\forall \mathcal{G} : \iota o.\forall xyz.r\ \mathcal{G}\ x\ z \to \mathsf{Ter}\ \mathcal{G}y \to r\ \mathcal{G}\ (\mathsf{A}\ x\ y)\ (\mathsf{A}\ z\ y)) \\ & & \to (\forall \mathcal{G} : \iota o.\forall xyz.r\ \mathcal{G}\ y\ z \to \mathsf{Ter}\ \mathcal{G}x \to r\ \mathcal{G}\ (\mathsf{A}\ x\ y)\ (\mathsf{A}\ x\ z)) \to r\ \mathcal{G}\ x\ y. \end{array}$$

We can then define $\mathsf{BetaE}\ \mathcal{G}$ to be the least equivalence relation (relative to the domain $\mathsf{Ter}\ \mathcal{G}$) containing $\mathsf{Beta}_1\ \mathcal{G}$. We omit the details here.

These definitions give us sufficient material to make conjectures that ask for certain kinds of untyped λ-terms. Let $\emptyset$ be notation for the term $\lambda x : \iota.\bot$ (representing the empty set of variables). Consider the following sentences:

$$\exists F : \iota.\mathsf{Ter}\ \emptyset\ F \quad \wedge \quad \forall x : \iota.\mathsf{BetaE}\ (\emptyset, x)\ (\mathsf{A}\ F\ x)\ x \tag{1}$$

$$\exists Y : \iota.\mathsf{Ter}\ \emptyset\ Y \quad \wedge \quad \forall f : \iota.\mathsf{BetaE}\ (\emptyset, f)\ (\mathsf{A}\ Y\ f)\ (\mathsf{A}\ f\ (\mathsf{A}\ Y\ f)) \tag{2}$$

Sentence (1) asserts the existence of an identity combinator while sentence (2) asserts the existence of a fixed point combinator. In order to prove each sentence a combinator with the right property must be given as a witness and then be proven to have the property.[9]

As a demonstration, these sentences were published as conjectures (with bounties) in documents published into the Proofgold blockchain. The solutions were then published as two theorems (with proofs). The solutions contain the witnesses: $\mathsf{L}(\lambda x.x)$ for (1) and the famous Y-combinator $\mathsf{L}(\lambda f.\mathsf{A}(\mathsf{L}(\lambda x.\mathsf{A}\ f\ (\mathsf{A}\ x\ x)))(\mathsf{L}(\lambda x.\mathsf{A}\ f\ (\mathsf{A}\ x\ x))))$ for (2).

These simple examples suggest how Proofgold could be used to publish conjectures for verification conditions of programs or even conjectures asking for a program satisfying a specification. This could especially be useful when those programs are smart contracts.

5 Proofgold Lava Client

The existing client, Proofgold Core, has already included all the functionality needed to run the blockchain. However, certain parts of the implementation did not scale well. In particular as the number of proofs already in the blockchain grew operations such as synchronizing new clients or rechecking the blockchain became too costly. For these reasons we reimplemented parts of the client software and provide it as the Proofgold Lava Client and discuss the changes in this section. Proofgold Lava is primarily the work of the third author.

5.1 Database Layer

The Proofgold client software uses 19 databases. In the Core software they have been stored in 19 directories, each with an index file and and a data file. Lookups in this database, including locking, became a significant overhead for all Merkle tree operations. For this reason in the Lava implementation we switched to the standard Unix DBM interface, in particular using the GDBM library by default, which in addition to the already used operations provides atomic operations.

5.2 Cryptography Layer

Harrison has provided an efficient library[10] of field operations in the various cryptographic fields verified in the HOL Light theorem prover [10]. The library includes the Elliptic curve used by Bitcoin and Proofgold along with a number of other elliptic curves and operations provided for them [7]. In the Lava implementation we switched from the OCaml implementation of the cryptographic primitives to instead allow a low level efficient implementation. We provide the flexibility of switching between two implementations. First, we allow the use of the Bitcoin crypto implementation. It has been tested in Bitcoin and other cryptocurrencies, so it is likely to be correct. However, we also allow the use of the formally verified version (where the verified operations are the addition, multiplication, or inverse modulo in the field, but the verification of the actual additions and multiplication of points on the curve is still future work).

In addition to the much more efficient encryption and signing, we also switched to a low-level implementation of SHA256 used for hashing, including the recursive hashing of all sub-structures used in the Merkle tree. That last operation is used quite often, as all subterms used in proof terms are hashed this way.

[9] Note that in a classical calculus, it would be sufficient to prove such an existential statement by proving it is impossible for a witness not to exist.
[10] `https://github.com/awslabs/s2n-bignum`

5.3 Networking and Proofchecking Layers

The Lava client also includes a number of improvements to the networking layer and to proof checking. We have decided not to change the actual communication protocol between the nodes, but rather to improve the implementation. In particular, we have reduced the complexity of preparing block deltas and improved the efficiency of serialization. We have also replaced the implementation of the checker by a more efficient one. The new checker for the variant of simple type theory used in Proofgold includes perfect term sharing and preserves a number of invariants (e.g., $\beta\eta$-normal forms) is discussed elsewhere [2].

6 A HOL4 Interface for Mining Bounties from the HF theory

HOL4 [21] is an interactive theorem prover (ITP) for higher-order logic (HOL) that helps users to produce formal proofs and thus verify theorems. We are developing a HOL4 interface to Proofgold for two reasons. The first one is to enable people familiar with the HOL4 system to check and share their proofs in Proofgold. This way, HOL4 users would benefit from the additional features provided by Proofgold such as authorship recognition and the bounty system. The second one, which is the focus of this section, is to provide a way to manually or automatically prove bounties in HF. For this task, we chose HOL4 because it is equipped with powerful automation. The source code of this interface can be downloaded at `http://grid01.ciirc.cvut.cz/~thibault/h4pfg.tar.gz`. The HOL4 interface is primarily the work of the second author.

6.1 Importing the HF theory into HOL4

We import the 6 axioms and 97 definitions of the HF theory into HOL4. A translation between the two systems is straightforward since the logics of HOL4 and HF are similar and in particular the formula structures are almost identical. When reading a HF statement, the logical constants of the HF theory in Proofgold (e.g., $\wedge, \vee, \forall, \rightarrow, \ldots$) are mapped to their HOL4 native versions. For other HF constants (e.g., $\in, \subset, \mathit{exactly5}, \ldots$), new HOL4 constants are created. The same process is used to import HF bounties into HOL4.

6.2 Exporting HOL4 proofs to HF

To verify theorems proved in HOL4 with Proofgold, we first need to derive the HOL4 kernel rules from the IHOL rules and HF axioms. For instance, the HOL4 reflexivity rule can be derived from the IHOL rules in the following way:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{ptt \vdash ptt}}{\vdash ptt \rightarrow ptt}}{\vdash \forall p.ptt \rightarrow ptt}}{\vdash t = t}$$

Every HOL4 theorem is proved by composing applications of the HOL4 kernel inference rules. Therefore, to produce a HF proof, we trace these applications during the proof process and substitute them by their corresponding derivations in HF.

6.3 Proving Bounties

To reward the first users, a finite set of automatically generated bounties was included at the beginning of the Proofgold blockchain by the developers. The newer bounties proposed by developers and users are now usually based on textbook mathematical knowledge (often from

$$\dfrac{\dfrac{\dfrac{\text{Definition for } \subseteq}{\vdash (a \subseteq b) \leftrightarrow (\forall y. y \in a \rightarrow y \in b)}}{\vdash (t_0 \subseteq t_0) \leftrightarrow (\forall y. y \in t_0 \rightarrow y \in t_0)} \quad \dfrac{y \in t_0 \vdash y \in t_0}{\forall y. \vdash y \in t_0 \rightarrow y \in t_0}}{\vdash t_0 \subseteq t_0}$$

$$\dfrac{\dfrac{\dfrac{}{\vdash x_1 = x_1}}{\vdash q t_0 x_1 \rightarrow x_1 = x_1}}{\vdash \forall x_1. q t_0 x_1 \rightarrow x_1 = x_1}$$

$$\dfrac{\dfrac{\vdash t_0 \subseteq t_0 \qquad \vdash \forall x_1. q t_0 x_1 \rightarrow x_1 = x_1}{\vdash (t_0 \subseteq t_0) \wedge (\forall x_1. q t_0 x_1 \rightarrow x_1 = x_1)}}{\vdash \exists x_0. (x_0 \subseteq t_0) \wedge (\forall x_1. q x_0 x_1 \rightarrow x_1 = x_1)}$$

Figure 2 A HOL4 Proof of a HF Bounty.

interactive theorem provers) and are considerably harder than the automatically generated ones (see Section 7). We now show how to prove, using the HOL4 interface, some of the first "easy" bounties manually and automatically.

6.3.1 Manual Proof

The following auto-generated bounty has a relatively easy proof and therefore is one of the first we could manually prove:

$$\exists x_0. x_0 \subseteq t_0 \wedge \forall x_1. (\forall x_2. x_2 \subseteq x_1 \rightarrow \forall x_3 x_4. (\neg c_0 x_3 x_4 \wedge c_1 x_0 \wedge \neg c_2 x_2) \rightarrow c_3(c_4(c_5 x_0)) x_4) \rightarrow x_1 = x_1$$
where $t_0 = \wp(\wp(\wp(\wp\emptyset)))$ and $[c0, c1, c2, c3, c4, c5] = [\mathit{tuple}, \mathit{exactly5}, \mathit{atleast2}, \mathit{SNo}, \mathit{Sing}, \mathit{SNoLev}]$

The main difficulty, when manually proving such an automatically generated bounty, is to identify the relevant part of the formula. After a careful analysis, we found that the truth of this formula can be derived from this abbreviated version $\exists x_0.(x_0 \subseteq t_0) \wedge (\forall x_1. q x_0 x_1 \rightarrow x_1 = x_1)$ where the predicate q is used to hide the irrelevant part. Our proof, shown in Figure 2, relies on the imported definition of $\subseteq$.

6.3.2 Automated Proof

In general, proof automation tools help speed up formalization of theorems in interactive theorem provers. As a demonstration of the possible benefits, we have developed a way to automatically prove HF bounties by relying on the automation available in HOL4.

To prove a bounty, we first call HOL(y)Hammer [8] which is one of the strongest general automation techniques available in HOL4. It tries to prove the conjecture from the 6 HF axioms and the 97 HF definitions by translating the problem to external automated theorem provers (ATPs). When an external ATP finds a proof, it also returns the axioms that are necessary to find that proof. With this information, a weaker internal prover such as Metis [12] is usually able to reconstruct a HOL4 proof. The Metis proofs however typically exceed the Proofgold block size limit of 500kb and include dependencies to HOL4 axioms that are not present (and sometimes not provable) in HF. Thus, we have developed a custom internal first-order ATP for HOL4 that produces small proofs and only relies on the HF axioms. A reduction in proof size is achieved by making definitions for large terms (e.g. irrelevant parts of the conjecture and Skolem functions, similar to the example given in Section 6.3.1) and proving auxiliary lemmas for repeated sequences of proof steps (e.g., when permuting literals in clauses). With these optimizations, the automated proof for the bounty from Section 6.3.1 is only four times as large as the manual one (16kb instead of 4kb). The manual proof for this bounty has been submitted and included in the blockchain and the

bounty associated with it has been collected. In addition to that, we have so far automatically found and submitted six proofs of the HF conjectures with bounties. All these proofs were accepted by the Proofgold proof checker and the bounties were collected.

This automated system is currently limited to essentially first-order formulas. In the future, we plan to support automated proofs for higher-order formulas based on existing automated translations to first-order [15].

7 The Bounty System and its Applications

One of the main extra features of Proofgold beyond proof verification is the possibility for users and developers to attach bounties to propositions. Bounties can be used to reward users for finding proofs in mathematical domains of general interest or subproofs of a larger formalization.

7.1 Current Bounties

As mentioned in Section 4.1 for the first 5000 blocks the Proofgold consensus algorithm automatically placed a bounty of 25 Proofgold bars (half of the block reward) on a pseudorandom proposition. We say more about these pseudorandom propositions below. For the next 10000 blocks 25 Proofgold bars (half of the block reward) were placed into a "bounty fund" which was used to place larger bounties on meaningful propositions decided upon through a community forum. The propositions chosen vary from first-order problems derived from Mizar proofs, finite Ramsey properties (e.g., $R(5,7)$ is larger than the cardinality of $\wp 5$), properties of specific categories (e.g., the category of hereditarily finite sets), and numerous others. Since Block 15000 the full block reward is 25 bars and none of this goes towards the creation of bounties, and so bounties are placed by intention rather than automation.

The pseudorandom propositions from the first 5000 blocks can be classified into 8 classes. We briefly describe these here to give a concrete idea of the current bounties. The classes were determined by the initial Proofgold developers.

Random

Conjectures in this class are generally not meaningful, but the choices made during the generation are also not uniformly random. The conjecture must start with at least two (possibly bounded) quantifiers. When a term of type ι must be generated and a bound variable is not being chosen, then half the time the binary representation of a number between 5 and 20 is used, a quarter of the time the unary representation of a number between 5 and 20 is used. In the remaining quarter of the cases, half the time a unary function is chosen (leaving the argument to be generated), a quarter of the time a binary function is chosen (leaving two arguments to be generated) and the remaining quarter some other set former is used (e.g., Sep). In case the generation seems to be running out of bits of information, then it restricts the choices available.

There are three subclasses of random conjectures. The first kind is simply a sentence constructed roughly as described above. The second kind is of the form $\forall p : \iota o. \forall f : \iota\iota. s$ where s is generated as above but is allowed to use the (uninterpreted) unary predicate p and unary function f. The third kind is of the form $\forall xyz. \forall f : \iota\iota. \forall pq : \iota o. \forall g : \iota\iota\iota. \forall r : \iota\iota o. s$ where s is a generated as above though it is allowed to use x, y, z, f, g to construct sets, to use p, q, r to construct atomic propositions and is (mostly) disallowed from using the constants from the HF set theory.

The automated miner from Section 6 was tested on problems from this family.

Quantified boolean formulas (QBF)

Conjectures in the QBF class are of the form $Q_1 p_1 : o. \cdots .Q_n p_n : o.s \leftrightarrow t$ where $50 \leq n \leq 55$, each Q_i is $\forall$ or $\exists$ and s and t are propositions such that $\mathcal{F}(s) = \mathcal{F}(t) = \{p_1, \ldots, p_n\}$. The propositions s and t are generated using a similar process.

Set Constraints

One of the most challenging aspects of higher-order theorem proving is instantiating set variables, i.e., variables of a type like ιo [1]. The only known complete procedure requires enumeration of $\beta\eta$-normal terms of this type.

The set constraint conjectures are of the form

$$\forall P_1 : \alpha_1.\forall P_2 : \alpha_2.\forall P_3 : \alpha_3.\forall P_4 : \alpha_4.\varphi_1^1 \to \varphi_2^1 \to \varphi_3^2 \to \varphi_4^2 \to \varphi_5^3 \to \varphi_6^3 \to \varphi_7^4 \to \varphi_8^4 \to \bot$$

where each α_i is a small type of the form $\beta_1 \cdots \beta_{m_i} o$ and each proposition φ_j^i is a lower bound constraint for P_i over $\{P_1, P_2, P_3, P_4\}$ if j is odd and an upper bound constraint for P_i over $\{P_1, P_2, P_3, P_4\}$ if j is even. A lower bound constraint for a variable P is a formula that implies P must at least be true for certain elements. An upper bound constraint for a variable P is a formula that implies P cannot be true for more than some number of elements. Such constraints may also be recursive, e.g., saying if $P\ z$ holds then $P\ (f\ z)$ must hold. Recursive constraints can in principle be both lower bound and upper bound constraints.

The positive version of the conjecture states that there is no solution to this collection of set constraints. The negative version can be proven by giving a solution.

Higher-Order Unification

Unlike first-order unification, higher-order unification is undecidable. In spite of this Huet's preunification algorithm [11] provides a reasonable method to search for solutions. A great deal of research has been done on higher-order unification and is ongoing today [24].

The generated conjectures in this class are essentially higher-order unification problems with eight flex-rigid pairs and four variables to instantiate. The problems are given in a universal form, so that the positive form states that there is no solution. The negative form could be proven by giving a solution. In general the conjectures have the form

$$\forall X_1 : \alpha_1.\forall X_2 : \alpha_2.\forall X_3 : \alpha_3.\forall X_4 : \alpha_4.\varphi_1^1 \to \varphi_2^1 \to \varphi_3^2 \to \varphi_4^2 \to \varphi_5^3 \to \varphi_6^3 \to \varphi_7^4 \to \varphi_8^4 \to \bot$$

where α_i is a small type not involving o and φ_j^i is a proposition corresponding to a disagreement pair of a unification problem.

Untyped Combinator Unification

Since we are in a simply typed setting the untyped combinators are encoded as sets. The generated conjectures are in the form of eight flex-rigid pairs using four variables to be instantiated. Each conjecture is stated in a universal form that means there is no solution. Proving the negation of the conjecture will usually mean giving a solution, though given the classical setting it is also possible to provide multiple instantiations and prove one must be a solution. (This was also the case for the previous two classes of conjectures.) The conjectures have the form

$$\forall X.\mathsf{combinator}\ X \to \forall Y.\mathsf{combinator}\ Y \to \forall Z.\mathsf{combinator}\ Z \to \forall W.\mathsf{combinator}\ W \to \varphi_1^X \to \varphi_2^X \to \varphi_3^Y \to \varphi_4^Y \to \varphi_5^Z \to \varphi_6^Z \to \varphi_7^W \to \varphi_8^W \to \bot$$

where φ_i^V is a proposition giving a flex-rigid pair with local variables and with V as the head of the left. To be more specific each φ_i^V has the form

$$\forall x.\mathsf{combinator}\ x \to \forall y.\mathsf{combinator}\ y \to \forall z.\mathsf{combinator}\ z \to \forall w.\mathsf{combinator}\ w \to \mathsf{combinator_equiv}\ (V\ v_1\ v_2\ v_3\ v_4\ s_1\ \ldots\ s_n)\ t$$

where each $v_i \in \{x, y, z, w\}$, t is a random rigid combinator and each of $s_1, \ldots, s_n$ is a random combinator. In this context a random rigid combinator is either $K\ t_1$ or $S\ t_1$ where t_1 is a random combinator, or $S\ t_1\ t_2$ where t_1 and t_2 are random combinators, or $v\ t_1\ \cdots\ t_n$ where $v \in \{x, y, z, w\}$ and $t_1, \ldots, t_n$ are random combinators. A random combinator is $h\ t_1\ \cdots\ t_n$ where $h \in \{S, K, X, Y, Z, W, x, y, z, w\}$ and $t_1, \ldots, t_n$ are random combinators.

Each of these problems can be viewed as a first-order problem. In the first-order variant we could assume everything is a combinator (so combinator can be omitted) and use equality to play the role of combinator_equiv. It should generally be possible to mimic the equational reasoning of a first-order proof in the set theory representation by using appropriate lemmas about combinator and combinator_equiv.

Furthermore it should be possible to define a notion of reduction and prove that if two terms are equivalent via combinator_equiv, then they must have a common reduct. This would allow one to prove the positive version of the conjecture (meaning there is no solution).

Abstract HF problems

The conjectures in the Abstract HF class are about hereditarily finite sets, but without assuming the full properties about the relevant relations, sets and functions. We fix 24 distinct variables: r_0, r_1 and r_2 of type $\iota\iota o$, x_0, x_1, x_2, x_3 and x_4 of type ι, f_0 and f_1 of type $\iota\iota$, g_0, g_1 and g_2 of type $\iota\iota\iota$ and p_0, p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9 and p_{10} of type ιo. Each of these variable has an intended meaning which can be given by a substitution θ. For example, $\theta(r_0) = \in$, meaning r_0 is intended to correspond to set membership. Each generated conjecture is of the form

$$\forall r_0 r_1 r_2 : \iota\iota o. \forall x_0 x_1 x_2 x_3 x_4. \forall f_0 f_1 : \iota\iota. \forall g_0 g_1 g_2 : \iota\iota\iota. \forall p_0 \cdots p_{10} : \iota o. \varphi_1 \to \cdots \to \varphi_n \to \psi.$$

The propositions $\varphi_1, \ldots, \varphi_n, \psi$ are chosen from a set of 1229 specific propositions which hold for HF sets, but may not hold in the abstract case. The conjecture essential states that the selections of φ_i are sufficient to infer the selected ψ.

AIM Conjecture Problems

There are two kinds of AIM Conjecture [13] related problems: one using Loop_with_defs_cex1 and one using Loop_with_defs_cex2. In both cases the conjecture states that no loop exists with counterexamples of the first or second kind satisfying a number of extra equations. The two kinds of counterexamples assert that the loop has elements violating one of two identities. An AIM loop violating either of the identities would be a counterexample to the AIM Conjecture. The pseudorandom propositions do not assume the loop is AIM, but only assume some AIM-like identities hold. That is, instead of assuming all inner mappings commute, the assumption is that some inner mappings commute. Furthermore, in some cases some specific inner mappings are assumed to have a small order (which would not be true in all AIM loops).

Unfortunately there was a bug in the HF defining equation for loops (omitting that the identity element must be in the carrier). This made the negation of all of the pseudorandom propositions in this class easily provable. A Proofgold developer used this bug to collect the bounties and redistribute the bounties to the corrected versions.

Diophantine Modulo

A Diophantine Modulo problem generates two polynomials p and q in variables x, y and z and a number m (of up to 64 bits). The conjecture states there is no choice of (hereditarily finite) sets x, y and z such that the cardinality of p plus 16 is the same as the cardinality of q modulo m. The negation of the conjecture could be proven by giving appropriate x, y and z and proving they have the property.

Diophantine

The final class is given by Diophantine problems (either equations or inequalities). Two polynomials p and q in variables x, y, z are generated (as described above). Each polynomial uses 256 bits of information. The generated conjecture either states there are no (hereditarily finite) sets x, y and z such that the cardinality of p plus 16 is the same as the cardinality of q, or that the cardinality of p plus 16 is no larger than the cardinality of q.

7.2 Large Formalization Projects

Hales's Flyspeck [9] project formalizing the proof of the Kepler Conjecture has been one of the largest challenges in interactive theorem proving so far, involving several ITP communities and to some extent a centralized bounty system. It took more than 10 years to complete and combined the expertise of proof assistant users of the HOL Light, Isabelle/HOL and Coq systems. With our bounty system, the effort could have been shared with an even wider community of researchers interested in formal verification. This would involve making a plan of the steps required to prove the final theorem, splitting the formalization into multiple independent parts, and putting them as conjectures into Proofgold with bounties on them. A knowledgeable independent user of an interactive theorem prover interface capable of producing Proofgold terms, could then decide to provide a proof for a particular part. The final proof is completed when all the bounties have been collected. The reward for a particular proof may be increased if it is harder than initially thought and/or to motivate Proofgold users to solve it sooner. In the long run, an attempt at formally proving Fermat's last theorem in Proofgold could be made using this approach. An even better target to test the effectiveness of the bounty system would be the classification of finite simple groups. Its proof required the combined effort of about 100 authors for 50 years and consists of tens of thousands of pages distributed over several hundred journal articles.

8 Related Work

The most obvious related work is Qeditas, the project that evolved into Proofgold, as described in Section 2. The authors are also aware of two other ideas for projects for doing formal mathematics on a blockchain.

Mathcoin [23] describes high level ideas for a blockchain on which users can use their tokens to "bet" on whether or not a mathematical statement is true. This would allow mathematical knowledge to be reflected in the blockchain before a full proof is available.

Additionally a blockchain project for collaborative formalization of mathematics is described by Lim, et. al., in [14]. The focus in this case is on "recording and encouraging" collaboration between humans (and AI tools) formalizing in various different theorem proving systems. The authors describe in some detail the intended high level architecture (given as various layers) for such a system, and give examples for how collaboration would take place.

The two projects introduce some interesting ideas. However, so far as the authors are aware, neither project has corresponding software or a currently running network.

References

1. Chad E. Brown. Solving for set variables in higher-order theorem proving. In Andrei Voronkov, editor, *Automated Deduction - CADE-18, 18th International Conference on Automated Deduction, Copenhagen, Denmark, July 27-30, 2002, Proceedings*, volume 2392 of *Lecture Notes in Computer Science*, pages 408–422. Springer, 2002.
2. Chad E. Brown, Mikoláš Janota, and Cezary Kaliszyk. Proofs for higher-order SMT and beyond. URL: http://cl-informatik.uibk.ac.at/cek/submitted/smt2022pfs.pdf.
3. Chad E. Brown and Karol Pąk. A tale of two set theories. In Cezary Kaliszyk, Edwin C. Brady, Andrea Kohlhase, and Claudio Sacerdoti Coen, editors, *Intelligent Computer Mathematics - 12th International Conference, CICM 2019, Prague, Czech Republic, July 8-12, 2019, Proceedings*, volume 11617 of *Lecture Notes in Computer Science*, pages 44–60. Springer, 2019. doi:10.1007/978-3-030-23250-4_4.
4. Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5:56–68, 1940.
5. John H. Conway. *On numbers and games, Second Edition*. A K Peters, 2001.
6. Qeditas Developers. Qeditas technical documentation, 2016. URL: https://qeditas.org/docs/QeditasTechDoc.pdf.
7. Warren E. Ferguson, Jesse Bingham, Levent Erkök, John R. Harrison, and Joe Leslie-Hurd. Digit serial methods with applications to division and square root. *IEEE Trans. Computers*, 67(3):449–456, 2018. doi:10.1109/TC.2017.2759764.
8. Thibault Gauthier and Cezary Kaliszyk. Premise selection and external provers for HOL4. In Xavier Leroy and Alwen Tiu, editors, *Conference on Certified Programs and Proofs (CPP)*, pages 49–57. ACM, 2015. URL: http://doi.org/10.1145/2676724.2693173.
9. Thomas C. Hales, Mark Adams, Gertrud Bauer, Dat Tat Dang, John Harrison, Truong Le Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Thang Tat Nguyen, Truong Quang Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason M. Rute, Alexey Solovyev, An Hoai Thi Ta, Trung Nam Tran, Diep Thi Trieu, Josef Urban, Ky Khac Vu, and Roland Zumkeller. A formal proof of the kepler conjecture. *CoRR*, abs/1501.02155, 2015. arXiv:1501.02155.
10. John Harrison. HOL light: A tutorial introduction. In Mandayam Srivas and Albert Camilleri, editors, *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 265–269. Springer-Verlag, 1996.
11. Gérard P. Huet. A unification algorithm for typed lambda-calculus. *Theor. Comput. Sci.*, 1(1):27–57, 1975.
12. Joe Hurd. First-order proof tactics in higher-order logic theorem provers. *Design and Application of Strategies/Tactics in Higher Order Logics, number NASA/CP-2003-212448 in NASA Technical Reports*, pages 56–68, 2003.
13. Michael K. Kinyon, Robert Veroff, and Petr Vojtěchovský. Loops with abelian inner mapping groups: An application of automated deduction. In Maria Paola Bonacina and Mark E. Stickel, editors, *Automated Reasoning and Mathematics - Essays in Memory of William W. McCune*, volume 7788 of *Lecture Notes in Computer Science*, pages 151–164. Springer, 2013.
14. Jin Xing Lim, Barnabé Monnot, Shaowei Lin, and Georgios Piliouras. A blockchain-based approach for collaborative formalization of mathematics and programs. In Yang Xiang, Ziyuan Wang, Honggang Wang, and Valtteri Niemi, editors, *2021 IEEE International Conference on Blockchain, Blockchain 2021, Melbourne, Australia, December 6-8, 2021*, pages 321–326. IEEE, 2021. doi:10.1109/Blockchain53845.2021.00051.
15. Jia Meng and Lawrence C. Paulson. Translating higher-order clauses to first-order clauses. *Journal of Automated Reasoning*, 40(1):35–60, 2008. doi:10.1007/s10817-007-9085-y.
16. F. Pfenning and C. Elliot. Higher-order abstract syntax. *SIGPLAN Notices*, 23(7):199–208, June 1988.

17 Andrew Poelstra. On Stake and Consensus, 2015. URL: `https://download.wpsoftware.net/bitcoin/pos.pdf`.

18 Dag Prawitz. *Natural deduction: a proof-theoretical study*. Dover, 2006.

19 R. Diaconescu. Axiom of choice and complementation. *Proceedings of the American Mathematical Society*, 51:176–178, 1975.

20 Bertrand Russell. *The Principles of Mathematics*. Cambridge University Press, 1903.

21 Konrad Slind and Michael Norrish. A brief overview of HOL4. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5170 of *LNCS*, pages 28–32. Springer, 2008. `doi:10.1007/978-3-540-71067-7_6`.

22 M.H.B. Sørensen and P. Urzyczyn. *Lectures on the Curry-Howard Isomorphism*. Rapport (Københavns universitet. Datalogisk institut). Datalogisk Institut, Københavns Universitet, 1998.

23 Borching Su. Mathcoin: A blockchain proposal that helps verify mathematical theorems in public. *IACR Cryptol. ePrint Arch.*, 2018:271, 2018.

24 Petar Vukmirovic, Alexander Bentkamp, and Visa Nummelin. Efficient full higher-order unification. In Zena M. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, June 29-July 6, 2020, Paris, France (Virtual Conference)*, volume 167 of *LIPIcs*, pages 5:1–5:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

25 Bill White. Qeditas: A formal library as a bitcoin spin-off, 2016. URL: `http://qeditas.org/docs/qeditas.pdf`.

Multi: A Formal Playground for Multi-Smart Contract Interaction

Martín Ceresa
IMDEA Software Institute, Madrid, Spain

César Sánchez
IMDEA Software Institute, Madrid, Spain

Abstract

Blockchains are maintained by a network of participants, miner nodes, that run algorithms designed to maintain collectively a distributed machine tolerant to Byzantine attacks. From the point of view of users, blockchains provide the illusion of centralized computers that perform trustable verifiable computations, where all computations are deterministic and the results cannot be manipulated or undone.

Every blockchain is equipped with a crypto-currency. Programs running on blockchains are called smart-contracts and are written in a special-purpose programming language with deterministic semantics[1]. Each transaction begins with an invocation from an external user to a smart contract. Smart contracts have local storage and can call other contracts, and more importantly, they store, send and receive cryptocurrency.

Once installed in a blockchain, the code of the smart-contract cannot be modified. Therefore, it is very important to guarantee that contracts are correct before deployment. However, the resulting ecosystem makes it very difficult to reason about program correctness, since smart-contracts can be executed by malicious users or malicious smart-contracts can be designed to exploit other contracts that call them. Many attacks and bugs are caused by unexpected interactions between multiple contracts, the attacked contract and unknown code that performs the exploit.

Moreover, there is a very aggressive competition between different blockchains to expand their user base. Ideas are implemented fast and blockchains compete to offer and adopt new features quickly.

In this paper, we propose a *formal playground* that allows reasoning about multi-contract interactions and is extensible to incorporate new features, study their behaviour and ultimately prove properties before features are incorporated into the real blockchain. We implemented a model of computation that models the execution platform, abstracts the internal code of each individual contract and focuses on contract interactions. Even though our Coq implementation is still a work in progress, we show how many features, existing or proposed, can be used to reason about multi-contract interactions.

2012 ACM Subject Classification Theory of computation → Program reasoning; Theory of computation → Program constructs; Theory of computation → Abstract machines

Keywords and phrases blockchain, formal methods, theorem prover, smart-contracts

Digital Object Identifier 10.4230/OASIcs.FMBC.2022.5

Related Version *Full Version*: https://arxiv.org/abs/2207.06681

Funding This work was funded in part by the Madrid Regional Government under project "S2018/TCS-4339 (BLOQUES-CM)" and by a research grant from Nomadic Labs and the Tezos Foundation.

[1] Although the behaviour of smart-contracts may depend on values to be known at runtime, i.e. block number; hashes; etc, their behaviour is deterministic.

© Martín Ceresa and César Sánchez;
licensed under Creative Commons License CC-BY 4.0

1 Introduction

Smart-contract manipulate cryptocurrency, which has a corresponding value as money. Since smart-contracts cannot be modified once installed and their computations cannot be undone ("the contract is the law"), all interactions with the contract are considered valid. Therefore, there is an incentive for malicious users to take advantage from unexpected behaviors and interactions. Also, errors in contracts can result in losses and cryptocurrency being locked indefinitely, even when used but by well-intentioned users. We focus in this paper on the computational notion of correctness, and not on the real legal implications resulting from interactions in the blockchain or the use of smart-contracts to enforce legally binding contracts [8].

One important reason why it is very difficult to reason about smart contracts is that they live in an *open universe*. Even though the code of a given smart-contract C cannot be modified once installed, other contracts that call and are called from C can be programmed and deployed after malicious users study C. Therefore, programmers and auditors of contract C did not have to analyze all possible code that can invoke or be invoked from C.

At the same time, users demand blockchains to implement new features. Since there is a big competition between blockchains, this puts pressure on architects of blockchains on the time to market of new features. And each new feature potentially increases the attack surface of smart contracts.

There are different kinds of errors found in smart-contracts.

- *Logical problems* are related to errors in the logic of the smart-contract. Usually, attackers detect a corner case that can be exploited to generate an unwanted behaviour.
- *Low-level execution* problems that arise from a misunderstanding on details of the low-level execution platform. Examples include underflow, overflow or exploiting unexpected behavior after the stack limit is reached.
- Programmer can also employ *bad idioms* that they are familiar with from other areas of software applications, but which may be dangerous in interactive platforms like blockchains, where all data (including the state of the contracts) is public and verifiable.

Most bugs are related to multi-contract interactions. For example, the infamous DAO attack where malicious code *legally* exploited the machinery of the Ethereum blockchain creating unexpected re-entrant calls from remote contracts led to the loss of $60 million [14].

In this article, we present a formalization in Coq of a general blockchain model of computation that allows us to study new multi-contract interactions as well as new features. We aim to develop a formal and rigorous way to analyze the possible interactions between contracts and also to study how new features affect contracts before they are implemented and deployed. Our Coq library allows simulating the execution of smart-contracts, abstracting away the internal code of the contract. Our abstraction is based on the Tezos blockchain, but it is general enough to cover other blockchains like Ethereum. We model smart-contract (almost) as pure functions from the current storage and state of the blockchain into (possibly) a list of operations to do next plus changes in the storage.

2 Motivation

After successful attacks like DAO [14] there is a growing interest in formal methods for smart-contracts. First, there is an interest in verifying that a contract satisfies a specification so certain properties can be guaranteed, e.g. the owner will be able to fetch all funds or that a bidder will either gain the bidding or recover the funds. Second, it is also important to

formally study different mechanisms and features proposed for a given blockchain before they are offered so new attacks can be prevented. Some of these mechanisms are proposed to allow users to use more effective defensive programming idioms.

For example, by analyzing the DAO attack [9] proposed a property called *effectively callback free* which restricts the interactions within smart-contracts disabling these attacks. Later on, the Tezos blockchain [1] implements such property by construction: smart-contracts are functions that either fail or returning a list of operations to be executed plus a new storage. Therefore, the storage is updated before the operations are executed, which prevents attacks like the DAO using this programming style.

In order to prevent these attacks, the Tezos blockchain followed a conservative scheduling strategy. In Tezos, as is the general case, every transaction begins with a request by an external user indicating the smart-contract to invoke, method and arguments, and balance of the initial operation. Assume user *Alice* starts a transaction invoking method f of smart-contract C, and that, after executing $C.f$ we have a list of operations $[o_0, \ldots, o_n]$. To compute the result of the transaction, the blockchain will execute each operation o_i in order, until the gas is exhausted or the list of pending operations is empty. The order in which the operations are executed affects the outcome of the transaction. Two conventional strategies are: (1) to insert the new list of operations at the beginning of the list of pending operations (DFS) (2) to insert the new list of operations at the end (BFS). The first one, DFS, allows us to implement a call-and-return flow of computation and it is the more conventional in most blockchains. The second one, BFS, prevents call injection attacks by construction as one can guarantee that two operations are executed back-to-back and was used until version 8 of Tezos (Protocol Edo) [5]. In our example, assuming that executing o_1 generates bs operations, the result of the previous execution would be $[o_2, \ldots, o_n] \cdot bs$. While in DFS, the result would be $bs \cdot [o_2, \ldots, o_n]$, and thus, the instructions in bs will be executed before $o_2, \ldots, o_n$. However, BFS suffers from other classes of problems.

Assume a bank contract that holds money for a customer and the bank contract is willing to send money as long as the balance stays above threshold `threshold`. In a solidity like language, the contract could be as follows:

```
contract Bank {
  uint threshold;
  address owner;
  constructor(uint _threshold, address _owner) public {
    threshold = _threshold;
    owner = _owner;
  }
  function deposit() payable public{
    return([]);
  }
  function withdraw(uint ret) public {
    if (sender = owner) then
        if (balance - ret > threshold) then
            return ([transfer(owner.Receive, ret)])
        else
            fail("breaking invariant")
    else
        fail("not owner")
  }
}
```

Normal usage of a such a bank contract can be:

```
contract GoodClient{
    address bank;
    // ...
    function askMoney(uint m){  // Requests m from the vault
        return([bank.withdraw(m)]);
    }
}
```

On the other hand, the following is a simple attack exploiting the bank contract:

```
contract Bad{
   address bank;
   //...
   function rob(uint n, uint m){ // BFS attack to the vault!
       return(ntimes n [bank.withdraw(m)])
   }
}
```

The new method called `rob` generates a list of invocations to the vault. Assume the vault contract has a threshold of 9 and that is in a state in which it stores 15 units of cryptocurrency. A simple examination suggests that the vault will send money back to its owner whenever its balance is greater than 9, effectively allowing only one withdrawal. However, consider the following execution starting from `[rob(3,5))]`. After executing the operations, we would have the following pending queue:

```
[(Bad, vault.withdraw(5)), (Bad, vault.withdraw(5)), (Bad, vault.withdraw(5))]
```

Then the BFS sequence of executions leads to the following sequence of pending operations:

```
[(Bad, vault.withdraw(5)),  (Bad, vault.withdraw(5)),  (Bad, vault.withdraw(5))]  ⇝
[(Bad, vault.withdraw(5)),  (Bad, vault.withdraw(5)),  (Vault, Bad.Receive())]    ⇝
[(Bad, vault.withdraw(5)),  (Vault, Bad.Receive()),    (Vault, Bad.Receive())]    ⇝
[(Vault, Bad.Receive()),    (Vault, Bad.Receive()),    (Vault, Bad.Receive())]    ⇝
[(Vault, Bad.Receive()),    (Vault, Bad.Receive())]                               ⇝
[(Vault, Bad.Receive()) ]                                                         ⇝
[]
```

First, the operation sending the money back to contract `Bad` is added at the end, as dictated by BFS. Second, according to the semantics of feature "transfer" in the Tezos blockchain, funds are subtracted from the sending contract `Vault` after the transfer is executed. Therefore, the second `withdraw` request does not see the effect of attending the first one. The combined effect is that all three requests are attended resulting in a total extraction of 15 units leaving 0 in contract `Vault` *without noticing the attack*. The attack is based on the separation between the creation of a transfer and its execution. The lesson is that even though a BFS order prevents injection attacks, it allows attacks based on the delayed effect of emitted operation. The contract `Vault` can be easily fixed by encoding in a variable in the storage the balance that has been compromised with a future transfer. If necessary, `withdraw` can create two operations (1) the transfer, and (2) an invocation to a new private method in `Vault` whose purpose is to note that the compromised balance created by a withdraw has been effectively arrived.

Another lesson is that relying on the balance of contracts is considered a bad smart-contract programming practice. Assume now that programmers would like the architects of the blockchain to implement not only `balance` but also `pending_balance`, which accounts for transfers sent but not executed. Moreover, assume also that the blockchain also implements the feature of *views*, an apparently innocent feature that simply returns information about the storage of a contract without any effect. We illustrate that these two features combined can lead to undesirable effects. For example, if we would like to maintain the invariant that at every moment the amount of combined funds between a collection of contracts is constant, the combination of `pending_balance` and views can break such an invariant.

For example, consider three smart-contract A, B, C, and the following pending queue of operations:

$$[\underbrace{A_1, \ldots, A_o}_{A}, \underbrace{C_1, \ldots, C_m}_{C}, \underbrace{B_1, \ldots, B_n}_{B}]$$

where A sends money to B – in operations that are going to be executed after C but that update A pending balance. This leaves C in a difficult position. If C observes (using views) the balances of A and B there is going to be a mismatch with their real balances, because C will see the pending compromised balance but not the pending receives, which may induce bad behaviour in C. If C depends on $A.balance + B.balance$, for example, to buy some NFT it may incorrectly fail to take the right decision. A possible solution is to introduce yet another feature that captures pending receives.

In our line of work, we aim to build a *formal playground* where different features and mechanisms can be encoded and reasoned about easily and formally, while also simulating the execution of multiple contracts.

3 Previous Work

In our work, we follow a static verification approach where contracts and features are analyzed before deployment. The idea is to encode how blockchains are implemented and study the behavior of contracts and features by formally proving properties. Several approaches have been suggested for testing, model checking and functional and temporal verification of smart-contracts. We review the most relevant.

Mi-Cho-Coq. Mi-Cho-Coq is the first verification tool implemented in Coq for the Tezos blockchain ecosystem [4]. The main difference between Mi-Cho-Coq and our effort (Multi) is that Mi-Cho-Coq focuses on the analysis of the code of a single contract (or collection of calling contracts for which the code is available). We say that Mi-Cho-Coq implements small-step semantics to prove *functional properties*, which requires to have a concrete specification of a smart-contract and either its code or a higher level specification.

The main difference with Mi-Cho-Coq is that our goal is to prove properties *emerging* from interactions between smart-contracts. Our tool is a complementary effort to lower-level verification tools as Mi-Cho-Coq.

Concert. Concert [3] is another framework written in Coq to prove formal properties of smart-contracts, and in this case, they accept multi-contract interaction [11]. The fundamental idea of Concert is to model of smart-contracts as agents and computation as interaction (message passing) between these agents. They also implement specific mechanisms, for example, they implement delegation primitives in the Tezos blockchain. Moreover, Concert has an extraction mechanism to extract high-level smart-contracts written in Ligo [10].

Our main difference is that we implement a very flexible framework with the idea of encoding new potential blockchain features and prove properties of how different features interact with each other. Including BFS and DFS scheduling in the Tezos blockchain, but there may be other scheduling strategies.

Concert implements blockchains in a generic way using specific features of Coq (class system) and meta-programming features to easily embed blockchain smart-contract languages. Concert also builds proofs by inspecting the trace representing the evolution of the blockchain observed by a *small step relation*.

Implementing new blockchain features relating to how smart-contracts are executed is an important feature in our framework, and moreover, we want to be able to reason and prove properties about such features. For example, what would happen if smart-contracts can inspect *runtime* information as the stack call (what the next operations or pending operations are). Another difference is that (so far) we observe the state of the blockchain comparing just the state of the blockchain before a transaction begins and after a transaction ends. We are also able to inspect intermediate transition steps, but we are not exploiting that feature yet.

Scilla. Scilla is a smart-contract language embedded in Coq [16] that allows some temporal reasoning (see [17]). Scilla is an embedded domain-specific language in Coq which also abstracts smart contracts as functions returning a list of operations. The main difference between Multi and Scilla is that we do not present a language to write smart-contract but use Coq functions directly. We share the point where the effects of executing smart-contracts are simple a list of operations that are propagated by the executer. As Concert, we have a clean separation between the language of smart-contracts and the machinery required to execute smart-contracts. However, in our case, we decoupled the scheduler from the execution of single instructions, and thus, we can implement different scheduling strategies independently of the set of operations.

VerX. VerX is an automatic software verification tool that checks custom functional properties of smart-contract entrypoints. VerX works on a similar level to Mi-Cho-Coq, in the sense that they prove functional properties of smart-contracts, but it is built to be completely automatic and also to handle some multi-contract interactions. The interaction between smart-contracts comes from performing analysis on the possible onchain behaviours of a set of smart-contracts. VerX restricts the analysis to a set of smart-contracts, S, that have a condition called *effectively external callback free contracts*, which states that any behaviour generated by an interaction between smart-contracts in set S that has an external call is equivalent to a one without external calls [13]. This follows the lines of [9]. Because of that restriction, they can reason about smart-contract, proving PastLTL specifications, but it also restricts them to work in a **close universe**.

SmartPulse. SmartPulse [18] is another automatic verification tool for smart-contracts. The main goal is to verify temporal properties including some simple liveness properties. This tool is similar to *VerX* but it is focused on proving liveness properties of a single contract in a closed universe. They do not support multi-contract interaction.

4 Model of Computations

Blockchain Model. We ignore the internals of the infrastructure of blockchain implementations (like cryptographic primitives, consensus algorithms or mempools) and focus exclusively on the model of computation that blockchains offer to external users. The blockchain is then abstracted by a partial map from addresses to smart-contracts. Smart-contracts are programs with some structure:

- Storage: a segment of memory that can only be modified by the smart-contract.
- Balance: an attribute of contracts that indicates the amount of cryptocurrency stored in the contract.
- The program code: a well-formed program that represents the implementation of the smart-contract.

The state of a smart-contract is a proper value of its storage plus the balance its stores. The model of computation consists of the sequential execution of transactions, each of which is started by the invocation of an operation. In the current version, we ignore how gas or fees are paid or how new currency is created during the evolution of the blockchain to pay the bakers. Smart-contracts can be executed upon request from an external user that initiates a transaction or by the invocation from a running contract. Upon invocation, the blockchain evaluates the result of executing the smart-contracts program following a given semantics producing effects on the blockchain (further invocations) and changes on the smart-contracts' storage or they may fail.

Open Universe. We introduce now the concept of *universe of computation*. Once a smart-contract has been installed on a blockchain, every other entity in the blockchain can interact with it. The smart-contract itself can invoke or be invoked by older or newer contracts. The case of smart-contracts invoking just older and well known contracts can be useful sometimes but in general smart-contracts may not know a priori who they are going to interact with. This differs from conventional software where components are built from well-known trustable components and the surface of interaction with potentially malicious usage is small and well defined. The classical way of programming exposes the internals of complex software and leaves open attack vectors. For example, to guarantee certain behaviour high-level smart-contracts invoke low-level smart-contracts following a protocol to logically guarantee a result. However, malicious software **may not** follow such protocols possibly breaking or leaving low-level smart-contracts in an incorrect state. This open universe model of computation forces smart-contracts to implement defensive mechanisms to prevent undesired executions.

Most verification techniques and frameworks mentioned previously do not take into care such assumption. They operate under the idea that smart-contracts behave the way they are supposed to, in the sense, that either they avoid external call invocations by removing interactions or by assuming they are interacting with good smart-contracts. However, this is not the case, the blockchain is an aggressive environment, a so called *dark forest* [15]. In this paper, we study this problem attempting to formalize properties of smart-contracts operating under a more realistic (and pessimistic) view of the world and also to develop new mechanisms or features to explicitly guarantee that we are working under a safe environment. Such mechanisms could be implemented inside smart-contracts, but not every mechanism can be implemented using current blockchain technologies, like transaction monitors [6, 7].

5 Formalization

In this section, we describe the building blocks of our Coq library implementation that allows us to reason about different blockchain execution mechanisms. Our goal is to study how smart-contracts interact with other smart-contracts, and thus, we abstract away the internal execution of the instructions of the smart-contract. Moreover, we need a framework flexible enough to implement new features (i.e. different execution models, scheduling strategies, etc) and, additionally, a formal system to prove and verify properties of interactions between smart-contracts implementing and using such features. In short, we implemented a *formal playground* simulating the model of computation of blockchains.

We abstracted blockchains following the model described in Section 4 in the proof-assistant Coq. We interpret smart-contracts as pure functions in the host language Coq and every additional feature is implemented on top of pure functions.

Smart-contracts are implemented as a structure with three fields (Listing 1): a storage, a balance, and a pure function implementing the smart-contracts code.

```
Structure SmartContract (Ctx Param Storage Error Result : Type) : Type :=
  mkSmartContract {
       (* Storage *) _Sst : Storage ;
       (* Balance *) _Sbalance : ℕ ;
       (* Computation that result in an element of type Result *)
       _Sbody : Ctx → Param → Storage → Error + (Result * Storage)
    }.
```

Listing 1 Smart contract Definition.

Note that structure `SmartContract` is highly parametric:

- Parameter `Ctx` represents what smart-contracts can observe about the blockchain and the execution model as: current block level, the total balance of the transaction, who the sender and source are, etc.
- Parameter `Param` represents the parameters the body of the smart-contract expects to receive; using `Param` we model the different entrypoints of a contract.
- Parameter `Storage` represents the storage of the smart-contract.
- Parameter `Error` represents the type of errors that can result from the execution of the smart-contract.
- Parameter `Result` represents the resulting type of smart contracts, which in the Tezos model is a list of further operations.

The type `SmartContract` represents the most basic structure of a smart-contract. It is simply a structure with some storage, balance and a body.

The Smart-contracts body is modeled as a pure functions from the current state of the blockchain and its storage to a sequence of operations. In this way, we abstract away concrete blockchain programming languages or implementations. Even though our formalization is based on the semantics of method invocations in the Tezos blockchain, different programming language can be modeled in this paradigm using standard compiler techniques (essentially dividing a complex function with effects into its basic blocks that are pure functions as modeled here).

5.1 Execution

The execution of a smart-contracts, aside from changes in the storage, also produces a sequence of operations to be executed. Therefore, we have to take care of two things: how to execute these operations, and how to order the execution. We split the execution model into two main pieces: a scheduler and an executor.

Scheduler. The scheduler is in charge of the order of execution, adding new operations the pending queue (either at the beginning or the end, etc). The scheduler is also in charge of creating new contexts. Finally, it is in charge of building the graph/tree of transactions, every information that descendants of an operation may share is kept and organized by the scheduler.

Executer. The executer is in charge of executing an operation in a given context, and it is the same independently of the evaluation order. The most basic operation of an executor is smart-contract invocation, which requires that the executor collects and builds the environment in which such invocation should be executed. The context is the blockchain state from the point of view of the contract execution. Another operation is smart-contract creation, which in this case it is going to generate a modification to the blockchain, and communicate it to the scheduler.

Operations. We assume the blockchain has a simple set of operations. We start from a minimal set of operations that is simple enough to enable smart-contracts interaction, and later add new operations as needed afterward.

We begin our implementation with two operations: `Transfer` and `Create_Contract`.

- Operation `Transfer` performs an invocation to a given address while also sending money.
- Operation `Create_Contract` installs a new smart-contract at an indicated address with an initial amount of balance and storage.

```
Inductive EnvOps : Type :=
| Transfer : forall (T : Mich_Type),
    (* Parameter *) (Type_Interpret T) →
    (* Amount to transfer *) Mutez →
    (* Contract address to invoke *) (Type_Interpret (ContractT T)) →
    EnvOps
| Create_Contract : forall (PTy StTy : Mich_Type),
    (* Pre-computed Address *) Address →
    (* Initial amount *) Mutez →
    (* Initial Storage *) (Type_Interpret StTy) →
    (* Body *) MichBodyTy PTy StTy (list EnvOps) →
    EnvOps.
```

Where `Mich_Type` is an enumeration type of the different data structures supported by the blockchain, i.e. natural numbers, strings, etc. In our case, since we are working close to the implementation of the Tezos blockchain, we implement most of its data structures, and we represent them as an inductive type `Mich_Type`. Using the previous operations, we can define smart-contracts simply as the following structure:

```
Structure MichContract : Type := mkMich {
    (* Contract parameter type *) _Param : Mich_Type ;
    (* Storage type *) _Storage : Mich_Type;
    (* Contract body*)
    _Soul : SmartContract
                (TzCtxt _Param)
                (Type_Interpret _Param)
                (Type_Interpret _Storage)
                OError
                WritingContext;
    }.
```

Essentially, we capture smart-contracts as their body plus information about their types. Hiding away the type information forces us to implement a lot of type matching clauses when it comes to the execution of smart-contracts. However, it enables us to represent the state of the blockchain simply as a (partial) map of addresses to smart-contract.

```
Definition TezosEnvironment := string → option MichContract.
```

Given an operation, the executer is in charge of building the required information to execute. In the case of an invocation to an address `addr`, the executer looks up the address `addr` into the current environment to see if there is a smart-contract matching the expected type at that address, and in that case, executes its body to obtain either a new storage and further operations or a fail. In the case of a smart-contract creation operation, the executer is in charge of checking that the address is actually free and updating the environment adding such smart-contract. Finally, the executer is also in charge of checking that smart-contracts have enough balance to perform transactions and update the current environment with the new balances.

We can characterize our executer as follows:

```
Definition ExecuterTy : Type :=
   (* Input context information *) (ctx : ExecutionContext)
→ (* Operation to execute*) (o : EnvOps)
→ (* Current state *) (env : BCEnvironment)
→ MFail (* possibly returning: *)
        (option(
          (* Address emitting new operations, next sender *) Address *
          (* Effects generated (new operations) *) WritingContext)
          * (* Updates to the environment *)
          (list (Address * MichContract))).
```

Different executers exercising type `ExecuterTy` can interpret operations in different ways. Executers receive two arguments, `ctx` and `env`, representing the execution context and the environment of the blockchain, respectively, and in return, provide the modifications to the environment and possibly a list of new operations. Note that `ExecuterTy` leaves some proofs obligations if we want to simulate current blockchains, i.e. we need to show that `ExecuterTy` does not modifies or upgrades exiting smart-contracts' code (see Section 5.2).

Schedulers are in charge of gluing together the effects generated by the execution of operations in the current blockchain. We model them in Coq as a type listed in Listing 2 where `SchedulingStrategy` implements the execution order to follow. In other words, schedulers keep track of the evolution of the state of the blockchain while managing the pending queue of operations. Schedulers take the first operation on the pending queue,

build the information required by the executor, and pass everything to an executer. When executers return, schedulers take the resulting operations and updates to the current state of the blockchain.

```
Definition Scheduler : Type
  := (* Strategy *) SchedulingStrategy
  → (* External user *) Address
  → (* Executor *) ExecuterTy
  → (* Current environment *) BCEnvironment
  → (* Time *) Timestamp
  → (* Pending Execution list *) list (list EnvOps * ExecutionContext)
  → (MFail BCEnvironment * Timestamp).
```

Listing 2 Schedulers type.

The computation of a transaction begins with an external user (outside the blockchain) posting one or more operations to be executed, defined in Listing 3. The initial transaction

```
Structure SignedTrans : Type := mkSignedTrans {
      _author : Address; _trans : list EnvOps
    }.
```

Listing 3 Signed transactions definition.

is given to the scheduler, which also receives a scheduler strategy, an executer, and a context to compute the transaction and its descendants operations. The result is a pair composed of a possible new environment and the next timestamp. We need timestamps to represent the passage of time, and thus, time progresses even in the case that a transaction is reverted. In practice, the scheduler strategy is fixed for a given blockchain.

Since blocks in the blockchain are just sequences of signed transactions, `SignedTrans`, we can generate arbitrary traces with systems like QuickChick [12]. Given a logical program (reflected in a set of smart-contracts), we can codify the possible logical operations in an inductive type in Coq. Therefore, we can generate a sequence of actions translating the logical steps into transactions in the blockchain and verify that the smart-contracts do not reach an invalid state.

5.2 Proof of Correctness

We can define a specification of how a proper blockchain should behave and check that our implementation follows the specification. For example, a basic property is *no-double spending* which states that transfers (remote contract invocations) are paid once, i.e. the sender is not charged twice for the same operation. We can go even further and prove that executing a transfer does exactly what it is supposed to do (Listing 4), i.e. invokes another smart-contract, executes its code, deduce the expected amount from the sender's account, and adds it to the destination's account, or fail (in which case the transfer has no effect).

An alternative approach would be to define a small step inductive relation defining how blockchains should behave and prove that the scheduler follows it step by step. The framework Concert [3] follows that approach.

```
Lemma SimpleTransferCheck :
  forall callerContract calleeContract parameterTy BCCtxt send
    (parameter : Type_Interpret parameterTy)
    (storage storage' : Type_Interpret (_Storage calleeContract))
    (contractContext : TzCtxt parameterTy),
    successWith (ops, (caller', callee'))
                (SimpleTransfer callerContract send calleeContract)
    → _st calleeContract ≡ storage
    → successWith (ops, storage') (exec calleeContract BCCtxt parameter)
    ∧ ((_balance callerContract) - send) ≡ _balance caller'
    ∧ ((_balance calleeContract) + send) ≡ _balance callee'
    ∧  _st callee' ≡ storage'.
```

Listing 4 Transfer is correct.

5.3 Multi-Contract Interaction Proofs

The most important part of our framework is that we can simulate executions of smart-contracts and inspect the effects generated by smart-contract interactions. In other words, we have a big-step semantics of blockchain operations where we can study how smart-contracts using different mechanisms (i.e. BFS/DFS, etc) interact with each other. We can build proofs either by observing the evolution of the transaction execution operation by operation, or analyzing its final state after the transaction terminates. In other words, we have a definition of observational equivalence of smart contracts modulo the particular blockchain employed as evaluator.

This is extremely useful because we can abstract away entire smart-contracts and event simulate the more realistic scenario: a demonic environment. Either we know the code of smart-contract and we can predicate over these code during the proof, or we do not have these code, which requires reasoning with universal quantification over all possible smart-contracts. In other words, to prove that smart-contracts are prepared to operate properly in the open universe of the blockchain requires to reason about the interactions with all possible contracts.

We can model angelic computations by expanding our known universe of smart-contracts simply by implementing smart-contract on our framework and having them installed in the blockchain inside a simulation.

6 Conclusion

In this paper, we present Multi, a formal playground to reason about smart multi-contract interaction and to study features of the blockchain before deployment. Additional features and mechanisms are described in Appendix C and Appendix B where we introduce the idea of *Bundles* of operations: semantic restrictions on the execution of a sequence of operations. Our framework, based on the Tezos blockchain, is very general and allows us to reason about different execution orders, abstracting away each operation on a contract by a pure function whose output is either a failure or the changes in the local storage plus further operations.

Future work includes:

- Examples and study cases: implement and study complex use cases.
- Integrate Multi to the Tezos formal ecosystem and study interactions with Concert and Mi-Cho-Coq.

- Implement additional features, e.g. transaction monitors, views, etc, and study how they interact between each other.
- Design and implement a DSL to easily encode specific smart-contracts easing the translation from existing languages into Coq functions.
- Write more expressive smart contract types following the steps of Concert since Coq functions are more general than the contracts accepted by most blockchains (like Tezos).
- Implement complex features as *tickets*/NFT using some mechanisms (like monads) to better capture the space of functions that represent smart-contracts.

Finally, we aspire to implement a richer specification language using ATL [2] to describe the interaction between smart-contracts and fully verify their specification in Coq. The idea consists in describing programs as interactions between agents (i.e. smart-contracts) where agents cooperatively guarantee certain properties or exercise certain rights. At the semantic level, we would connect the evaluation of smart-contracts in a blockchain with their semantic given by ATL and concurrent games. In other words, with Multi, we can interact between a rich specification language of smart-contracts and their behaviour defined by the execution of blokchains.

References

1. Victor Allombert, Mathias Bourgoin, and Julien Tesson. Introduction to the tezos blockchain, 2019. `doi:10.48550/ARXIV.1909.08458`.
2. Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. *J. ACM*, 49(5):672–713, September 2002. `doi:10.1145/585265.585270`.
3. Danil Annenkov, Jakob Botsch Nielsen, and Bas Spitters. Concert: a smart contract certification framework in coq. *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, January 2020. `doi:10.1145/3372885.3373829`.
4. Bruno Bernardo, Raphaël Cauderlier, Zhenlei Hu, Basile Pesin, and Julien Tesson. Micho-coq, a framework for certifying tezos smart contracts. *CoRR*, abs/1909.08671, 2019. `arXiv:1909.08671`.
5. Tezos Blockchain. Tezos agora: Florence, no ba (psflorena), 2021-12-24. URL: `https://agora.tezos.com/period/46`.
6. Margarita Capretto, Martin Ceresa, and Cesar Sanchez. Transaction monitoring of smart contracts, 2022. `doi:10.48550/ARXIV.2207.02517`.
7. Alberto Cuesta Cañada, Fiona Kobayashi, fubuloubu, and Austin Williams. Eip-3156: Flash loans. URL: `https://eips.ethereum.org/EIPS/eip-3156`.
8. Joshua Ellul, Jonathan Galea, Max Ganado, Stephen Mccarthy, and Gordon J. Pace. Regulating blockchain, dlt and smart contracts: a technology regulator's perspective. *ERA Forum*, 21(2):209–220, October 2020. `doi:10.1007/s12027-020-00617-7`.
9. Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. Online detection of effectively callback free objects with applications to smart contracts. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. `doi:10.1145/3158136`.
10. Ligo. Ligo: A friendly smart contract language for tezos, 2022. URL: `https://ligolang.org/`.
11. Jakob Botsch Nielsen and Bas Spitters. Smart contract interactions in coq. In *FM Workshops (1)*, volume 12232 of *Lecture Notes in Computer Science*, pages 380–391. Springer, 2019.
12. Zoe Paraskevopoulou, Cătălin HriŢcu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin C. Pierce. Foundational property-based testing. In Christian Urban and Xingyuan Zhang, editors, *Interactive Theorem Proving*, pages 325–343, Cham, 2015. Springer International Publishing.

13 Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. Verx: Safety verification of smart contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1661–1677, 2020. `doi:10.1109/SP40000.2020.00024`.

14 Daian Phil. Analysis of the dao exploit, 2016. URL: `https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/`.

15 Dan Robinson and Georgios Konstantopoulos. Ethereum is a dark forest, 2020. URL: `https://www.paradigm.xyz/2020/08/ethereum-is-a-dark-forest`.

16 Ilya Sergey, Amrit Kumar, and Aquinas Hobor. Scilla: a smart contract intermediate-level language, 2018. `doi:10.48550/ARXIV.1801.00687`.

17 Ilya Sergey, Amrit Kumar, and Aquinas Hobor. Temporal properties of smart contracts. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, pages 323–338, Cham, 2018. Springer International Publishing.

18 Jon Stephens, Kostas Ferles, Benjamin Mariano, Shuvendu Lahiri, and Isil Dillig. Smartpulse: Automated checking of temporal properties in smart contracts. In *42nd IEEE Symposium on Security and Privacy*. IEEE, May 2021. URL: `https://www.microsoft.com/en-us/research/publication/smartpulse-automated-checking-of-temporal-properties-in-smart-contracts/`.

A Angelic/Demonic

Given the open universe nature of blockchains, smart-contracts are forced to identify who are they interacting with. Programmers when they are designing complex software do not think that they are in a dangerous and aggressive environment, as it is now, and simply think that smart-contracts will interact with good pieces of software doing what they are supposed to do. However, as we saw before, this may not be true.

In this section, we present a new characterization when it comes to classifying the interaction between multiple smart-contracts. We call this characterization *Angelic/Demonic* where we mark smart-contracts as *angelic* when they do what they are supposed to do, or as *demonic* when we cannot assume anything about their behaviour, and thus, we cannot predict nor predicate about their behaviour. Note that this is not a property enforced by blockchains, but it is more of a mindset at the moment of designing complex software that is going to run on the blockchain.

There are essentially two basic models to reason about multi-contract interaction:

Closed World Assumption: every smart-contract knows and trusts the smart-contracts that it is invoking (directly and transitively). In particular, every smart-contract C only invokes contracts that are older than C and whose properties are known.

Open World Assumption: every contract C runs in an adversarial environment and smart-contracts should protect against possible *evil* smart-contracts.

A closed world assumption is feasible on many occasions because of the public and *immutable*[2] character of the blockchain. Since everything is public and smart-contracts do not change, as smart-contract developers, we can observe the state and code of smart-contracts that we are going to interact with and decide if they are *angelic*, i.e. if they do what they are supposed to do.

Note that "the angelic state" is fragile and it may change. For example, assume we invoke a smart-contract B that in turn invokes another smart-contract whose address *addr* is stored in B's storage. As we are about to submit our smart-contracts to the blockchain, we can

[2] Although it is possible to implement mutable and upgradable smart-contracts, this is not the general case, and even if the nature of the smart-contract was to mutate this would be known by the invoker.

explore and decide that B and the current *addr* are angelic. However, eventually, B may change it to another smart-contract $addr'$ that may also be angelic to B, or B is protected towards possible attacks from $addr'$, but it may open an attack on our smart-contract.

The second option, an open world assumption, is a more real situation and sometimes the only possible case for certain smart-contracts. One of the most prominent cases is exchange houses: let *Dex* be a smart-contract that is always willing to exchange token A for token B for a certain fee in behave of a set of investors. In this case, the smart-contract *Dex* is doomed to interact with unknown addresses.

Another example is that we can implement a call-and-return model using *continuation passing style* between smart-contracts in BFS blockchains. However, implementing such interactions between smart-contracts requires to assume that *every smart-contracts is going to behave accordingly*, and thus, we are under an angelic assumption. Therefore, we need a framework that can handle angelic and demonic assumptions.

B Bundles of Operations

In this section, we introduce the concept of *bundles of operations* high level restrictions on how we want a sequence of operations to be executed. For example, we can abstract away what is important about a scheduler following a BFS strategy: atomicity of a sequence of operations. In other words, the operations generated by a smart-contract are going to be executed one after another without other smart-contracts injecting operations between them.

A *bundle* is a semantic condition (or restriction) on the execution of a sequence of operations. Instead of forcing *the whole blockchain* to use a particular execution order, we theorize on having a domain-specific language (DSL) describing how we would want to execute a set of operations. In other words, we would like to predicate on how operations are to be executed explicitly, either by assuming a BFS/DFS or other mechanisms.

B.1 Atomic Sequence

Given a sequence of operations $\langle s_0, s_1, \ldots, s_n \rangle$, we want them to be executed atomically without interleaving operations independently of the execution order followed by the scheduler. BFS schedulers respect such bundle by definition, while DFS schedulers should check that the effects generated by each s_i with $i \leq n$ does not affect the rest of the smart-contracts.

B.2 Contexts

The call and return pattern enables us to reason about units of functionality, in the sense, that when we invoke a method in a smart-contract is because we expect a result independently of how many other functions that method is invoking. When we program smart-contracts under the demonic assumption, where giving control to other (possibly unknown) smart-contract may result in an attack, we want to encapsulate their behaviour while still interacting with them to obtain some functionality.

Independently of the execution order, we can devise an encapsulation mechanism enabling us to reason about the functionality of external invocations in a *context*. The general idea is to encapsulate the execution of smart-contracts and all of its descendant operations in a *context*. Instead of having a pending queue of operations, we would have a sequence of pending queues, each one representing an encapsulated context. Operationally, each context is completely executed before passing to the next. Contexts give us the ability to invoke functions and execute them as if they were the only procedures being executed in the machine, i.e. in a completely isolated context.

▶ **Example 1.** Let A and B be two smart-contracts such that the result of executing A is two operations $[A_1, A_2]$, while the result of executing B is just $[B_1]$. Moreover, operations A_1, A_2 do not generate new operations.

Assuming we have a pending queue formed by a context invocation to A followed by a normal invocation to B, we will have the following execution sequence:

$$[[A], B] \rightsquigarrow [[A_1, A_2], B] \rightsquigarrow [[A_2], B] \rightsquigarrow [[], B] \equiv [B] \rightsquigarrow \dots$$

Implementing contexts is easy and very useful to encapsulate functionality. However, this brings some questions: how are contexts created? who creates them? From the point of view of defense programming, we have two possible answers:

Caller contextual call: upon invoking a remote procedure, the caller can specify the execution to be encapsulated in a context. This mechanism protects the callee since the new procedure cannot inject operations interleaving the ones already on the pending queue (as a DFS blockchain would do).

Callee contextual call: when invoked, the callee internally decides if its functions are to be executed in a context. This mechanism enables the function being called to assume that the pending execution queue is empty and nothing is going to modify it aside from itself or the invoked smart-contracts.

C Restricting Smart-Contracts Interaction

We implemented two kinds of restrictions: one where the blockchain enters into a mode where the smart-contract interactions are not allowed, and another where we can reduce the set of addresses that can be invoked.

End of Interactions. The executor only accepts transactions from and to the same smart-contract.

Address Universe. We can dynamically restrict the universe of addresses that smart-contracts (and their descendants) can invoke, either by restricting the known universe of addresses or by specifying addresses that cannot be invoked. In other words, we would have two sets of addresses:

Allow addresses: the set of addresses that can be invoked during execution. Invoking an address outside this set will force the transaction to fail.

Block addresses: the set of addresses that cannot be invoked during execution. Invoking one of these addresses will force the transaction to fail.

Both mechanisms suggest the addition of a shared state between a smart-contract and its descendants during the execution of smart-contracts. If we see transaction executions as trees, we can add restrictions to such tree. Moreover, we can analyze *transaction trees* to restrict or predict the behaviour of smart-contracts.