Finding smart contract vulnerabilities with ConCert's property-based testing framework

Mikkel Milo
Computer Science, Aarhus University

Eske Hoy Nielsen
Computer Science, Aarhus University

Danil Annenkov
Computer Science, Aarhus University

Bas Spitters
Computer Science, Aarhus University

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/LIPIcs.CVIT.2016.23

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.

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. 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.

[1] https://www.wired.com/2016/06/50-million-hack-just-showed-dao-human/

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:12
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

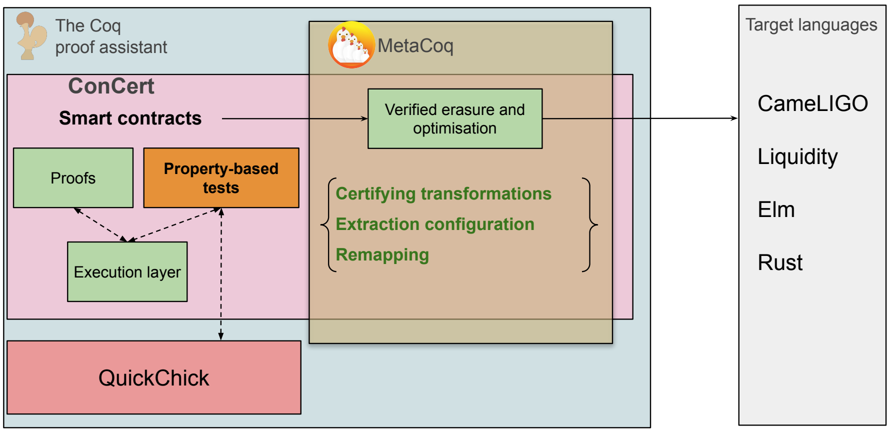

Figure 1 The pipeline

Contributions.

In this paper, we present the details of the property-based testing functionality of the ConCert framework [3, 2]. We present three case studies demonstrating how ConCert can be used to find real-world bugs in smart contracts.

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 as a function in Coq using the ConCert infrastructure. 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).

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 of only MetaCoq and the pretty-printers into the target languages.

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.[2] 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.
- `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

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

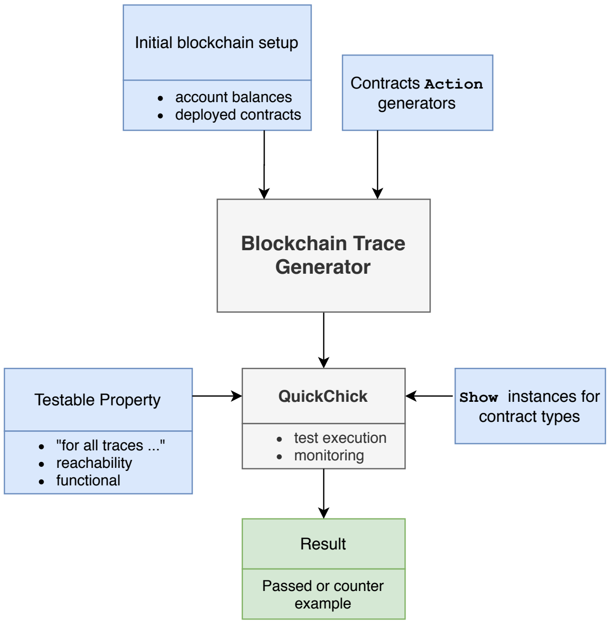

Figure 2 Property-based Testing in ConCert

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[3]. 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

[3] 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

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.

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[4]. 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

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

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[5] 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.

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[6]. 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.
```

[5] Tezos moved to depth-first execution order after Dexter was developed
[6] https://bzx.network/blog/incident

```
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 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. 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

```
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

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 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 batFund (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{...})}];
  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[7] for Ethereum or SmartPy[8] for Tezos mostly cover conventional unit testing that can be insufficient. The testing framework for LIGO[9] 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[10] use fuzzing-like techniques for testing smart contracts. The common challenge for this approach is that randomly generated transaction data might

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

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.

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[11] 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.

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[12], 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. 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 Subsection 2.3. Implementing these requires only some expertise with Coq and QuickChick, and can in some cases be derived automatically. Hence, the setup is relatively simple, only requiring little extra effort compared to writing traditional tests.

[11] https://iohk.io/en/blog/posts/2022/01/27/simple-property-based-tests-for-plutus-validators/
[12] https://github.com/AU-COBRA/ConCert/tree/master/examples/eip20

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) passed — usually in than 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.

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[13].

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.[14]

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

References

1 Guillermo Angeris, Hsien-Tang Kao, Rei Chiang, Charlie Noyes, and Tarun Chitra. An Analysis of Uniswap markets. *Cryptoeconomic Systems*, 1(1), 2021. URL: https://cryptoeconomicsystems.pubpub.org/pub/angeris-uniswap-analysis, doi:https://doi.org/10.21428/58320208.c9738e64.

2 Danil Annenkov, Mikkel Milo, Jakob Botsch Nielsen, and Bas Spitters. Extracting Smart Contracts Tested and Verified in Coq. CPP 2021. 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. FMBC19, 2019.

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.

[13] https://github.com/AU-COBRA/ConCert
[14] e.g. https://forum.cardano.org/t/cip-proposal-cardano-audit-best-practice-guidelines/100022

6 Kim Grauer, Will Kueshner, and Henry Updegrave. Chainalysis 2022 Crypto Crime Report. 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*, Feb 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.