Property-Based Testing of Smart Contract in Coq using QuickChick

Masters Thesis Defence

Mikkel Milo

Department of Computer Science
Aarhus University

June 25, 2020

AARHUS UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Table of Contents

Table of Contents

Motivation

- Smart Contracts are typically used for sensitive transaction
- Financial and legislative usage
- Once deployed, impossible to change
- Attacks:
  - The DAO: $50 million worth of ETH lost
  - Parity Wallet: $280 worth of ETH lost
  - UniSwap Microtrading Exploit: 99.5% funds lost (March 2020)

Motivation

Conclusion: need effective methods for finding bugs/vulnerabilities in smart contracts

Existing Methods

- Contract language (type system, compiler, etc.)
- Model Checking
- Formal verification
- Specialized Static Analysis/Symbolic Execution tools

Existing(?) Methods

But what about testing?

(Property-Based) Testing as a Semi-Formal Method

"Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence"[Dij72]

— Edsger W. Dijkstra, The Humble Programmer (1972)

(Property-Based) Testing as a Semi-Formal Method

```
for all inputs x, y, ...
such that precondition(x, y, ...) holds,
P(x, y, ...) is true
```

- P is a mathematical property
- Inputs `x, y, ...` are generated *arbitrarily* (using some generative func)
- Potentially thousands of test cases are generated & executed

PBT versus other testing methods

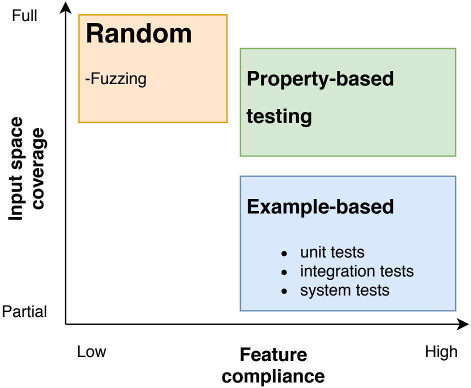


Table of Contents

Contributions

- A PBT framework for ConCert smart contracts in Coq
- Functional and temporal properties on automatically generated execution traces
- Several case studies of complex contracts
  - ERC20 Tokens, Congress, FA2 Tokens, UniSwap token exchange
  - Successfully tested many safety properties on these
  - Discovered known re-entrancy bugs and other vulnerabilities using testing

Table of Contents

A Brief Introduction to ConCert

- Blockchain & Smart Contract formalisation in Coq
- Certified extraction to Liquidity and Midlang
- Contains a certified, *executable* execution framework
- Functional contracts as Coq terms
- Contracts consist of an `init` and `receive` function

A Brief Introduction to ConCert

Contract Representation

```
Parameter (State Msg Setup : Type).
init :
  Chain →
  ContractCallContext →
  Setup →
  option State;
```

A Brief Introduction to ConCert

Contract Representation

```
Parameter (State Msg Setup : Type).
receive :
  Chain →
  ContractCallContext →
  State →
  option Msg →
  option (State * list ActionBody);
```

A Brief Introduction to ConCert

Execution Model

- Each block holds the states of all deployed contracts
- and a list of `Actions` to execute in this block
- `Actions` can be: contract calls, transfers, contract deployments
- Execution trace: a sequence of blocks where no actions failed

Table of Contents

A Motivating Example: ERC20 Tokens

- Tokens can represent any asset
- Transferable
- Widely used and backbone of many smart contracts
- ERC20: `transfer`, `transfer_from`, `approve`

A Motivating Example: ERC20 Tokens

ConCert Contract Implementation

```
Inductive Msg :=
| transfer : Address → N → Msg
| transfer_from : Address → Address → N → Msg
| approve : Address → N → Msg.

Record State := {
  total_supply : N;
  balances : FMap Address N;
  allowances : FMap Address (FMap Address N)
}.
```

A Motivating Example: ERC20 Tokens

ConCert Contract Implementation

```
Definition receive chain ctx state maybe_msg :=
  ...
  match maybe_msg with
  | Some (transfer to amount) ⇒ try_transfer ...
  | Some (transfer_from from to amount) ⇒ try_transfer_from ...
  | Some (approve delegate amount) ⇒ try_approve ...
  | None ⇒ None
  end.
```

A Motivating Example: ERC20 Tokens

Specification

What is the specification of ERC20?

- `transfer` updates balances correctly
- `transfer_from` updates balances correctly, and access control is applied correctly
- `approve` updates the allowances correctly

Can easily be stated as functional properties, and either tested or proved. All is good, then? Not quite...

A Motivating Example: ERC20 Tokens

A Vulnerability

Attack on `approve+transfer_from`:

1. Alice approves Bob for N tokens
2. Alice re-approves Bob for M ($M < N$) tokens
3. Bob notices this and transfers N of Alices tokens somewhere
4. if Bob's transaction is executed *before* Alice's, then he can now transfer another M tokens.

- Thus, Bob can transfer up to $N + M$ tokens, while Alice expected at most M tokens.
- **All ERC20 compliant tokens are vulnerable to this (in Ethereum)**

A Motivating Example: ERC20 Tokens

Safety Property Guarding against the Attack

- What is the safety property?
- Must necessarily be defined over an entire *execution trace*
- Should be able to compare state of ERC20 contract at different steps in execution trace

A Motivating Example: ERC20 Tokens

Safety Property Guarding against the Attack

In "verbatim":

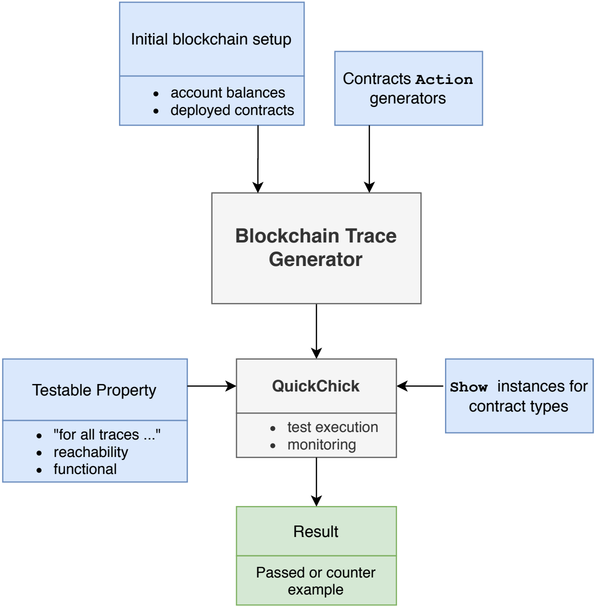
- Let S, S' be ERC20 contract states.
- If S is the result of an `approve` act P for N tokens,
- and if S' is the result of the same `approve` act but with M tokens,
- if $S \rightsquigarrow S'$
- then the delegate has transferred $\leq N$ tokens from the owner in this interval

I have stated and tested this property (QC finds a counterexample just like the attack)

Next: How the testing framework supports this

Table of Contents

Overview of the Testing Framework



Overview of the Testing Framework

Testable Properties on Execution Traces

Property	Testable interpretation
$\forall t : \mathit{Trace},\ P(t)$	Test P holds on many generated traces
$\exists c : \mathit{Chain}$, $\mathit{reachable}(c) \wedge P(c)$	Assert that the test of $\neg P(c)$ fails in some step of a generated trace. Print counterexample as witness of $P(c)$
Given a contract C, $\forall m : \mathit{Msg}$, $\{P(C,m)\}$ $C.\mathtt{receive}(m)$ $\{Q(C,m)\}$	For each generated trace, check for each step if there are messages to C satisfying P. If so, execute $C.\mathtt{receive}$ and check if Q holds.

Table of Contents

ERC20 Token Example Revisited

Implementing the generator

Composing optional generators with `backtrack`:

```
backtrack [
  (1, gTransfer      token_state) ;;
  (1, gTransfer_from token_state) ;;
  (1, gApprove       token_state) ;;
]
```

ERC20 Token Example Revisited

Implementing the generator

```
Definition gTransfer_from (state : EIP20Token.State)
                              : G (option (Address * Msg)) :=
  (allower, allowance_map) ← sampleFMapOpt state.(allowances) ;;
  (delegate, allowance)   ← sampleFMapOpt allowance_map ;;
  (receiver, _)           ← sampleFMapOpt state.(balances) ;;
  let allower_balance := with_default 0
                           (FMap.find allower state.(balances)) in
  amount ← if allower_balance =? 0
           then returnGen 0
           else choose (0, min allowance allower_balance) ;;
  returnGen (Some
    (delegate, transfer_from allower receiver amount)).
```

ERC20 Token Example Revisited

Testing the Contract

Testing a Functional property:

```
QuickChick (
  {{msg_is_transfer}}
  EIP20Token.contract
  {{post_transfer_correct}}
).
```

Testing a Reachability/Temporal property:

```
QuickChick (
  initial_chain ⇝ transfer_from_is_safe_P
).
```

Table of Contents

Conclusions

- Our approach is based on generating arbitrary blockchain execution traces
- this allows stating both functional and temporal properties
- and test interacting contracts (not shown in this presentation)
- we have sacrificed some automation to obtain the necessary performance...
- we have (re-)discovered many known vulnerabilities/bugs using our testing framework
- hence, the approach is capable, and seems effective at findings bugs
- Since the development is in Coq, we can combine testing and verification efforts (not shown in this presentation)

Future Work

- improve automation of deriving generators, e.g. using Luck[LGWH+16]
- certified generators
- shrinking for minimal counter examples
- align testable execution traces with ConCert's notion of execution traces
- integrate this work into the official ConCert repository (pull request currently under review...)

References I

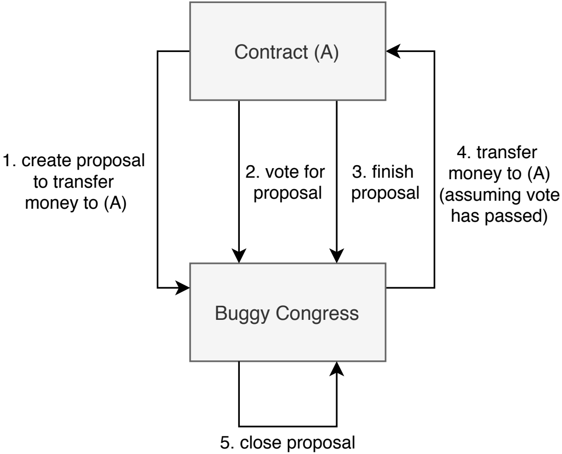
Edsger W. Dijkstra, *The humble programmer*, Commun. ACM **15** (1972), no. 10, 859–866.

Leonidas Lampropoulos, Diane Gallois-Wong, Catalin Hritcu, John Hughes, Benjamin C. Pierce, and Li yao Xia, *Beginner's luck: A language for property-based generators*, 2016.

Extras

Extra Stuff...

Congress/DAO Re-entrancy



Congress/DAO Re-entrancy

Safety Property

```
Definition cacts_preserved new_state resp_acts old_state msg :=
  num_cacts_in_state new_state + length resp_acts <=
  num_cacts_in_state old_state + proposal_cacts msg.

Lemma receive_state_well_behaved
      chain ctx state msg new_state resp_acts :
  receive chain ctx state msg = Some (new_state, resp_acts) →
  cacts_preserved new_state resp_acts old_state msg.

QuickChick (
  {{fun _ _ ⇒ true}}
  Congress_Buggy.contract
  {{receive_state_well_behaved_P}}
).
```

UniSwap Exploit

Exchange Rate Formula

1. calculate the exact exchange rate
2. send corresponding Ether to the caller
3. transfer tokens to the liquidity contract

The exchange rate formula:

$$getInputPrice = \frac{Ts \cdot 997 \cdot ETHr}{Tr \cdot 1000 + Ts \cdot 997}$$

where

$$\begin{aligned} Ts &: \text{nr. of tokens being sold by caller} \\ Tr &: \text{current token reserve held by the liquidity contract} \\ ETHr &: \text{current Ether reserve held by the liquidity contract} \end{aligned}$$

Dexter Exchange Protocol