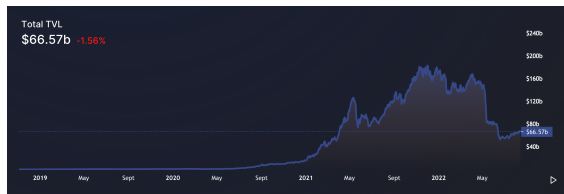


Low-overhead functional property checking for smart-contract systems

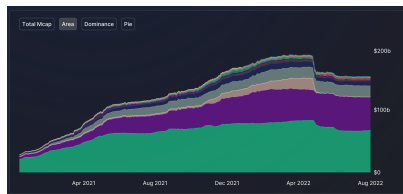
Valentin Quelquejay-Leclère
August 16th, 2022

Securing smart-contracts is critical

- Smart contracts are programs running on the blockchain. They manipulate digital assets.
- Most Ethereum smart contracts are written in Solidity, a *statically-typed* language that compiles to bytecode executed on the *Ethereum Virtual Machine*.
- The value of assets stored in smart contracts increases every day. Securing them is critical.



Ethereum Total Value Locked (TVL)






Total Stablecoins Market Cap (All chains)

Securing smart contracts is not easy

- Smart contract systems are complex.
- Smart-contract systems often interact with many other systems. They are vulnerable to common software threats but also blockchain-specific threats.
- DeFi interoperability brings a lot of power to the DeFi ecosystem but also leads to increased security risks. It enables to mount *multi-step attacks* involving several protocols.
- These threats are challenging to detect with security tools. One needs proper **specifications** to detect these **business-logic vulnerabilities**.
- Property-based testing can help detect those vulnerabilities by expressing key functional specifications of systems in **a few properties**.
- Existing techniques to check functional properties requires code instrumentation or only target program verification. Our approach enables us to check properties at runtime with **low-overhead**.

Our specification language: *Jessy*

- Jessy is a *specification language* for EVM smart contracts.
- Goal: make the language as easy and straightforward to use as possible with existing Solidity experience.
- Jessy can express functional properties that can be checked at runtime **without code instrumentation**. This enables to check properties of deployed systems using only their ABI (Application Binary Interface).
- 3 types of safety properties with different semantics:
 - Contract invariant (*invariant*)
 - CALL post-condition (*if_succeeds*)
 - CALL error post-condition (*if_aborts*)

<code>if_succeeds</code>	<code>"a is updated"</code>	<code>old(this.a()) != this.a();</code>
		
keyword	message	expression

The syntax of a Jessy property

The checker contract (1/3)

- The condition of each property is an evaluation function

$$\begin{cases} f_{c_{pre}}^{c_{post}, \mu, \sigma} : expr \longrightarrow \{true, false, \emptyset\} & \text{if } p \in \{invariant\} \\ f_{c_{pre}, S_{pre}}^{c_{post}, S_{post}, \mu, \sigma} : expr \longrightarrow \{true, false, \emptyset\} & \text{if } p \in \{if_succeeds, if_aborts\} \end{cases}$$

- The expression of a condition is a *well-defined* **typed Boolean Solidity expression**.
- CALL post-conditions may be augmented with the `old` operator to refer to *pre-state values*.
- We evaluate each property inside the EVM using a **checker contract** that we automatically generate from the *checked contract ABI* annotated with the Jessy properties.

The checker contract (2/3)

- For each invariant, we generate **a single function** that evaluates its condition.
- For `if_succeeds` properties, we generate **two functions**: `if_succeeds_pre`, evaluates the pre-state values of the `old` expressions before the call. `if_succeeds_post`, evaluates the condition of the property after the call to the function returns.
- Every function in the checker contract is a **view function**. It means the checker contract cannot modify the state of the checked contract. Additionally, we use *static calls* to invoke functions of the checker contract.

Let's look at an example for the following property of the function:

```
process(bytes _message) external returns (bool ret_0)
```

```
if_succeeds "the message is proven" old(this.messages(keccak256(_message))) != this.  
    LEGACY_STATUS_NONE();
```

The checker contract (3/3)

This is the **checker contract** we generate for the property:

```
// some parts are omitted
interface IChecked {
    //solidity interface to interact with the checked contract
}
contract Checker {
    IChecked private _checked;
    constructor(address checked) {
        _checked = IChecked(payable(checked));
    }
    function if_succeeds_pre_0(bytes memory _message, txParams calldata params) external view returns (
        bytes32 _messages_0) {
        (_messages_0) = _checked.messages(keccak256(_message));
    }
    function if_succeeds_post_0(bytes32 old_messages_0, bytes memory _message, bool ret_0, txParams
        calldata params) external view returns (bool) {
        bytes32 LEGACY_STATUS_NONE_0 = _checked.LEGACY_STATUS_NONE();
        return old_messages_0 != LEGACY_STATUS_NONE_0;
    }
}
```

Language semantics

- Each property has a **well-defined semantics**. It defines execution points in the smart contract where the property is expected to hold.
- For contract invariants, we adapt the **visible states semantics**. We must ensure invariants hold **whenever the contract is visible**.
- `if_succeeds` properties specify post-conditions that should hold **on normal termination of calls**. They might refer to *pre-state* expressions evaluated before the call (using `old`). However, properties are checked in the *post-state* of the call.
- `if_aborts` properties specify *error pre-conditions* that should hold **if a call throws an error and the transaction reverts**. Properties are checked after the call reverts, but the expression is evaluated before the call.

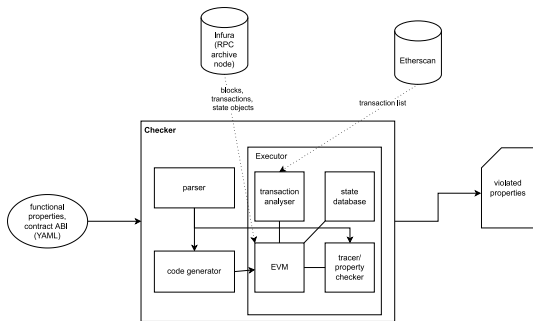
Our approach to monitoring properties

We build a prototype tool that uses the official EVM implementation from *go-ethereum* (geth).

The tool replays transactions from mainnet in a local "fork" (copy) of the blockchain.

We leverage the **tracing feature** of geth EVM to check properties at runtime. Our tracers maintain a **callstack** and check properties by calling the **checker contract** at the right execution points.

1. Parse the Jessy properties from the user-provided YAML file.
2. Generate the checker contract.
3. Deploy the checker contract on the local EVM.
4. Replay transactions from mainnet.
5. Check properties and output results.



Architecture of the tool

Overall tool architecture

The prototype tool is written in *Go*. It is composed of 4 main components:

- The **parser** parses the Jessy properties contained in the YAML file provided as input.
- The **generator** generates the **checker contract** from the parsed properties according to the rules we described. It also generates the required Solidity interfaces for interacting with the checked contract and external contracts. It compiles the checker contract using *solc*, the official Solidity compiler, and outputs a bytecode that we deploy in our local fork.
- The **transaction simulator** fetches transactions and blockchain state from an *RPC archive node* (Infura) through HTTP. It executes transactions locally by mirroring the Ethereum state.
- The **property checker** receives callbacks from the EVM. It evaluates pre-state values and checks the properties according to their semantics. It reports violated properties at the end of each transaction.

A few details

The main performance bottleneck is fetching the blockchain state while executing the transactions. We try to improve performance by:

1. Minimize the data we fetch and maximize usage of the local cache.
2. Bundle the calls to the RPC node to reduce the number of requests.
3. Pre-fetch as much data as possible in parallel.

We minimize the data we fetch by only running blocks where there is at least one transaction to monitor. Running non-consecutive blocks requires us to implement a smarter caching strategy than our initial strategy.

Remark

A way to drastically increase performances is to connect to a local archive node, either through WebSocket or through an IPC socket (if the node runs on the same machine).

Research questions

In this thesis, we try to answer the following research questions:

- What kind of functional properties are important to capture in smart-contract systems?
- How to design a specification language that expresses blockchain-specific smart contract properties that can be checked at runtime without instrumentation?
- How to design a low-overhead checker that checks blockchain-specific functional properties at runtime on non-instrumented code?
- Is it possible to monitor and mitigate vulnerabilities at runtime and limit the impact of a potential exploit on smart contract systems?

More specifically, our evaluation focuses on answering the two following questions:

- Is Jessy, our specification language, expressive enough to catch hacks that happened in the past?
- What extensions of the language could be useful?

Experimental setup

1. We compile a list of **major Ethereum exploits** that occurred in the last two years and led to high monetary losses (several million USD worths of assets). We discard all the attacks that do not result from smart contract vulnerabilities.
2. We select a list of eight exploits whose root cause seems related to a **business-logic issue in the Solidity code**, focusing on recent attacks.
3. For each of these attacks, we study the protocol documentation to understand the main specifications. We also *study exploits* by analyzing attackers' transactions and the code of the vulnerable contracts.
4. We **write functional properties** to capture the root vulnerabilities.
5. We **run the tool** before the attack to check that properties hold. We also check that when the attack occurs, the properties are reported as being violated.

We manage to express interesting properties

We manage to write a property that captures the vulnerability in 5 out of the 8 cases we study. This **supports our language design decisions** and shows that it is possible to express interesting functional properties using our language, **relying only on a contract ABI**.

Example: *Nomad bridge* is a bridge protocol which was attacked on August 2, 2022. \approx \$190M were stolen.

The attack exploits a **logic bug** in one of the core contracts in charge of processing cross-chain messages if they are valid.

An upgrade of the contract (in May) broke some checks and enabled the attacker to process messages that were not proven

Nomad properties

We manage to capture Nomad attack with the following functional properties expressed in Jessy. We write the properties for the `process` function that has the following signature:

```
function process(bytes memory _message) public returns (bool _success)
```

```
if_succeeds "the message is proven" old(this.messages(keccak256(_message))) != this.  
    LEGACY_STATUS_NONE();
```

What we can improve

For the attacks that we do not manage to capture using properties, we realize that the following **language improvements** could help us catch similar bugs:

- **information flow properties** could help to express security policies. For instance, there should be no path to call a contract A from a given function f of contract B .
- Adding support for some **temporal properties**. For instance, a "once" property which asserts that a function is called at most once after a given condition occurs.
- Adding support for **hyper-properties** would help by enabling us to compare the traces of multiple executions.

To conclude

- Several tools based on different techniques have been developed for securing smart contracts. These tools face different challenges with different objectives and strengths/weaknesses. **Property-based testing** provides a good cost-efficiency trade-off and can detect business-logic issues.
- Our approach to functional property checking **does not require instrumenting the code**. This reduces the effort needed to check properties and enables to check properties on deployed contracts using only their ABI.
- Our evaluation confirms that **our approach can benefit smart-contract security**. It also provides insights into language extensions that would be helpful for detecting additional bugs.
- **Future work** could focus on improving the language by adding new types of properties or improving existing constructs, e.g, by adding support for reading private contract state.

Acknowledgments

I would like to thank everyone that made this thesis possible.
In particular, my supervisor **Valentin Wüstholtz** and my advisor **Matthias Payer**.
My co-supervisors, **Joran Honig** and **Dimitar Bounov**.
All the other amazing people in the Diligence team ♡.

Questions ?



Low-overhead functional property checking for smart-contract systems

Valentin Quelquejay-Leclère

valentin.quelquejay@pm.me

Consensus Diligence

Zürich, August 16th, 2022

Additional material

```

contract Bar {
// invariant "invariant: a is 51" this.a() == 51;
uint256 public a;

constructor() {
    a = 51;
}

function increment() private {
    a += 1;
}

function decrement() private {
    a -= 1;
}

// if_succeeds "kind: a is still the same" old(this.a()) == a_ && this.a() == a_;
function kind() public returns (uint256 a_) {
    //invariant not violated
    increment();
    decrement();
    a_ = a;
}

// if\_aborts "mean: a is 53" this.a() == 53;
function mean() public {
    // violate invariant
    decrement();
    decrement();
    require(a != 51, "a is not 51");
}
}

```

```

function process(bytes memory _message) public returns (bool _success) {
    // ensure message was meant for this domain
    bytes29 _m = _message.ref(0);
    require(_m.destination() == localDomain, "!destination");
    // ensure message has been proven
    bytes32 _messageHash = _m.keccak();
    require(acceptableRoot(messages[_messageHash]), "!proven");
    // check re-entrancy guard
    require(entered == 1, "!reentrant");
    entered = 0;
    // update message status as processed
    messages[_messageHash] = LEGACY_STATUS_PROCESSED;
    // call handle function
    IMessageRecipient(_m.recipientAddress()).handle(
        _m.origin(),
        _m.nonce(),
        _m.sender(),
        _m.body().clone()
    );
    // emit process results
    emit Process(_messageHash, true, "");
    // reset re-entrancy guard
    entered = 1;
    // return true
    return true;
}

```

```
function acceptableRoot(bytes32 _root) public view returns (bool) {  
    // this is backwards-compatibility for messages proven/processed  
    // under previous versions  
    if (_root == LEGACY_STATUS_PROVEN) return true;  
    if (_root == LEGACY_STATUS_PROCESSED) return false;  
  
    // BUG: _root == 0x0 ==> _time = confirmAt[0x0] = 1  
    // (intialized at contract creation)  
    uint256 _time = confirmAt[_root];  
    if (_time == 0) {  
        return false;  
    }  
    return block.timestamp >= _time;  
}
```