Program Analysis & Blockchain Security

CS 243
May 17, 2022

David Tarditi
VP of Engineering CertiK
Lecture Goals

- Compilers and program analysis are about reasoning about programs.
- Show how what you are learning applies in practice to blockchain security.
- Provide a perspective on working on compilers and language implementation.
My Background

- Ph.D. in compilers/programming languages Carnegie Mellon (1997)
  - Typed intermediate languages for optimizing ML
- 25 years at Microsoft
  - Compilers for Java, C#, C++, and frameworks for compilers
  - OS written in type-safe languages.
  - Extension to C to make it type-safe
- Now VP of Engineering at CertiK
  - A leading blockchain security company.
  - 1700+ audits of blockchain projects in 2021
  - I lead the tools effort.
Outline

- Overview of blockchain and programmable blockchain
- Securing blockchain
  - Motivation
  - Common vulnerabilities in blockchain programs
- Approaches for finding vulnerabilities in programs
- Perspectives
- Summary
Blockchain – What Is It About?

- A technology for securely representing money (value) digitally.
- At its core,
  - A distributed ledger that maps accounts to values (integers).
  - The ledger can be the storage for a virtual machine (computer)
  - The virtual machine runs programs ("smart contracts") that operate on digital money (or items of value).
- This has created a lot of excitement about the possibilities.
Distributed Ledger

- Ledger has account data and transaction list.
- Replicated across network of computers.
- Proposed transactions broadcast to network.
- Consensus algorithm for resolving conflicts.
Programmable Blockchain

- **Ethereum** started in 2013 by Vitalik Buterin
- Ethereum Virtual Machine (EVM) combines blockchain ledger and a stack-based virtual machine
- Every computer (node) in the Ethereum network replicates the state of this VM.
- Every node runs all transactions in a block as part of the consensus mechanism
  - This is a “global universal computer”.
- Many programmable blockchains now.
Ethereum VM architecture

Runs smart contracts.
Charges Ether (gas) for each VM instruction executed.
Gas supplied up front.

Solidity Programming Language

Domain-specific language for EVM programming (https://soliditylang.org)

Object-oriented, follows JavaScript syntax, statically typed:

- Two location kinds: storage and memory. Storage is permanent. Memory is not.
- New type: address, a location in the ledger (blockchain).
  - Every address has an Ether balance associated with it.
  - Two kinds of addresses: externally-owned or for a contract. Contracts have additional growable storage associated with them.
  - Every address has a “send” method on it for sending Ether to it.
  - 160-bit integer

- Standard types: signed/unsigned integers (8-256 bits, default is 256), boolean, structs, enums, maps, static/dynamically-sized arrays, string.
Solidity, Continued

- A contract is like a class:
  - Defines a new type.
  - Has functions and state variables associated with it. The data is persistent and stored in storage.
- Instances of contracts live in the blockchain: new EVM bytecode is created when a contract is deployed and storage is allocated.
- Messages used for communication:
  - An external account sends a contract a message to invoke it and start a transaction.
  - Contracts can send messages to other contracts or external accounts on the chain.
- Transactional: “all or nothing”
  - If something goes wrong, changes are reverted.
contract Coin {
    address public minter;
    mapping (address => uint) public balances;

    // Constructor code run when the contract is created
    constructor() { minter = msg.sender; }

    // Sends newly created coins to an address. Can only be called by the contract creator
    function mint(address receiver, uint amount) public {
        require(msg.sender == minter);
        balances[receiver] += amount;
    }

    // Sends an amount of existing coins from any caller to an address
    function send(address receiver, uint amount) public {
        if (amount > balances[msg.sender])
            revert ...
        balances[msg.sender] -= amount;
        balances[receiver] += amount;
        emit Sent(msg.sender, receiver, amount);
    }
}
ERC-20 Token Standard

Standard interface for tokens. Semantics specified informally.

function name() public view returns (string)
function symbol() public view returns (string)
function decimals() public view returns (uint8)
function totalSupply() public view returns (uint256)
function balanceOf(address _owner) public view returns (uint256 balance)
function transfer(address _to, uint256 _value) public returns (bool success)
function transferFrom(address _from, address _to, uint256 _value) public returns (bool success)
function approve(address _spender, uint256 _value) public returns (bool success)
function allowance(address _owner, address _spender) public view returns (uint256 remaining)

https://eips.ethereum.org/EIPS/eip-20
Solidity and Ethereum have some sharp edges

Solidity
- Order of evaluation not specified for operands.
- You can omit return statements.
- Code is immutable, but indirect jumps are allowed.

Ethereum
- Reentrancy attacks.
  - Contract 1 partially changes state variables.
  - Contract 1 calls Contract 2
  - Contract 2 is malicious and calls back into Contract 1.
- Contracts can be destroyed.
- There is no VM-level type system enforced contracts
Outline

- Overview of blockchain and programmable blockchain
- **Securing blockchain**
  - Motivation
  - Common vulnerabilities in blockchain programs
- Approaches for finding vulnerabilities in programs
- Perspectives
- Summary
Blockchain & Smart Contract Vulnerabilities

Total Value Lost to Hacks and Exploits 2021

- $0.00
- $500,000,000.00
- $1,000,000,000.00
- $1,500,000,000.00

Time:
- 3/1/2021
- 5/1/2021
- 7/1/2021
- 9/1/2021
2021 / 2022 Notable Hacks

April 2021
- Uranium Finance hacked for $50 million
  *Omission of a single character (lost a decimal place)*

October 2021
- Compound Finance gives out $90 million in unearned rewards
  *A 1-character mistake ‘>’ instead of ‘>=’*

November 2021
- DeFi protocol bZx hacked for $55 million
  *Developer phished by emails -> mnemonic seed phrase*

2022 (to date)
- Qubit Finance, Wormhole, Meter.io, Ronin
  *More than $1 billion*
Securing Blockchain & Web3 Projects

Hackers go after the easiest targets first:

- Socially engineer users.
- Attack the web UI front-end.
- Compromise private keys/passwords (“operational security”).
- Find flaws in smart contracts.
- Target blockchains or bridges across blockchains
Identifying Common Code Vulnerabilities

- Certik is the leading blockchain security company.
- We provide source code audits and verification ranging from smart contracts to blockchain implementations.
- We audited source code for **1,427 projects** from August 2021 to February 2022.
- We analyzed the types of issues found during these audits.
  - Focused on critical, major, and medium severity.
How Do CertiK Source-Code Audits Work?

- Project team provides source code (ideally before publishing).
- CertiK security experts review code and provide written report with issues.
- Project team and security experts discuss report; team fixes or acknowledges issues.
- CertiK publishes final report at certik.com
Top Smart Contract Vulnerabilities

In the 1,427 audits found 16,400 issues. **5,300** were critical, major, or medium severity.

<table>
<thead>
<tr>
<th>Vulnerability</th>
<th>Example problem</th>
<th># of occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Centralization risks</td>
<td>Owner can update rewards balances</td>
<td>3,066</td>
</tr>
<tr>
<td>2. Logical/correctness issues</td>
<td>Incorrect calculation of rewards</td>
<td>1,209</td>
</tr>
<tr>
<td>3. Withdrawal issues</td>
<td>Money locked in contract</td>
<td>142</td>
</tr>
<tr>
<td>4. Access control</td>
<td>Anyone can call mint</td>
<td>120</td>
</tr>
<tr>
<td>5. Lacks limits/bounds</td>
<td>No limit on fees</td>
<td>96</td>
</tr>
</tbody>
</table>
Centralization Risks

**Definition**: a privileged role (such as the owner) has control over one or more sensitive contract operations. The role is an overprivileged account.

**Examples**
- Update user account balances without any restrictions.
- Withdrawals
  - Send value locked in contract to an arbitrary address.
  - Withdraw all rewards.
- Initial token distribution: all tokens sent to the contract deployer or EOA.
- Contract upgrade.
- Setting key parameters of the contract without limits.

You have to trust the privileged role, not the code.
Example code

Withdrawal centralization risk: the owner can withdraw all users' reward tokens.

```solidity
/*
 * @notice Stop rewards
 * @dev Only callable by owner. Needs to be for emergency.
 */

function emergencyRewardWithdraw(uint256 _amount) external onlyOwner {
    rewardToken.safeTransfer(address(msg.sender), _amount);
}
```

Initial distribution risk: all the tokens initially belong to the owner of the contract.

```solidity
_isExcludedFromFee[_msgSender()] = true;
_isExcludedFromFee[address(this)] = true;
_balances[_msgSender()] = _tTotal;
emit Transfer(address(0), owner(), _tTotal);
```
## Fixing or Mitigating Centralization Risks

<table>
<thead>
<tr>
<th>Threat</th>
<th>Consequence</th>
<th>Fixes + Mitigations to Avoid This</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leak private key for privileged account</td>
<td>Hacker now has privileged power.</td>
<td>● Remove functionality or implement in code.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>● Renounce privileged role.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>● Secure private key using multisig wallet (2 of 3, 3 of 5) + use time-lock.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>● Use a DAO for community approval.</td>
</tr>
<tr>
<td>Rug pull / Internal Operations</td>
<td>Token holders lose all their money</td>
<td>● Remove functionality or implement in code.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>● Renounce privileged role.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>● KYC owners (without being doxxed).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>● Use a DAO for community approval.</td>
</tr>
</tbody>
</table>
Outline

- Overview of blockchain and programmable blockchain
- Securing blockchain
  - Motivation
  - Common vulnerabilities in blockchain programs
- Approaches for finding vulnerabilities in programs
- Perspectives
- Summary
Approaches

Can use widely-used approaches for finding vulnerabilities and issues:

- **Syntactic approaches**
- **Dataflow analysis**
- Fuzzing
- Symbolic execution
- **Model checking**
- Deductive program verification
Syntactic approach: code query languages

Use approach inspired by CodeQL (codeql.github.com)
- Build a graph that represents the program (the AST, control flow, data flow)
- Use a relational database (datalog) to represent the graph information
- Use a specially-designed query language to identify code with issues

Pluses:
- Easier to write queries – don’t have to be a compiler writer

Minuses:
- Queries can be slow to run.
- Expressivity limits: no fixed points, no lattices/equations like in dataflow analysis, no abstract domains like in abstract interpretation.
Syntactic approach: code query languages

Example hierarchy (simplified):

- Project
  - Files
    - Pragmas
    - Structs
    - Contracts
      - Function with signatures
        - Functions
          - Statements
            - Expressions
              - Operators
            - Calls
        - State variables
          - TypeName
        - Libraries
        - Interfaces
Internal function exposed publicly

Naming convention that function beginning with `/_` is internal.

This can easily trip up an unwary programmer:

```solidity
function _takeDev(uint256 tDev) public {
    _tOwned[address(this)] = _tOwned[address(this)].add(tDev);
}
```

Can find it with this pattern:

```sql
SELECT DISTINCT funcSig fileName startLine
WHERE {
    funcSig function.signature func .
    # location info
    func function.loc loc .
    loc location.file fileName .
    loc location.start startInfo .
    startInfo location.line startLine .

    func function.visibility "public"
    # function signature starts with "/_"
    FILTER regExMatch(funcSig, ".+_")
}
```
Approaches

Can use widely-used approaches for finding vulnerabilities and issues:

- Syntactic approaches
- **Dataflow analysis**
- Fuzzing
- Symbolic execution
- Model checking
- Deductive program verification
Dataflow analysis-based approaches

Slither is a static analysis framework for Solidity. Available at https://github.com/crytic/slither.
- Converts Solidity to an IR with a CFG and 3-address form.
- For some analyses, converts that IR to SSA form.
- Computes data dependency information (for taint analysis), read/writes of variables, class hierarchy information, and runtime access control information.

Detectors in slither

Slither has about 80 detectors, falling into the following categories:

- Reentrancy attacks.
- Uses of external (possibly attacker-controlled) addresses or values for transfer operations.
  - This is where taint analysis is used.
- Logical errors
  - Unprotected critical operations such as ability to destroy a contract.
  - Uninitialized storage (operations on storage are expensive).
- Simple mistakes allowed by the language or older versions
  - Unspecified order of operand evaluation, missing return statements, and so on.

80 sounds like a lot, but a lot of them are rarely used in practice.
Example detector

Functions that send Ether to arbitrary destinations:
Unprotected call to a function sending Ether to an arbitrary address.

```solidity
contract ArbitrarySendEth {
    address destination;
    function setDestination(){
        destination = msg.sender;
    }

    function withdraw() public{
        destination.transfer(this.balance);
    }
}
```
Challenges with syntactic and dataflow approaches

We grade issues by potential severity, based on impact.

Determining *impact* is often hard

- The desired semantics of the contract is not known.
- You need context: is a “bad” coding pattern really a bug?
- What could be impacted? Value? Control of the contract? Contract upgrade?
- Even a simple calculation or logical error could be catastrophic

False positives
Approaches

Can use widely-used approaches for finding vulnerabilities and issues:
- Syntactic approaches
- Dataflow analysis
- Fuzzing
- Symbolic execution
- **Model checking**
- Deductive program verification
Model Checking Smart Contracts

**Symbolic model checking**
Automatically prove or disprove that all possible executions of a program satisfy a logical formula:
- Attempts to construct a mathematical proof of the formula w.r.t. the program's semantics.
- If proof construction fails, generates a counterexample.

"Program testing can be used to show the presence of bugs, but never to show their absence!" (Edsger W. Dijkstra)

**Difference to testing**
- Model checking reasons about a model of the program without ever executing it.
- Successful proofs hold for all possible executions. ⇒ No cases (not even corner cases) can be missed.

**Fundamental limitation**
Automated reasoning about programs is generally undecidable. ⇒ An unknown result can occur
Formalizing Properties

**Linear temporal logic (LTL)**
- Reasons about sequences of discrete steps (traces).
- Formula defines set of admissible traces.
- **Safety**: Something bad never happens (e.g. overflow, disallowed withdrawal)
- **Liveness**: Something good eventually happens (e.g. receive refund from a failed crowdsale)

Let $AP$ be a finite set of atomic propositions, $p \in AP$. LTL formulas $\varphi, \psi$ are constructed as follows:

$$\varphi, \psi ::= p \mid \neg \varphi \mid \varphi \lor \psi \mid \varphi U \psi \mid \Box \varphi.$$ 

Let $\pi \in (2^{AP})^\omega$ be a trace, $i \in \mathbb{N}$. Then

$$\pi, i \models p \iff p \in \pi[i],$$
$$\pi, i \models \neg \varphi \iff \pi, i \not\models \varphi,$$
$$\pi, i \models \varphi \lor \psi \iff \pi, i \models \varphi \text{ or } \pi, i \models \psi,$$
$$\pi, i \models \Box \varphi \iff \pi, (i+1) \models \varphi, \text{ and}$$
$$\pi, i \models \varphi U \psi \iff \exists k \geq i. (\pi, k \models \psi \land \forall j. i \leq j < k \Rightarrow \pi, j \models \varphi).$$

Blockchain evolution as a trace:

$$\pi \in (2^{AP})^\omega = \{ \text{call(deposit(13))} \}, \{ \text{deposit(7)} \mid \text{balance=13} \}, \{ \text{withdraw(20)} \mid \text{balance=20} \}, \{ \text{withdraw(20)} \mid \text{balance=20} \}, \{ \text{deposit(1)} \}, \{ \text{withdraw(1)} \}, \ldots$$
Example Tool: SmartPulse

SmartPulse tool:
- SmartLTL as a property specification logic (similar to standard LTL) for smart contracts.
- Use Ultimate Automizer as backend engine: Full LTL model checking, i.e. safety, liveness, and combinations.

Verifying ERC-20 Token Properties

Example properties:

- transfer-normal: Transferred amount is credited to recipient and deduced from sender, if sender's balance suffices
- transfer-self: Transferring money to oneself does not change one's balance
- transfer-zero-address-fail: Transfers to the zero address revert
- transfer-fail: Revert if amount exceeds sender's balance or leads to overflow in recipient's balance.
- transfer-self-fail: Any transfer to oneself with an amount that exceeds one's balance reverts.

LTL formula for transfer-from

Invocations of the `transferFrom(a, b, amt)` function must fail whenever

- the requested amount `amt` exceeds the available balance of `a`, or
- the requested amount `amt` exceeds the allowance of the sender over the tokens belonging to `a`, or
- crediting the tokens to the balance of the recipient `b` would cause an overflow.

(formula simplified to ignore constraints on integer ranges)

\[
\psi = \Box \left( \text{started}(\text{transferFrom}(a, b, \text{amt})) \land a \neq b \land \neg \Box \left( \text{reverted}(\text{transferFrom})) \right) \right)
\]

\[\land (\text{amt} > \text{balances}[a]) \land (\text{amt} > \text{allowances}[a][msg.sender]) \land (\text{balances}[b] + \text{amt} \geq 2^{256}) \]

\[\rightarrow \Box \left( \text{reverted}(\text{transferFrom})) \right) \]
Model checking challenges

- No formal specifications of expected behavior.
- Non-linear arithmetic is common in financial calculations.
- 3 results: true, false, timeouts.
- Explaining causality.
- Multiple transaction attacks (state space can explode)
  - Transaction 1: create bad state
  - Transaction N: use bad state
Example of challenges: SafeMoon token

SafeMoon is an ERC20 token contract with 3 additions:
1. Automated liquidity acquisition (5% fee when you sell)
2. Token reflection (5% fee when you sell):
3. Depreciating supply & burn address

Verifying SafeMoon
1. Relevant properties most likely contract invariants (not pre/post-conditions).
   Requires reasoning about multiple transactions
2. Financial model influences almost all functions
3. Non-linearities are on critical paths in all central functions:
   tokenFromReflection, setMaxTxPercent, _getRValues, _getTValues, _getValues, _getRate, and so on.
Non-linearity in SafeMoon

The _getRValues and _getRate functions are central to all computations in SafeMoon. They contain non-linear multiplications of operands of type uint256.

```solidity
function _getValues(uint256 tAmount) private view returns (uint256, uint256, uint256, uint256, uint256, uint256) {
    (uint256 tTransferAmount, uint256 tFee, uint256 tLiquidity) = _getTValues(tAmount);
    (uint256 rAmount, uint256 rTransferAmount, uint256 rFee) = _getRValues(tAmount, tFee, tLiquidity, _getRate());
    return (rAmount, rTransferAmount, rFee, tTransferAmount, tFee, tLiquidity);
}

function _getTValues(uint256 tAmount) private view returns (uint256, uint256, uint256) {
    uint256 tFee = calculateTaxFee(tAmount);
    uint256 tLiquidity = calculateLiquidityFee(tAmount);
    uint256 tTransferAmount = tAmount.sub(tFee).sub(tLiquidity);
    return (tTransferAmount, tFee, tLiquidity);
}

function _getRValues(uint256 tAmount, uint256 tFee, uint256 tLiquidity, uint256 currentRate) private pure returns (uint256, uint256, uint256) {
    uint256 rAmount = tAmount.mul(currentRate);
    uint256 rFee = tFee.mul(currentRate);
    uint256 rLiquidity = tLiquidity.mul(currentRate);
    uint256 rTransferAmount = rAmount.sub(rFee).sub(rLiquidity);
    return (rAmount, rTransferAmount, rFee);
}

function _getRate() private view returns(uint256) {
    (uint256 rSupply, uint256 tSupply) = _getCurrentSupply();
    return rSupply.div(tSupply);
}
```
Constant factor automated market makers (CFAMM)

A constant factor AMM maintains two reserves of different tokens, its liquidity pool. Let \( x \) be the number of tokens in pool A and \( y \) be the number of tokens in pool B; further, let \( k = x \times y \).

Let \( x' \) and \( y' \) be the corresponding numbers of tokens after an exchange took place. A constant factor market maker guarantees that after any exchange, it holds \( k = x' \times y' \).

→ This invariant of a swap can be specified as a property and verified using model checking.
→ Requires non-linear arithmetic (product of two variables):
  Computationally hard.
Outline

- Overview of blockchain and programmable blockchain
- Securing blockchain
  - Motivation
  - Common vulnerabilities in blockchain programs
- Approaches for finding vulnerabilities in programs
- Perspectives
- Summary
Challenges/open problems in blockchain analysis

- Making it easy for programmers who aren’t experts to write domain-specific checks.
- Lack of specification of intended behavior.
- Catching design-time bugs
- Diversity of area:
  - Multiple programming languages for smart contracts. Solidity, Rust, Haskell.
  - Solidity comes in different flavors.
  - Different blockchains with different programming models
Developing secure and reliable software

Other areas have similar software development problems to blockchain:
- IoT, cloud services, autonomous systems, secure systems
- Example problems: authorization, information flow, reasoning about math, specifying intended behavior, reasoning about whether system meets that.
- The techniques from program analysis, verification, compilers, programming languages apply there too.

We can use blockchain to advance the state of the art in software development.
- We see a complete shift away from unsafe languages like C++, a willingness to experiment, and high stakes.
- The value of security is very apparent here and everything is transparent.
Career observations

“Reasoning about programs” is a specialized area growing steadily over the years.

Examples: PLDI sponsors, LLVM conference

Why?
- Important problems.
- Significant technical advances.
Summary

- Ideas from programming languages appear throughout the blockchain world.
- We can apply the full range of program analysis techniques to help people secure blockchain programs.
- The underlying problems for securing blockchain programs are similar to problems in other domains.
- There are some interesting problems to work on.
- “Reasoning about programs” is a steadily growing area.
Formal Verification Of Smart Contracts

Verisol (from Microsoft): Solidity + invariants specified separately.
- Automated checking of invariants using Z3 SMT solver.

DeepSEA (from Certik): Programming language designed for verification.
- Certified compiler that produce WASM bytecode.
- Extends the CompCert

Citations: