Verification of Solana Programs

Jorge A. Navas (Certora) and Arie Gurfinkel (University of Waterloo and Certora)

Venice, May 26th 2023
Symposium on Challenges of Software Verification (CSV)
DeFi in one slide

- Economic process completely defined by code
- Fairly complex code
- Examples
  - Lending
  - Exchange
  - Options
  - Auctions
- 50 Billion dollars in the bear market
Interesting DeFi Bugs 2022/3

- **Euler Finance $200M** – DonateToReserves() function didn’t check for account debt health, allowing for bad debt to accrue and for the collateral to be liquidated at a large discount to the attacker.
- **Yearn Finance V1 $10M** – Misconfiguration of one of the underlying asset addresses in the USDT pool allowed an attacker to drain the whole vault.
- **Safemoon $9M** – Upgraded contract didn’t use access control for the burn() function. The attacker burned tokens from the Safemoon pool on a DEX, inflated the price and sold tokens into the pool.
- **Platypus $8.5M** – EmergencyWithdraw() didn’t check for debt, so the attacker could take max loan for his collateral, and then simply emergency withdraw the collateral.
- **Hundred $7.4M** – “First depositor” bug where the attacker could manipulate the exchange rate and borrow way more than allowed.
Why Formally Verify DeFi?

❗ Code is law
💰 Billions of dollars at stake
Σ Code is typically medium-size/modular
🐛 But bugs are hard to find
Happens in rare scenarios
🔄 New code is produced frequently
The Certora Approach: Automatic Formal Verification

- Specification
- Code

Certora Prover

- Proofs that the spec holds
- Unknown Timeout
- A hard to find behavior which violates the invariants
Critical Bugs Found by Certora Prover

Solvency
- If everybody runs to the bank, the bank still fulfills all commitments.

- Bugs prevented by the Certora-Prover **missed in manual audits** by top auditors.

Users’ money cannot be locked or lost.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Version</th>
<th>Total Value</th>
<th>Value 1</th>
<th>Strategy</th>
<th>Version</th>
<th>Total Value</th>
<th>Value 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>SushiSwap</td>
<td></td>
<td>$807M</td>
<td></td>
<td>AAVE</td>
<td>V3</td>
<td>$6.5B</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>V2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Compound</td>
<td>V2</td>
<td>$2.7B</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Balancer</td>
<td>V2</td>
<td>$1.18B</td>
<td></td>
</tr>
</tbody>
</table>

"We thank all contributors who made this release possible. Special thanks goes to @johnadtoman of @CertoraInc for reporting the inline assembly memory side effects bug!"
Why Formally Verify Solana (https://solana.com)?

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>Protocols</th>
<th>1d change(TVL)</th>
<th>1w change(TVL)</th>
<th>1m change(TVL)</th>
<th>TVL</th>
<th>Mcap @</th>
<th>Mcap/TVL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ethereum ETH</td>
<td>810</td>
<td>↑ 0.38%</td>
<td>↑ 3.76%</td>
<td>↑ 12.83%</td>
<td>$51.48B</td>
<td>$220.16B</td>
<td>4.28</td>
</tr>
<tr>
<td>2</td>
<td>Tron TRON</td>
<td>32</td>
<td>↑ 0.05%</td>
<td>↑ 2.99%</td>
<td>↑ 2.89%</td>
<td>$5.40B</td>
<td>$9.26B</td>
<td>1.16</td>
</tr>
<tr>
<td>3</td>
<td>BSC BRB</td>
<td>612</td>
<td>↑ 0.19%</td>
<td>↑ 2.04%</td>
<td>↑ 7.26%</td>
<td>$5.21B</td>
<td>$48.51B</td>
<td>9.31</td>
</tr>
<tr>
<td>4</td>
<td>Arbitrum ARB</td>
<td>337</td>
<td>↑ 0.18%</td>
<td>↑ 1.76%</td>
<td>↑ 2.59%</td>
<td>$2.68B</td>
<td>$1.48B</td>
<td>0.55</td>
</tr>
<tr>
<td>5</td>
<td>Polygon MATIC</td>
<td>426</td>
<td>↑ 0.89%</td>
<td>↑ 2.55%</td>
<td>↑ 12.48%</td>
<td>$1.21B</td>
<td>$7.86B</td>
<td>6.49</td>
</tr>
<tr>
<td>6</td>
<td>Optimism OP</td>
<td>144</td>
<td>↑ 0.94%</td>
<td>↑ 4.01%</td>
<td>↑ 11.16%</td>
<td>$1.00B</td>
<td>$540.39M</td>
<td>0.54</td>
</tr>
<tr>
<td>7</td>
<td>Avalanche AVAX</td>
<td>318</td>
<td>↑ 0.47%</td>
<td>↑ 2.62%</td>
<td>↑ 15.41%</td>
<td>$979.23M</td>
<td>$4.95B</td>
<td>5.05</td>
</tr>
<tr>
<td>8</td>
<td>Solana SOL</td>
<td>114</td>
<td>↑ 0.53%</td>
<td>↑ 2.55%</td>
<td>↑ 16.62%</td>
<td>$514.09M</td>
<td>$8.17B</td>
<td>15.0</td>
</tr>
</tbody>
</table>

https://coinmarketcap.com/chain-ranking
Why Formally Verify Solana?

● Benefits:
  ○ Based on general purpose programming languages: Rust, C/C++
  ○ Reusing existing eBPF virtual machine:
    ■ Support multiple (or even combination of) input languages
  ○ Programs are stateless: all data is passed as function arguments
    ■ Non-interference (easier to shard)

● Challenges:
  ○ Verification of low-level eBPF/SBF is harder
  ○ No common format between apps (data format is up to the app):
    ■ Inputs are just array of bytes
    ■ Serialization/deserialization
  ○ Compiled Rust can be harder to verify than human-written C
    ■ Rust union types, dangling pointers, etc.
Solana Programming (not in this talk)

- **Accounts**
  - Fields: lamports, owner, executable, data, rent epoch
  - Program and Data accounts

- **Transactions** consist of instructions
  - All programs are *stateless*: any data they interact with is stored in separate accounts that are passed in via instructions

- **PDAs** (Program Derived Address): data account owned by programs instead of users
  - Used to implement associative maps

- **CPI** (Cross Program Invocations)

- **Deserialize/Serialize**

https://solanacookbook.com/
Certora Prover Architecture for Solana

In this talk:
1: Trust but Verify
2: Abstract Interpretation
   - Recover types
   - Memory analysis

Correctness preserving transformations

VC generator
SMT Solver
Analyzer

REPORT

CERTORA PROVER ARCHITECTURE FOR SOLANA

RUST
.rs

C++
.cpp

Spec
.spec

Compiler

LLVM

.bc

LLVM-sbf

.ebpf

SBF

Decompiler

SPEC

.SMT2

BUG

REPORT
eBPF/SBF Virtual Machine

Blockchain State (program inputs)

Stack (byte-addressable)

Heap (byte-addressable)

Registers

r0
r1
r2
r3
r10

0x400000000

0x200000000

Text (code + rodata)

0x100000000

0x300000000
eBPF/SBF Virtual Machine

(Deserialized) Blockchain State

Account

Stack (byte-addressable)

Heap (byte-addressable)

Registers

r0
r1
r2
r3
r10

Text (code + rodata)
SBF Instruction Set

- Currently, three different dialects with similar bytecodes: bpf/sbf/sbfv2
- RISC-like instruction set
  - 11 general-purpose, 64-bit registers
    - r10 is read-only frame pointer to access to stack
  - ALU, JUMP, LOAD, STORE, MOVE
    - Jumps use only relative constant offsets: CFG construction is decidable
- Syscalls and eBPF-to-eBPF (internal) calls
  - r0: return
  - r1, ..., r5: caller-saved (volatile) registers
  - r6, ..., r9: callee-saved (non-volatile) registers
- No type information: no distinction between numbers and pointers
- Direct and indirect function calls: call graph construction is undecidable
SBF Disassembler

1. Translate ELF to a sequence of three-address instructions
   ○ Resolve Solana-specific relocations
2. CFG and Call graph construction: one per function
   ○ Indirect calls not supported
3. Inline all internal functions
   ○ Explicit modeling of call semantics
4. Compute Cone-of-Influence and slice program
5. Memory analysis
6. Translation to TAC program
Memory Analysis Assumptions

The analysis is sound under the following assumptions:

1. Memory safety
   ○ Absence of out-of-bounds accesses
   ○ Stack/Heap/Blockchain memory is initialized

2. First read from blockchain state returns non-deterministic values
   ○ Pointers do not alias with any other pointer

3. Each memory read accesses the same number of bytes last written
   ○ Checked by the analysis
Rust compiles to large programs

- Many irrelevant paths:
  - error paths
  - free pointers
- We only care about paths that can influence the evaluation of assertions

Solution: dataflow analysis that removes any path that is not in the Cone-Of-Influence (CoI)
Rust enum types

```rust
pub fn process_withdraw(
    program_id: &Pubkey,
    accounts: &[AccountInfo],
    amount: u64,
    expected_decimals: u8,
    new_decryptable_available_balance: DecryptableBalance,
    proof_instruction_offset: i64,
) -> ProgramResult {
    let mut confidential_transfer_account : &mut ConfidentialTransferAccount =
    token_account.get_extension_mut::<ConfidentialTransferAccount>().?
}
```

Result<(), ProgramError>

Result<&mut V, ProgramError>

question mark (?) operator
Rust enum types

- We need to discriminate *error* from *ok* paths
- Path-sensitive analysis is expensive and it is not easy to identify the discriminants

r1 is the discriminant

\[ 0 = \text{Ok} \]
\[ 1 = \text{Err} \]
We typically prove properties under the assumption that functions return \texttt{ok}

\textbf{Solution: iterative forward+backward analysis (Cousot&Cousot JLP'92/ASE'99) to prune error paths}
Analysis of SBF code

- Disassembler needs to translate SBF into a TAC program **without side effects**
  - TAC memory operations have an explicit argument “mem” that represents the (possibly infinite) set of memory locations being accessed
  - Two TAC memory ops do not alias if they have different “mem” names
- How: static memory partitioning
  - Split all program memory (stack, heap, and inputs) into a **finite** set of **disjoint** regions
  - For each memory instruction, map the memory location to a region
- Challenges:
  - No explicit allocation sites for program inputs because they are allocated either before the SBF program is loaded or by deserialization
  - Strong vs weak updates
Analysis of SBF code

○ Solution 1: flow-insensitive/field-sensitive pointer analysis (*Gurfinkel&Navas SAS’17*)
  ■ Adopted in LLVM-based verifiers such as SeaHorn and SMACK
  ■ Easy to model in SMT: one single points-to graph for the whole program
Analysis of SBF code

- Registers must be tracked flow-sensitively
  - They can be re-assigned at each instruction
Analysis of SBF code

- Registers must be tracked flow-sensitively
  - They can be re-assigned at each instruction
- Stack must be tracked flow-sensitively
- LLVM back-end reuses stack allocations
  
  ```
  lifetime.start.p0i8(%p1)
call try_borrow_mut_lamports(%p1) // %p1 points to src
lifetime.end.p0i8(%p1)
lifetime.start.p0i8(%p2)
call try_borrow_mut_lamports(%p2) // %p2 points to dst
lifetime.end.p0i8(%p2)
  ```
- Same slot 3976 in SBF for %p1 and %p2
Analysis of SBF code

- Solution 2: flow-sensitive pointer analysis
  - Solution adopted by verifiers such as Predator
  - Very precise but expensive: one points-to graph per basic block
  - Harder to model in SMT: a memory instruction can use different “mem” depending on which predecessor reaches the instruction
Analysis of SBF code

- Our solution:
  - Flow-sensitive stack and registers
  - Flow-insensitive heap and program inputs
  - **Stack scalarization:**
    - Each stack slot is translated to a scalar variable
    - This allows **strong updates** on local variables
    - Precise and easy to model in SMT
  - Weak updates on heap and program inputs
    - Still easy to model in SMT
Conclusions

- Solidity/EVM has attracted most of the attention of the verification community
- Verification of Solana contracts is a very exciting new research area
- Based on thrilling Rust and eBPF technology
  - A lot of the ideas and solutions can be reused in different contexts
- Both (compiled) Rust and SBF pose unique challenges to verification
- **Certora is building the first automatic verifier for Solana contracts!**
Many challenges are still to solve ...

- **Solana**
  - a. Cross-program invocations (CPI)
  - b. Automatic handling of serialization/deserialization
  - c. Verifying multiple transactions/instructions
    - For now, we focus on one instruction at the time, and manually provide context invariants
    - However, most exploited vulnerabilities used multiple instructions and transactions
  - d. Fuller model of transaction state
    - e.g., support instruction introspection (heavily used for implementing confidentiality)
  - e. Richer model of the blockchain environment: e.g., PDA-based links between accounts

- **Rust/SBF**
  - a. More precise memory abstraction to support Rust enum types
  - b. More precise abstractions for the heap (e.g., Box, Vec, …)

- **SMT**
  - a. Improve domain-specific treatment of non-linear arithmetic