Verification of Solana Programs

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

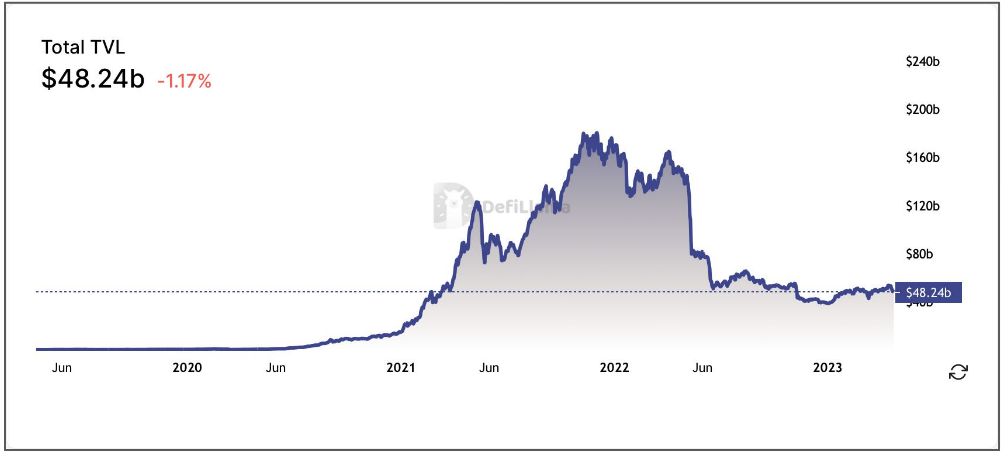
Venice, May 26th 2023

Symposium on Challenges of Software Verification (CSV)

certora

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



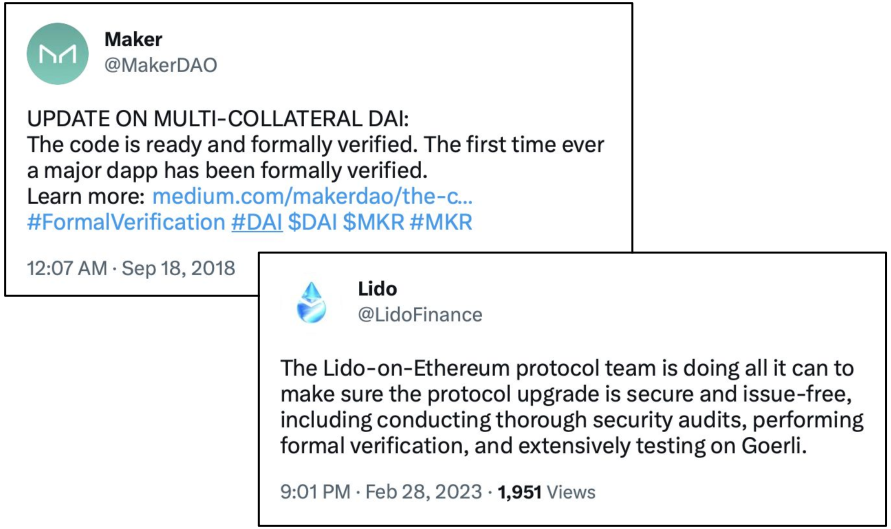
certora

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



Maker
@MakerDAO

UPDATE ON MULTI-COLLATERAL DAI:
The code is ready and formally verified. The first time ever a major dapp has been formally verified.
Learn more: medium.com/makerdao/the-c...
#FormalVerification #DAI $DAI $MKR #MKR

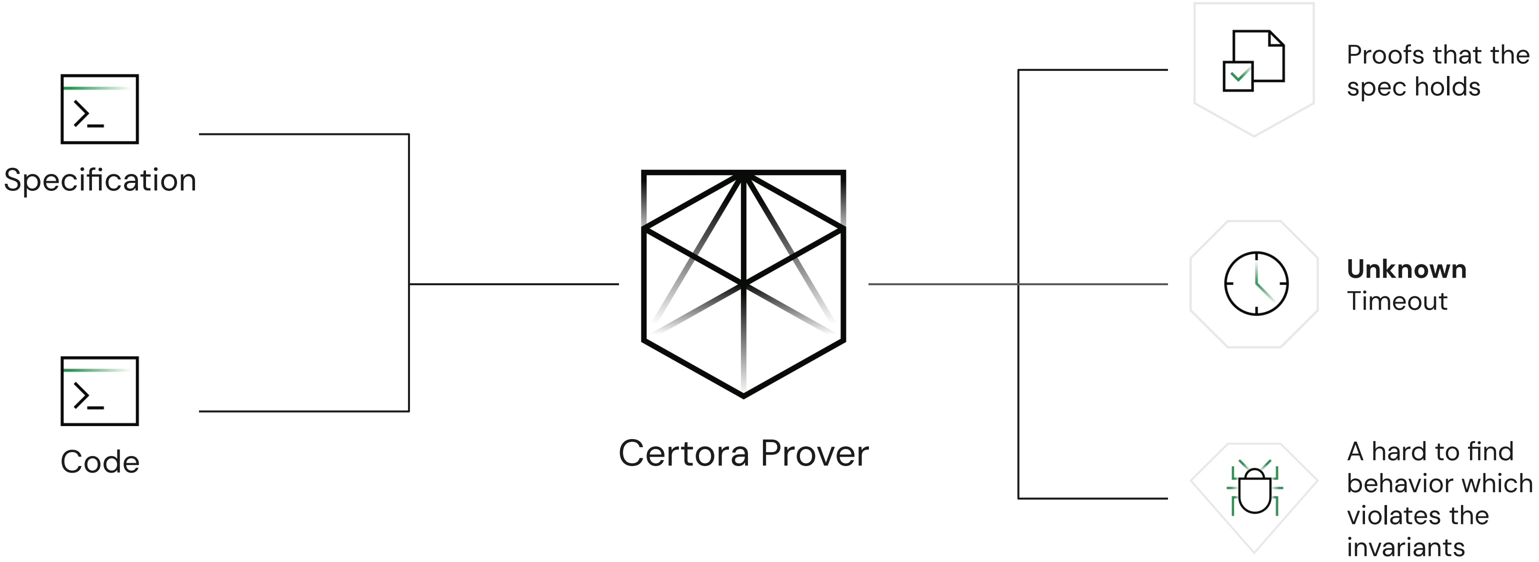
12:07 AM · Sep 18, 2018

Lido
@LidoFinance

The Lido-on-Ethereum protocol team is doing all it can to make sure the protocol upgrade is secure and issue-free, including conducting thorough security audits, performing formal verification, and extensively testing on Goerli.

9:01 PM · Feb 28, 2023 · **1,951** Views

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
  Bank still fulfills all commitments
- Users’ money cannot be locked or lost

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

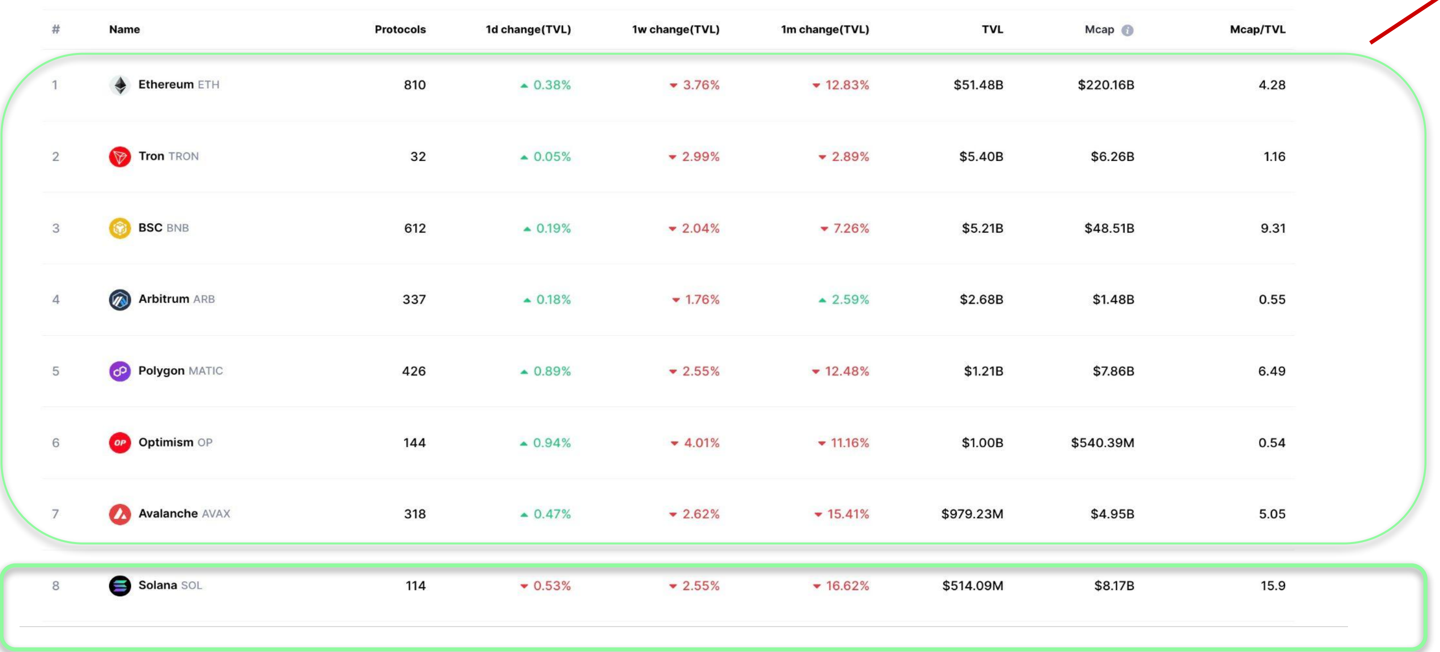
SushiSwap	$807M	AAVE	$6.5B	Compound	$2.7B	Balancer	$1.18B
Strategy	2	V3	1	Comet	5	V2	2
Trident	5	V2	2	V2	5		
KashiPair	3						
DutchAuction	1						

Solidity
@solidity_lang

“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!”

certora

Why Formally Verify Solana (https://solana.com) ?



#	Name	Protocols	1d change(TVL)	1w change(TVL)	1m change(TVL)	TVL	Mcap	Mcap/TVL
1	**Ethereum** ETH	810	▲ 0.38%	▼ 3.76%	▼ 12.83%	$51.48B	$220.16B	4.28
2	**Tron** TRON	32	▲ 0.05%	▼ 2.99%	▼ 2.89%	$5.40B	$6.26B	1.16
3	**BSC** BNB	612	▲ 0.19%	▼ 2.04%	▼ 7.26%	$5.21B	$48.51B	9.31
4	**Arbitrum** ARB	337	▲ 0.18%	▼ 1.76%	▲ 2.59%	$2.68B	$1.48B	0.55
5	**Polygon** MATIC	426	▲ 0.89%	▼ 2.55%	▼ 12.48%	$1.21B	$7.86B	6.49
6	**Optimism** OP	144	▲ 0.94%	▼ 4.01%	▼ 11.16%	$1.00B	$540.39M	0.54
7	**Avalanche** AVAX	318	▲ 0.47%	▼ 2.62%	▼ 15.41%	$979.23M	$4.95B	5.05
8	**Solana** SOL	114	▼ 0.53%	▼ 2.55%	▼ 16.62%	$514.09M	$8.17B	15.9

EVM

Non-EVM

certora

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)

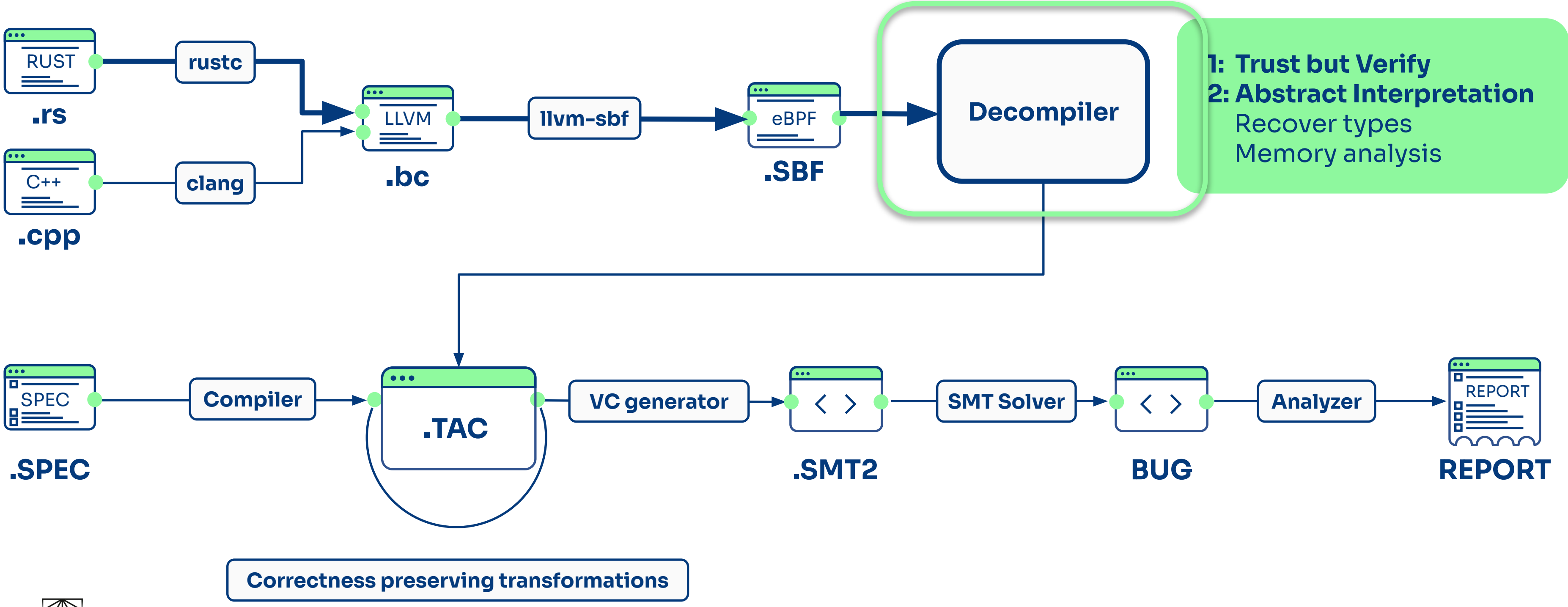
- **Accounts**
  - Fields: lamports, owner, executable, data, rent epoch
  - Program and Data accounts
- **Transactions** consist of **instructions**
- All programs are *stateless*: any data they interact with is stored in separate accounts that are passed in via instructions
- **PDAs** (Program Derived Address): data account owned by programs instead of users
  - Used to implement associative maps
- **CPI** (Cross Program Invocations)
- **Deserialize/Serialize**

https://solanacookbook.com/

certora

Certora Prover Architecture for Solana

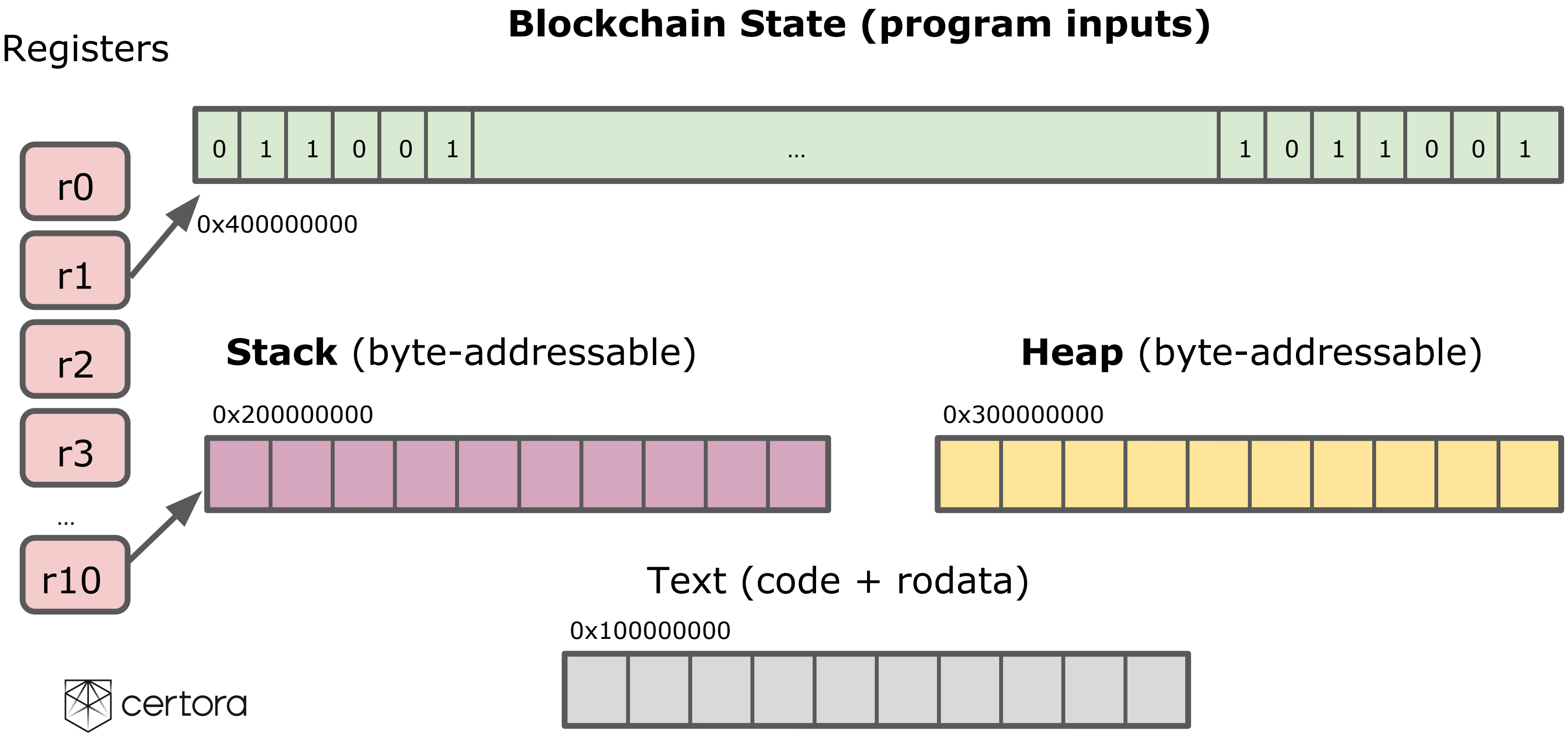
In this talk



1: Trust but Verify
2: Abstract Interpretation
Recover types
Memory analysis

certora

eBPF/SBF Virtual Machine



Registers

r0

r1

r2

r3

…

r10

Blockchain State (program inputs)

0 1 1 0 0 1 … 1 0 1 1 0 0 1

0x400000000

Stack (byte-addressable)

0x200000000

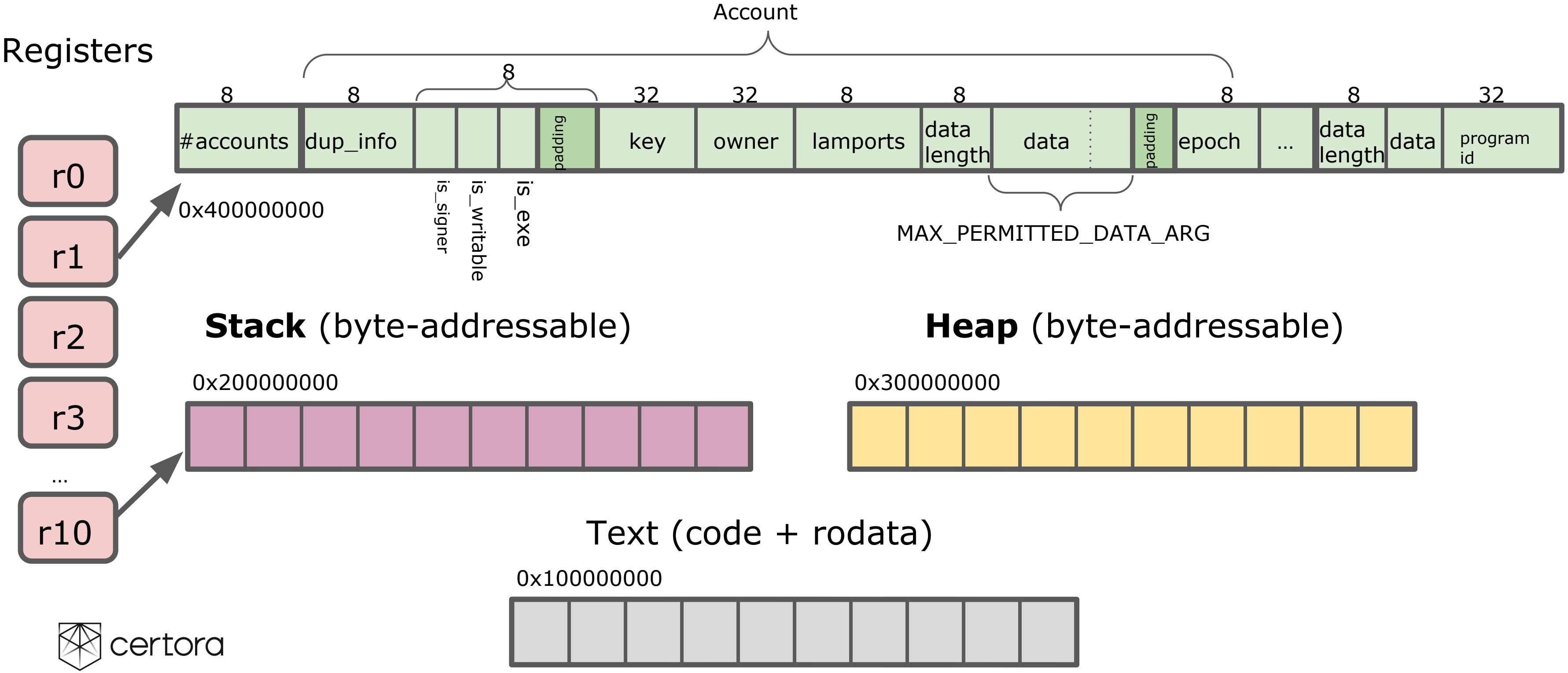
Heap (byte-addressable)

0x300000000

Text (code + rodata)

0x100000000

eBPF/SBF Virtual Machine



(Deserialized) Blockchain State

Account

8	8	8				32	32	8	8			8		8		32
#accounts	dup_info	is_signer	is_writable	is_exe	padding	key	owner	lamports	data length	data	padding	epoch	…	data length	data	program id

0x400000000

MAX_PERMITTED_DATA_ARG

Registers

r0

r1

r2

r3

…

r10

Stack (byte-addressable)

0x200000000

Heap (byte-addressable)

0x300000000

Text (code + rodata)

0x100000000

certora

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

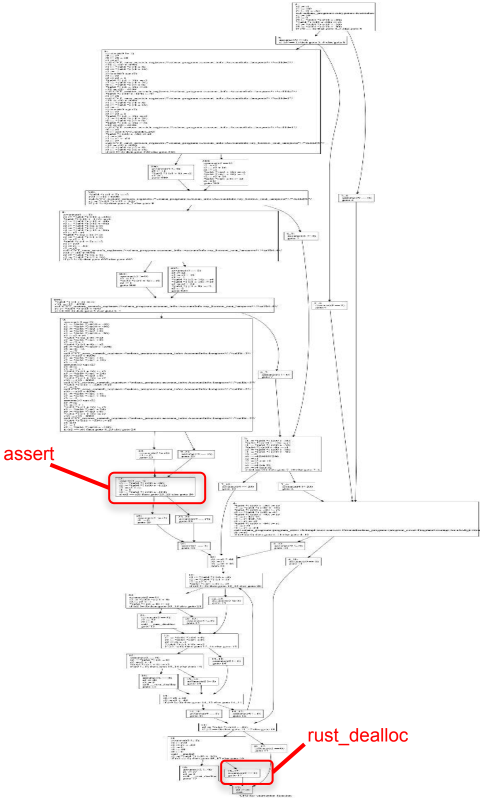
1. Translate ELF to a sequence of three-address instructions
    - Resolve Solana-specific relocations
2. CFG and Call graph construction: one per function
    - Indirect calls not supported
3. Inline all internal functions
    - Explicit modeling of call semantics
4. Compute Cone-of-Influence and slice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

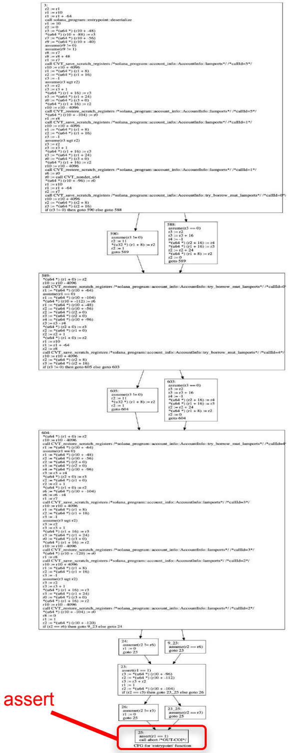
Memory Analysis Assumptions

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

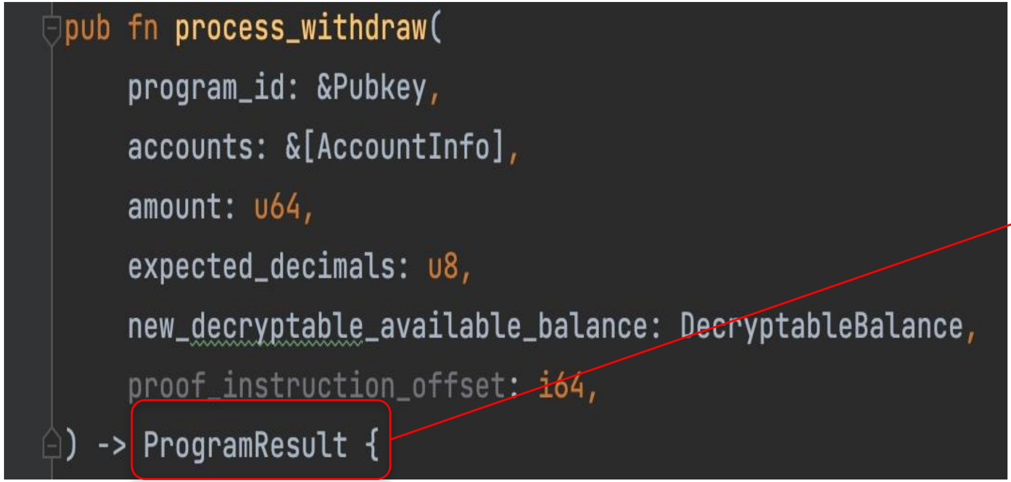




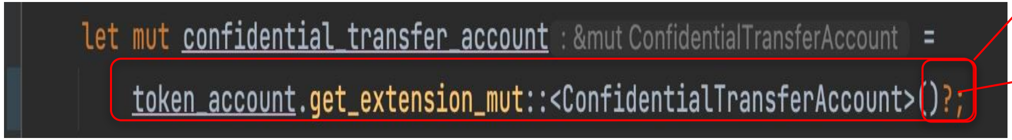
- Many irrelevant paths:
  - error paths
  - free pointers
- We only care about paths that can influence the evaluation of assertions

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

Rust enum types



```
pub fn process_withdraw(
    program_id: &Pubkey,
    accounts: &[AccountInfo],
    amount: u64,
    expected_decimals: u8,
    new_decryptable_available_balance: DecryptableBalance,
    proof_instruction_offset: i64,
) -> ProgramResult {
```



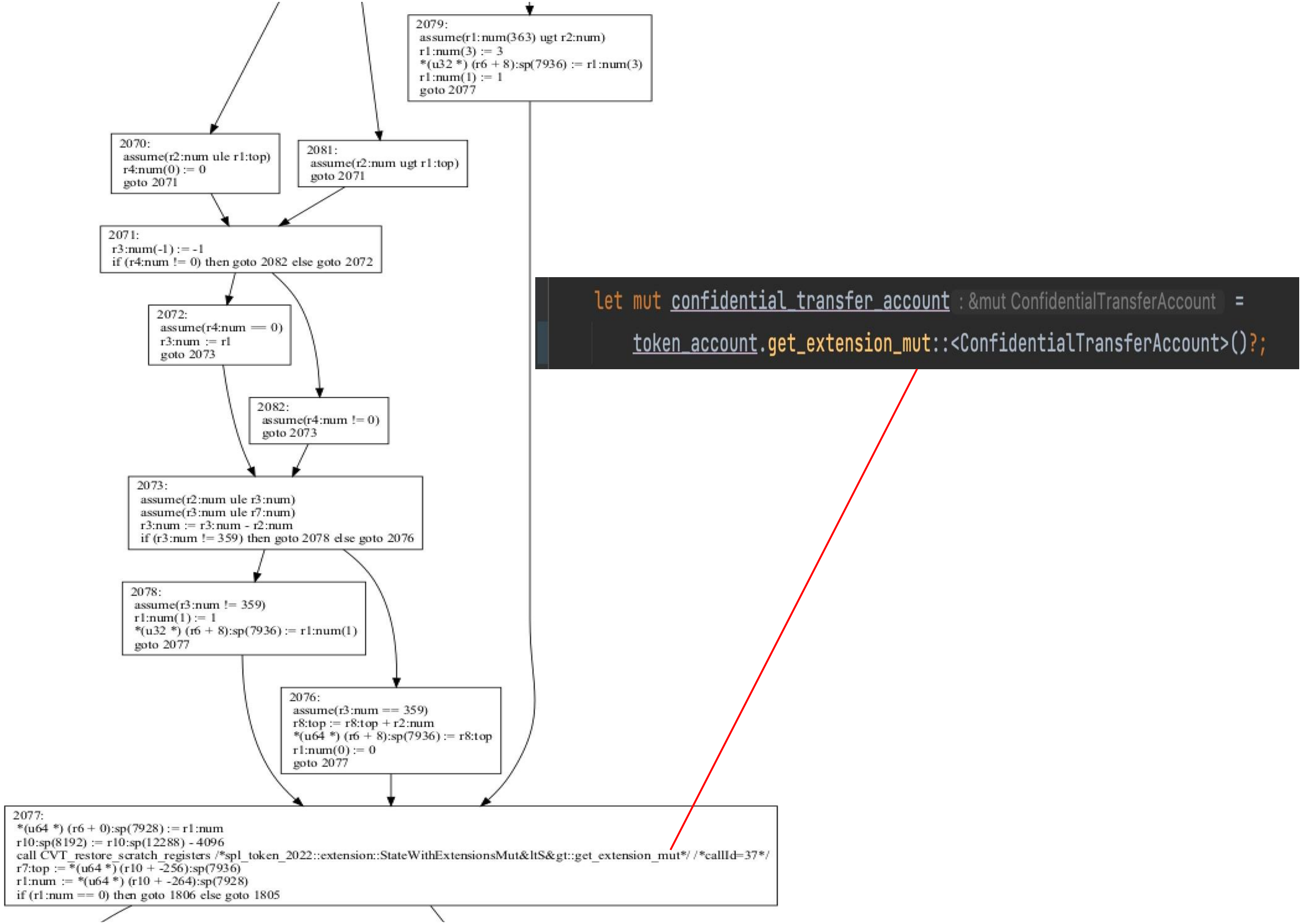
```
    let mut confidential_transfer_account : &mut ConfidentialTransferAccount =
        token_account.get_extension_mut::<ConfidentialTransferAccount>()?;
```

`Result<(),ProgramError>`

`Result<&mut V, ProgramError>`

question mark (`?`) operator

Rust enum types



```
2079:
 assume(r1:num(363) ugt r2:num)
 r1:num(3) := 3
 *(u32 *) (r6 + 8):sp(7936) := r1:num(3)
 r1:num(1) := 1
 goto 2077
```

```
2070:
 assume(r2:num ule r1:top)
 r4:num(0) := 0
 goto 2071
```

```
2081:
 assume(r2:num ugt r1:top)
 goto 2071
```

```
2071:
 r3:num(-1) := -1
 if (r4:num != 0) then goto 2082 else goto 2072
```

```
2072:
 assume(r4:num == 0)
 r3:num := r1
 goto 2073
```

```
2082:
 assume(r4:num != 0)
 goto 2073
```

```
2073:
 assume(r2:num ule r3:num)
 assume(r3:num ule r7:num)
 r3:num := r3:num - r2:num
 if (r3:num != 359) then goto 2078 else goto 2076
```

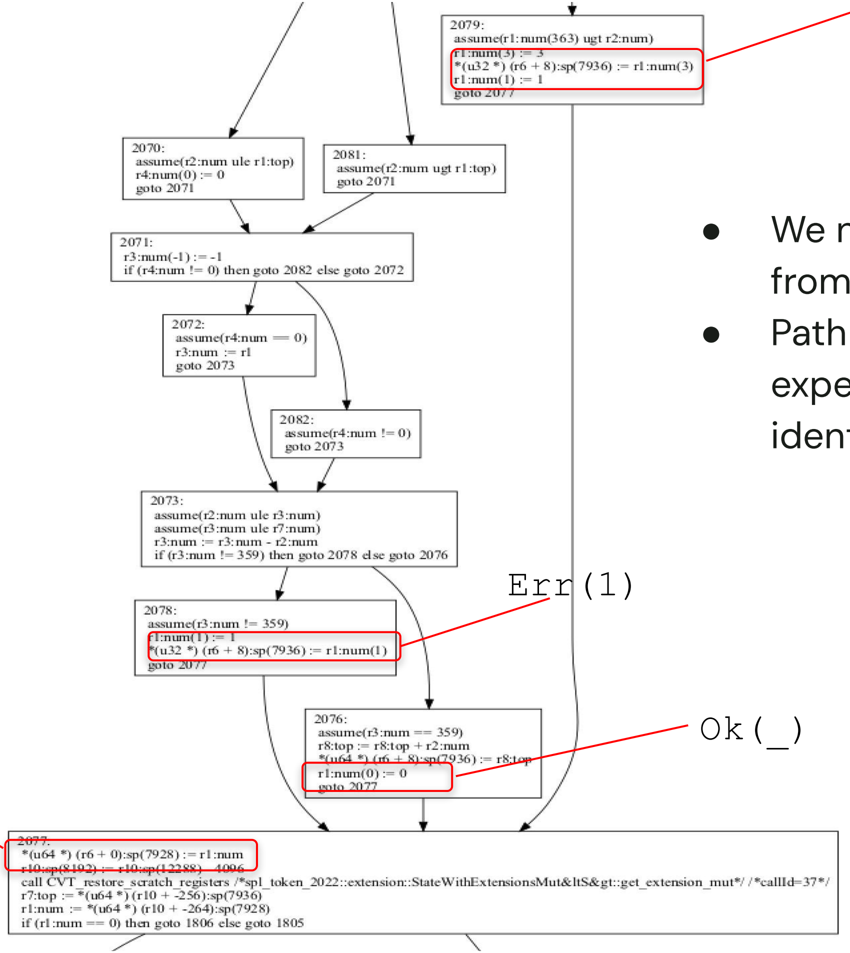
```
2078:
 assume(r3:num != 359)
 r1:num(1) := 1
 *(u32 *) (r6 + 8):sp(7936) := r1:num(1)
 goto 2077
```

```
2076:
 assume(r3:num == 359)
 r8:top := r8:top + r2:num
 *(u64 *) (r6 + 8):sp(7936) := r8:top
 r1:num(0) := 0
 goto 2077
```

```
2077:
 *(u64 *) (r6 + 0):sp(7928) := r1:num
 r10:sp(8192) := r10:sp(12288) - 4096
 call CVT_restore_scratch_registers /*spl_token_2022::extension::StateWithExtensionsMut&ltS&gt::get_extension_mut*/ /*callId=37*/
 r7:top := *(u64 *) (r10 + -256):sp(7936)
 r1:num := *(u64 *) (r10 + -264):sp(7928)
 if (r1:num == 0) then goto 1806 else goto 1805
```

```
let mut confidential_transfer_account : &mut ConfidentialTransferAccount =
    token_account.get_extension_mut::<ConfidentialTransferAccount>()?;
```

Rust enum types



```
2079:
 assume(r1:num(363) ugt r2:num)
 r1:num(3) := 3
 *(u32 *) (r6 + 8):sp(7936) := r1:num(3)
 r1:num(1) := 1
 goto 2077
```

```
2070:
 assume(r2:num ule r1:top)
 r4:num(0) := 0
 goto 2071
```

```
2081:
 assume(r2:num ugt r1:top)
 goto 2071
```

```
2071:
 r3:num(-1) := -1
 if (r4:num != 0) then goto 2082 else goto 2072
```

```
2072:
 assume(r4:num == 0)
 r3:num := r1
 goto 2073
```

```
2082:
 assume(r4:num != 0)
 goto 2073
```

```
2073:
 assume(r2:num ule r3:num)
 assume(r3:num ule r7:num)
 r3:num := r3:num - r2:num
 if (r3:num != 359) then goto 2078 else goto 2076
```

```
2078:
 assume(r3:num != 359)
 r1:num(1) := 1
 *(u32 *) (r6 + 8):sp(7936) := r1:num(1)
 goto 2077
```

```
2076:
 assume(r3:num == 359)
 r8:top := r8:top + r2:num
 *(u64 *) (r6 + 8):sp(7936) := r8:top
 r1:num(0) := 0
 goto 2077
```

```
2077:
 *(u64 *) (r6 + 0):sp(7928) := r1:num
 r10:sp(8192) := r10:sp(12288) - 4096
 call CVT_restore_scratch_registers /*spl_token_2022::extension::StateWithExtensionsMut&ltS&gt::get_extension_mut*/ /*callId=37*/
 r7:top := *(u64 *) (r10 + -256):sp(7936)
 r1:num := *(u64 *) (r10 + -264):sp(7928)
 if (r1:num == 0) then goto 1806 else goto 1805
```

`Err(3)`

`Err(1)`

`Ok(_)`

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

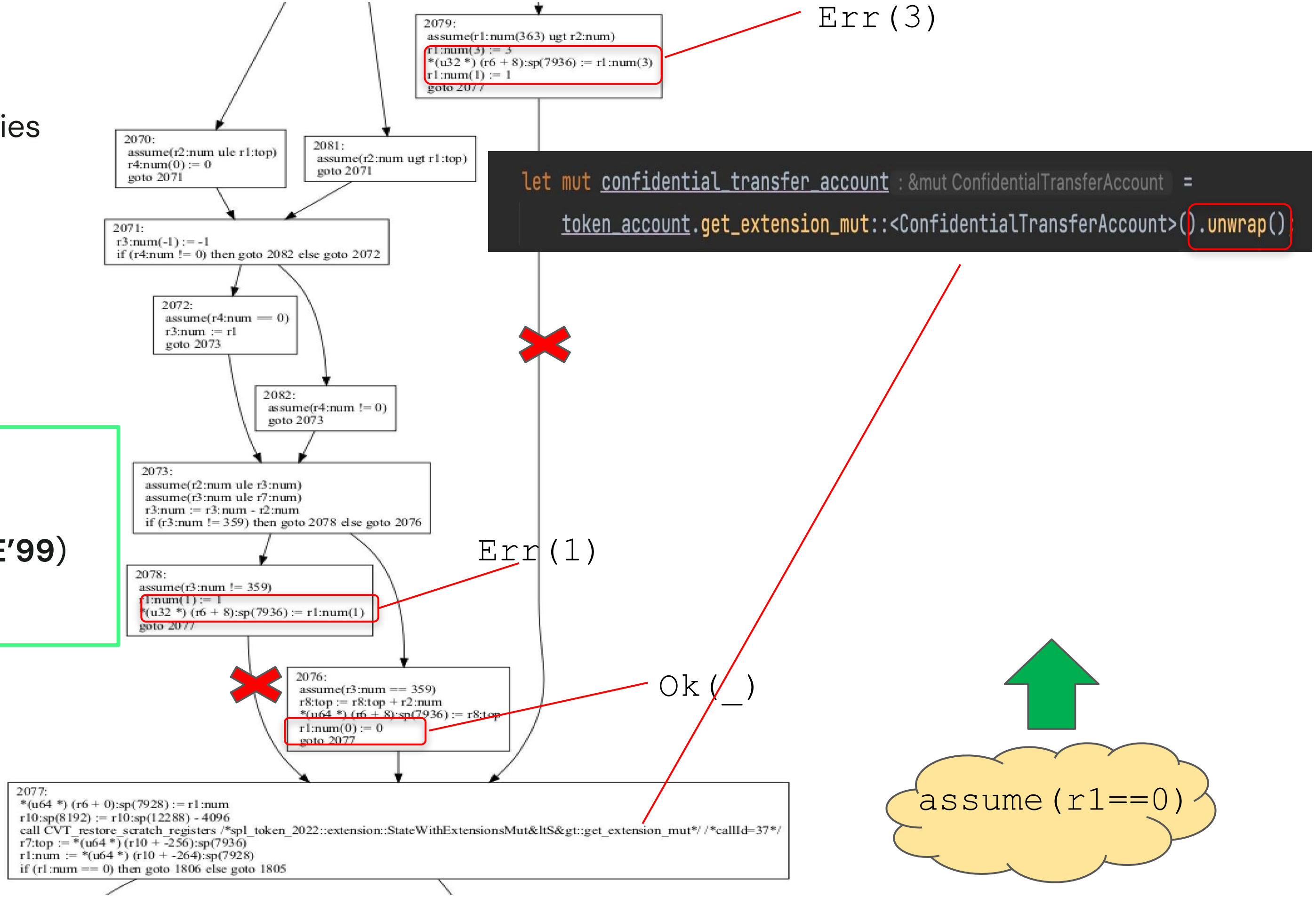
`r1` is the discriminant

0 = `Ok`

1 = `Err`

Rust enum types

We typically prove properties under the assumption that functions return `ok`



```
let mut confidential_transfer_account : &mut ConfidentialTransferAccount =
    token_account.get_extension_mut::<ConfidentialTransferAccount>().unwrap();
```

```
2079:
 assume(r1:num(363) ugt r2:num)
 r1:num(3) := 3
 *(u32 *) (r6 + 8):sp(7936) := r1:num(3)
 r1:num(1) := 1
 goto 2077

2070:
 assume(r2:num ule r1:top)
 r4:num(0) := 0
 goto 2071

2081:
 assume(r2:num ugt r1:top)
 goto 2071

2071:
 r3:num(-1) := -1
 if (r4:num != 0) then goto 2082 else goto 2072

2072:
 assume(r4:num == 0)
 r3:num := r1
 goto 2073

2082:
 assume(r4:num != 0)
 goto 2073

2073:
 assume(r2:num ule r3:num)
 assume(r3:num ule r7:num)
 r3:num := r3:num - r2:num
 if (r3:num != 359) then goto 2078 else goto 2076

2078:
 assume(r3:num != 359)
 r1:num(1) := 1
 *(u32 *) (r6 + 8):sp(7936) := r1:num(1)
 goto 2077

2076:
 assume(r3:num == 359)
 r8:top := r8:top + r2:num
 *(u64 *) (r6 + 8):sp(7936) := r8:top
 r1:num(0) := 0
 goto 2077

2077:
 *(u64 *) (r6 + 0):sp(7928) := r1:num
 r10:sp(8192) := r10:sp(12288) - 4096
 call CVT_restore_scratch_registers /*spl_token_2022::extension::StateWithExtensionsMut&ltS&gt::get_extension_mut*/ /*callId=37*/
 r7:top := *(u64 *) (r10 + -256):sp(7936)
 r1:num := *(u64 *) (r10 + -264):sp(7928)
 if (r1:num == 0) then goto 1806 else goto 1805
```

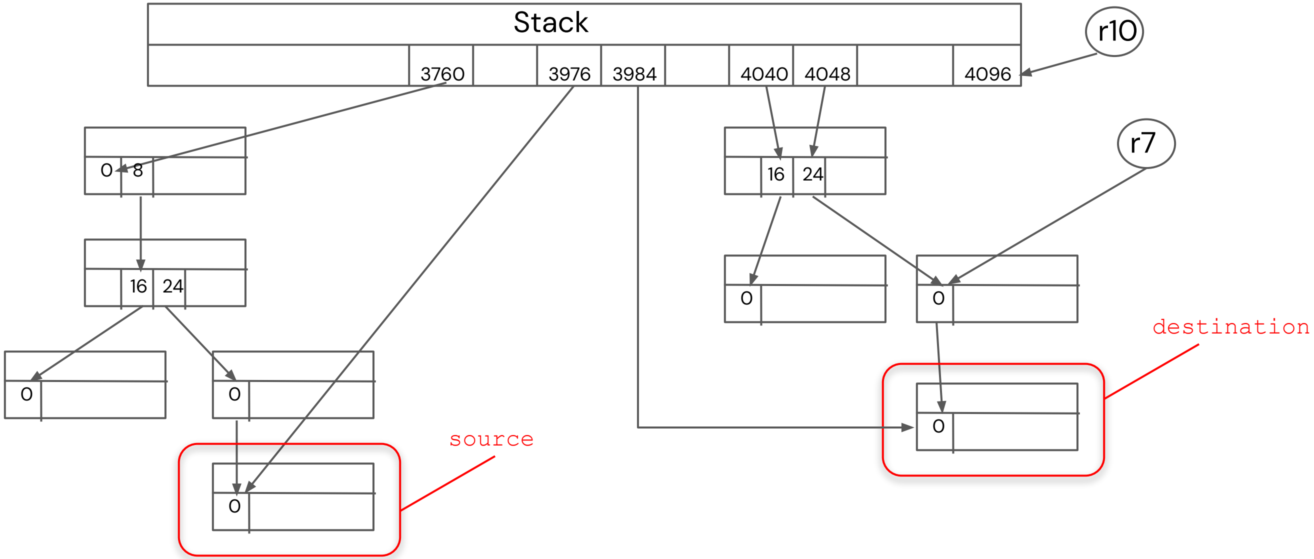
Solution: iterative forward+backward analysis (**Cousot&Cousot JLP'92/ASE'99**) to prune error paths

Analysis of SBF code

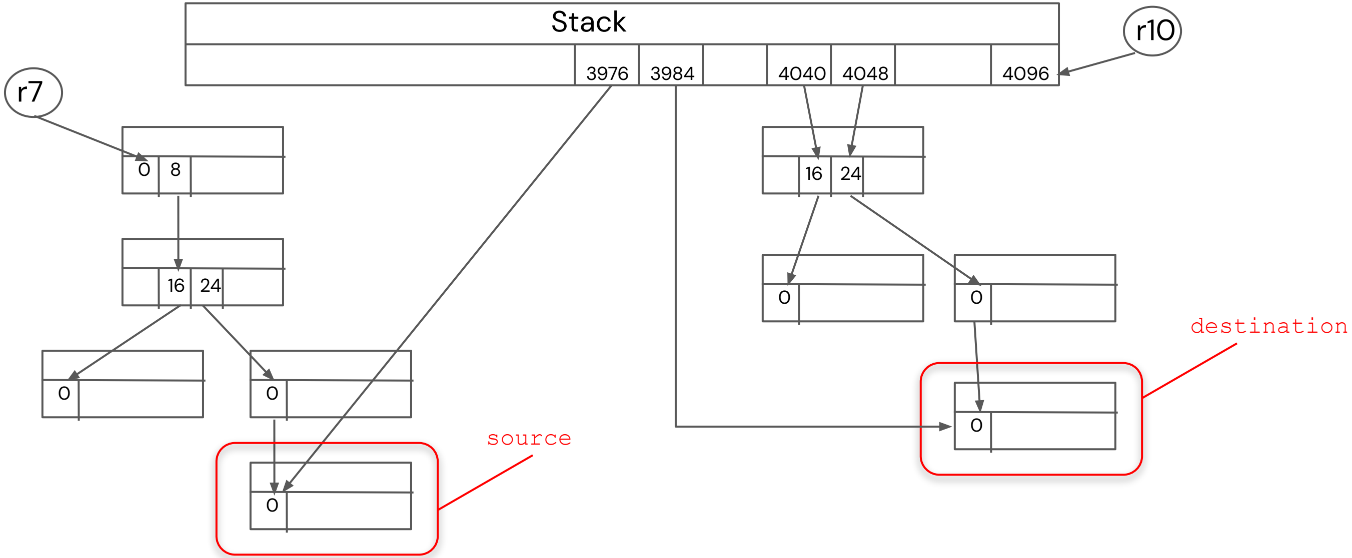
- Disassembler needs to translate SBF into a TAC program **without side effects**
  - TAC memory operations have an explicit argument “mem” that represents the (possibly infinite) set of memory locations being accessed
  - Two TAC memory ops do not alias if they have different “mem” names
- How: static 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



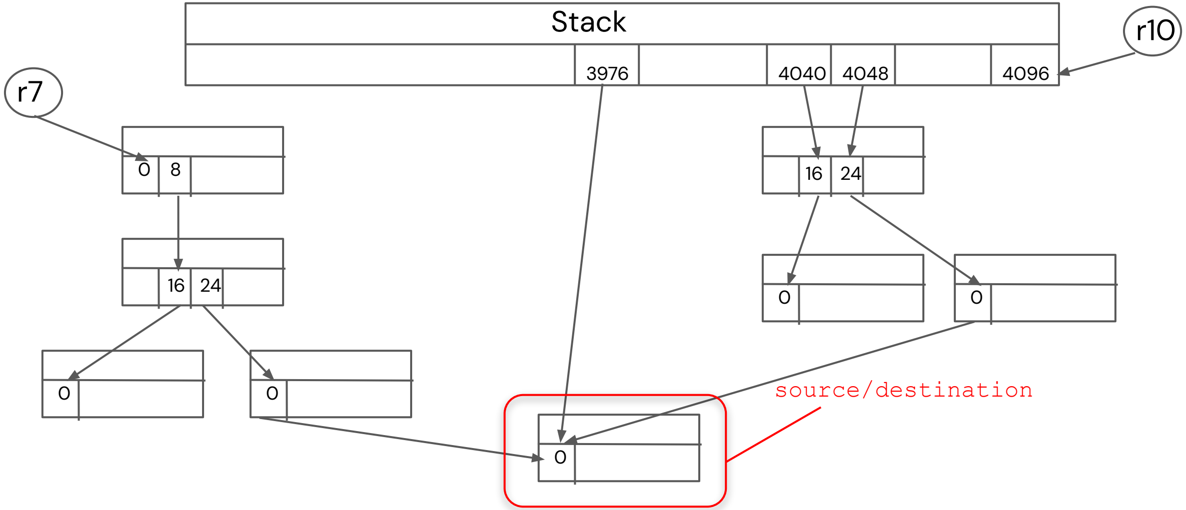
Analysis of SBF code



- Registers must be tracked flow-sensitively
    - They can be re-assigned at each instruction

certora

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`

certora

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