

# Fuzzing Solana Runtimes

alpharush

W3ST @ Cannes  
Mar. 31, 2026

**AR** asymmetric  
research

# Objective

Find bugs in Solana runtime execution:

- Logical bugs within a single implementation
- Regressions across versions of the same implementation, e.g. Agave vs Agave
- Divergences across independent implementations, e.g. Agave vs Firedancer

# Primer

Some differences that must be accounted for versus e.g. Ethereum transaction fuzzing:

- All accounts and permissions read/write specified up-front
- Cross-program invocation requires guest to prepare data structures in accordance with C/Rust ABI (various restrictions on where objects can be)

# Existing work: Solfuzz

- Define protobuf format and FFI interface
- Each client implements the interface and exposes it as a shared object
- Differential fuzzing performed with libprotobuf-mutator + libfuzzer
  - Output protobuf difference -> block hash mismatch (state digest)
- Bottom-up harness design:
  - Instruction
  - Transaction
  - Block

# Solfuzz (limitations)

- Programs generated are raw ELF
  - Extending and targeting is tedious and fragile
- Libprotobuf-mutator lacks domain specific mutations and understanding of structure validity constraints
  - Effect of mutation may need to be fixed-up e.g. to prevent sanitization errors

# Idea: Contrasol

A higher-level generator/fuzzer built to complement Solfuzz.

- Generate programs a la Fuzzilli and Syzkaller
- Integrate with LibAFL for coverage-guided, multi-core fuzzing
  - Custom input type with its own mutators
- CLI toolkit: enables extending e.g. deduplication and minimization.

# Manually Crafted Program

- Quick MVP of program-centric approach
- Program is an interpreter:
  - Interprets instruction data as actions and performs them
  - Covers memory ops, syscalls, account writes/resizes, and both Rust and C-style CPI

```
#[derive(Debug)]
pub struct FuzzData {
    actions: Vec<Action>,
    accounts: Vec<FuzzAccount>,
    seed: u64,
}

pub enum Action {
    Cpi {
        c_or_rust: u8,
        program_id_idx: u8,
        accounts: Vec<CpiAccount>,
        actions: Vec<Action>,
    }
}

fn fuzz_one(input: &FuzzData) {
    let instruction = prepare(input);
    svm.process_instruction(&instruction);
}
```

# Manually Crafted Program (cont)

## Upsides

- Found a bug
- Many actions/instructions can be reused in Contrasol

## Downsides

- Inherently limited complexity
- Indirection of mutations due to mixing data/code

```
pub fn process_instruction<'a>(  
    program_id: &Pubkey,  
    accounts: &'a [AccountInfo<'a>],  
    instruction_data: &[u8],  
) -> ProgramResult {  
    let mut cursor = 0;  
    loop {  
  
        match instruction_data[cursor] {  
            0 => {  
                msg!("CPI");  
                cursor += 1;  
                cursor =  
do_cpi(accounts,&instruction_data, cursor)?;  
            }  
        }  
    }  
}
```

# Naive Program Generation

Generate SSA-form IR, lower to Rust, compile to SBF.

- Produced similar, uninteresting programs that didn't resemble exploits
  - Unlikely to invoke other programs
  - Many lines increase likelihood of failing prior to observable side-effects

```
let var_0 = 6u8;
if let Some(var_1) = accounts.get(var_0 as usize) {
    let var_2 = 1usize;
    if let Ok(mut var_3) = var_1.try_borrow_mut_data() {
        let var_4 = *program_id;
        let var_5 = instruction_data;
        let var_6 = [34, 39, 41, 15, 187, 125, 78, 139,
63];

        let var_7 = unsafe {
std::ptr::read(var_6.as_ptr().add(var_2 as usize)) };
        let var_8 = sol_memcmp(&var_6[..], &var_5[..],
var_2 as usize) as i64;
        msg!("{:?}", String::from_utf8_lossy(&var_6));
        let var_9 = 21usize;
        let var_10 = *program_id;
        let var_11 = accounts.iter().take(var_0 as
usize).map(|acc| (acc.key, acc.is_writable,
acc.is_signer)).collect::<Vec<_>>();
        let var_12 = unsafe {
std::ptr::read_unaligned(var_5.as_ptr().add(var_2 as usize)
as *const u64) };
        sol_memset(&mut var_3[..], var_7 as u8, var_2 as
usize);

        let var_13 = var_7 & var_7;
    }
}
```

# Naive Solana Program Generation (improved)

- Guard panics e.g. slice OOB
- Provide “hints” on number of accounts serialized and which can be invoked as programs
- Limit # of syscalls e.g. cryptographic
- Favor statements prior to and following CPI

# Prior Art: Fuzzilli

Intermediate language translated into Javascript source code

- Insight #1: syntactic correctness not enough, need programs that run, not fail (semantic correctness)
- Insight #2: templates aimed at suspicious areas complement pure generation
- Insight #3: Evolve and fixup programs by “probing”: use runtime info to inform decision

# Applying Fuzzilli: Inspiration #1

- One program mutation introducing failures can cause all following mutations to be unreachable
- Use custom LibAFL stage
  - Incrementally mutate and backtrack if introduced failure
  - Program-focused mutations for this stage (account state and transaction unchanged)
- Conclusion: many mutations “wasted”, not fine-grained runtime observation to intelligently evolved input

# Applying Fuzzilli: Inspiration #2

- Start with more interesting programs
  - Extract Agave test suite into templated actions/scenarios
  - Result: start pool of programs with complex flows e.g. nested CPI
- Refine intermediate language
  - Transaction mutation might break program
    - Invoke -> FindExecutableAndInvoke
    - ChangeOwner -> ChangeOwnerFromProgramAToB

# Prior Art: Syzkaller

Program generation from declarative specification of syscall

- Insight #1: Resource aware
  - output of syscalls with overlapping resource types become inputs
  - API use meaningful ordering (call `foo`, then `bar`)
  - avoids many unused variables
- Insight #2: Scale with writing descriptions, not hand-crafting generators

# Applying Syzkaller

- Programs chain dependent operations, not disconnected statements.
  - If a program invokes a CPI, it should read the return data
  - If we create an instruction, it should be invoked
- Builtin/ system instructions promoted to “named” actions, not just program ID/data
  - Interface-definition language analogous to syscall descriptions
    - No longer treated opaquely/ blindly mangled

# Change Input “Shape”

- Protobuf input contains many self-referential/ duplicated info and very low-level
  - Writing mutations must honor these and is error-prone
  - Edit-distance longer than necessary
- Make some parts of input “symbolic”
  - Example: dedicated notion of fee payer kind that is added on-demand
  - Account keys are no longer indices: removing won't have cascading effects
- Resolve and convert to protobuf

# Challenges

- Compiling Rust -> SBPF is a bottleneck (custom compiler?)
- Creating objects at arbitrary memory locations with values that don't trivially fail
- Evolving program without introducing failure yet not favoring "simple" programs
- Balancing over-constraining "fixups" against rejection rates
- Alternative feedback to coverage once saturated

# Future Direction: Fandango-rs

- Treat validity as fitness function instead binary classification
  - Allows true evolution instead of one-shot generation of good input
- Mutations aware of which part of input is invalid and replace it (not necessarily valid)
- Mitigate risk of over-constraining input and biasing towards small/trivial programs
- Feedback is about input-diversity (grammar coverage)

# THANK YOU

QUESTIONS?