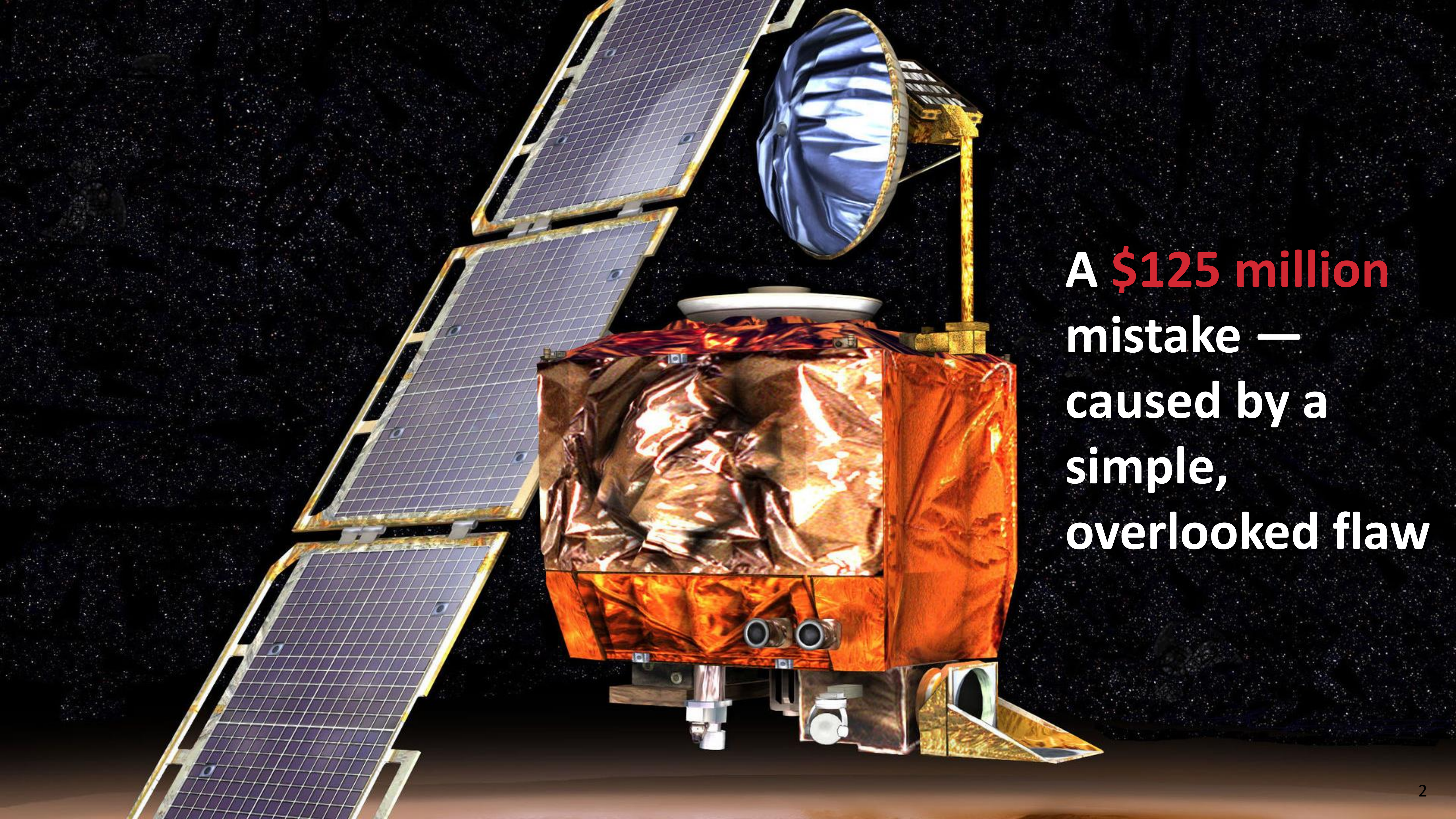
Fuzzing Rust Smart Contracts

Writing a bug printer engine from scratch

Kevin Valerio <kevin@srlabs.de>, Daniel Schmidt <schmidt@srlabs.de>

Security
Research
Labs



A $125 million mistake — caused by a simple, overlooked flaw

Nice to meet you 👐

Kevin Valerio

Security Engineer at SRLabs

Background in pentesting and Web3 security

Daniel Schmidt

Security Researcher at SRLabs

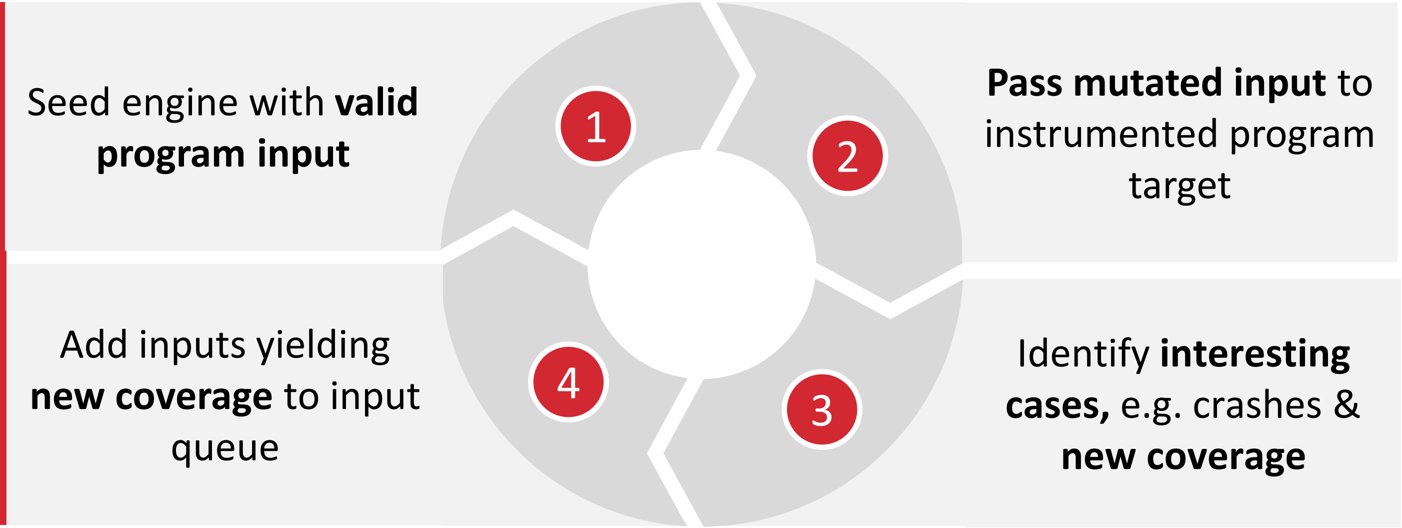
Background in protocol and virtual machine security

Agenda

1. Overview
2. **Background**
3. Challenges
4. Solutions
5. Success

Fuzzing identifies vulnerabilities via mutating valid program inputs

Fuzzing is an iterative loop



1 Seed engine with **valid program input**

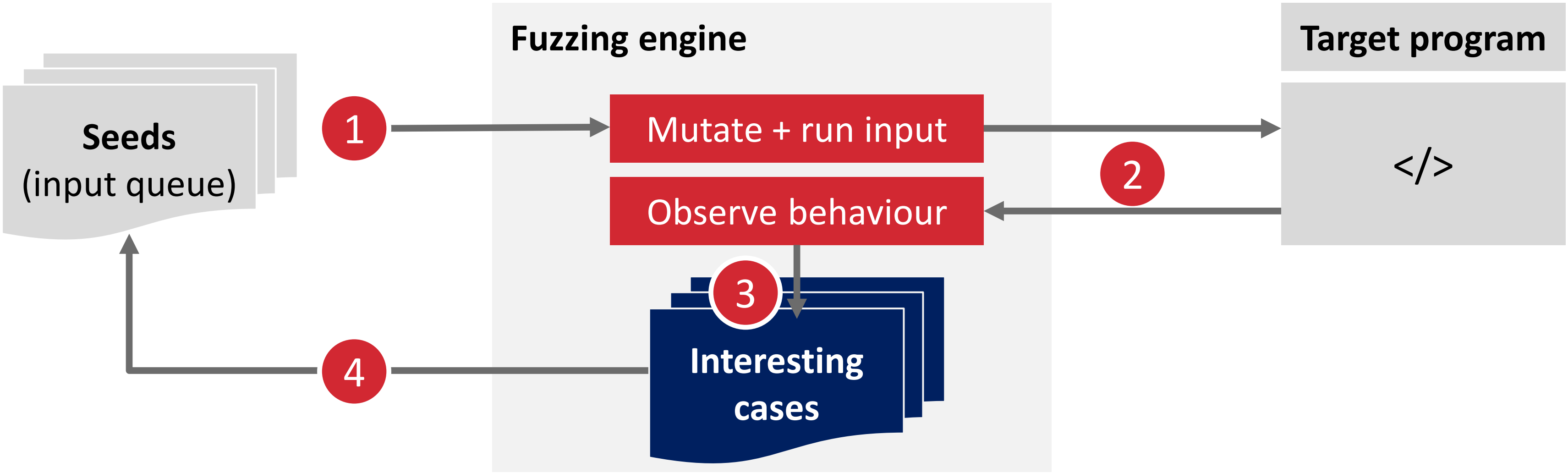
2 **Pass mutated input** to instrumented program target

3 Identify **interesting cases,** e.g. crashes & **new coverage**

4 Add inputs yielding **new coverage** to input queue

Coverage-guided fuzzing tracks code paths and leverages the gathered coverage to generate new test cases. To enable coverage-guided fuzzing, we need to **instrument** the target

Simplified fuzzing architecture



Instrument the target by injecting callbacks to enable coverage-guided fuzzing

Target instrumentation and coverage callbacks

1 **Identification** of basic blocks

2 **Insertion** of fuzzer callback at every basic block

3 **Callbacks** write to coverage map during execution

4 **Evaluation** of coverage by the fuzzing engine

Instrumentation example

Target Code

```
void parse_input(char *input) {
    if (input[0] == ‘F’) {
        if (input[1] == ‘U’) {
            if (input[2] == ‘Z’) {
                if (input[3] == ’Z’) {
                    // Crash here
```

Instrumented Code

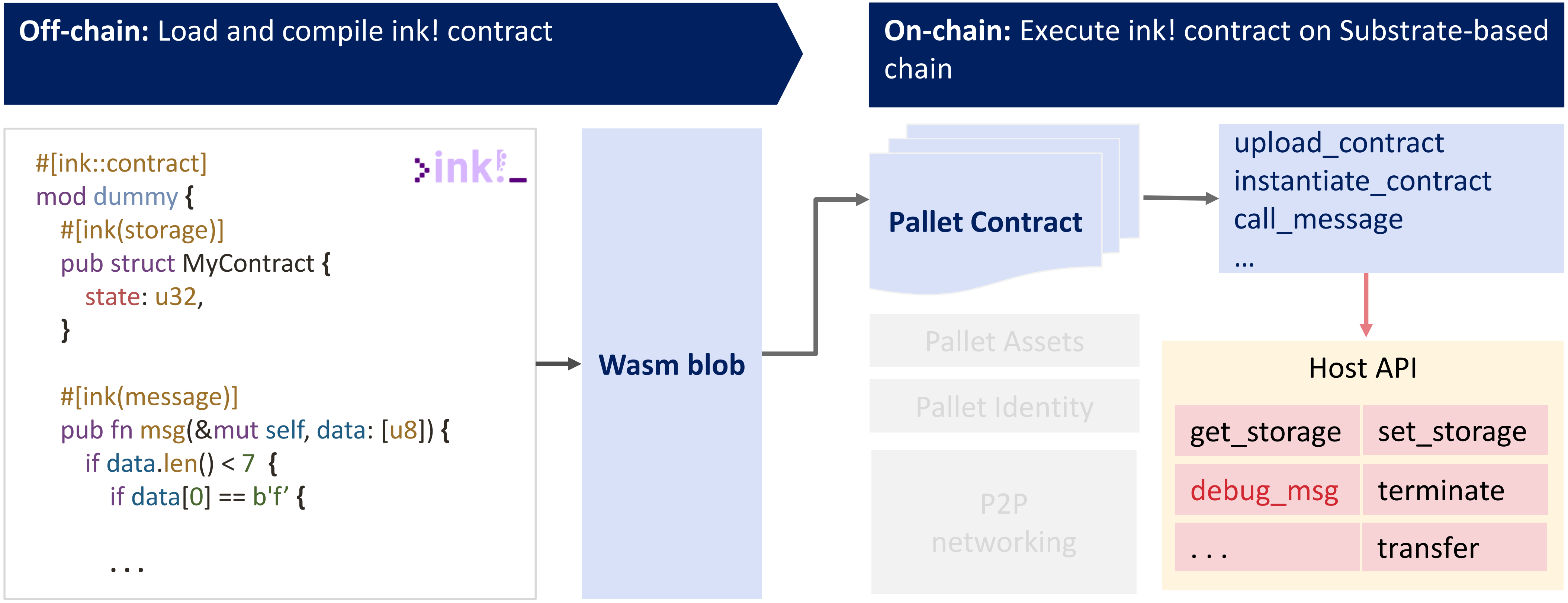
```
void parse_input(char *input) {
    if (input[0] == ‘F’) {
        __sanitizer_cov_trace_pc()
        if (input[1] == ‘U’) {
            __sanitizer_cov_trace_pc()
            if (input[2] == ‘Z’) {
                __sanitizer_cov_trace_pc()
                if (input[3] == ’Z’) {
                    __sanitizer_cov_trace_pc()
                    // Crash here
```

ink! smart-contracts are permissionless programmable extensions deployed on the blockchain

Description

- Smart contracts is **permissionless code** running inside the blockchain
- **ink! is a programming language** for smart contracts within the Polkadot ecosystem
- Being able to execute **cross-chain transactions** from ink! makes it special within the ecosystem of smart contracts

Architecture



Off-chain: Load and compile ink! contract

On-chain: Execute ink! contract on Substrate-based chain

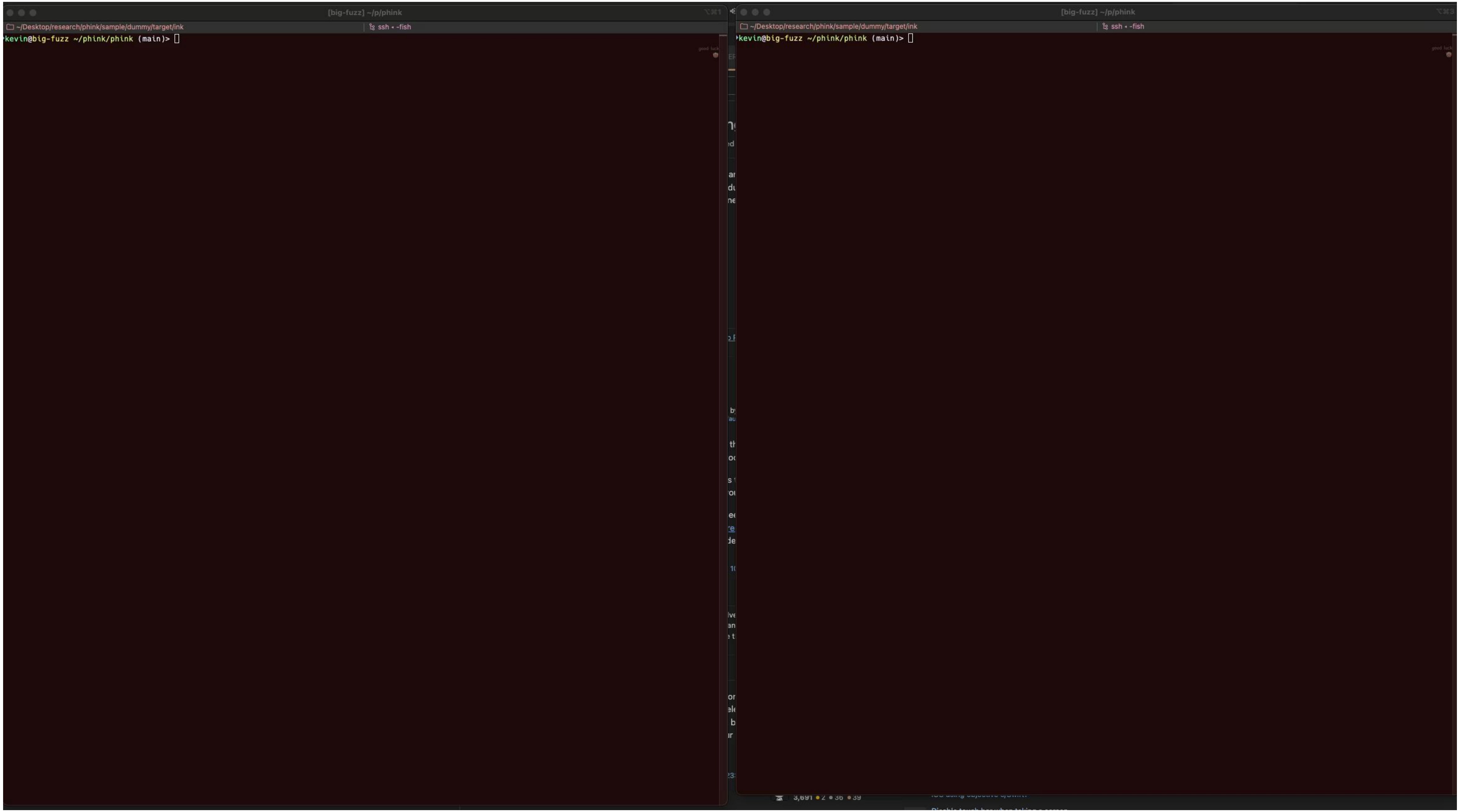
```
#[ink::contract]
mod dummy {
   #[ink(storage)]
   pub struct MyContract {
      state: u32,
   }

   #[ink(message)]
   pub fn msg(&mut self, data: [u8]) {
      if data.len() < 7  {
         if data[0] == b'f' {

         . . .
```

upload_contract
instantiate_contract
call_message
…

Host API

get_storage	set_storage
debug_msg	terminate
. . .	transfer

We present Phink, a coverage guided fuzzer for ink! smart contracts



[big-fuzz] ~/p/phink
~/Desktop/research/phink/sample/dummy/target/ink
ssh • -fish

```
kevin@big-fuzz ~/phink/phink (main)> 
```

[big-fuzz] ~/p/phink
~/Desktop/research/phink/sample/dummy/target/ink
ssh • -fish

```
kevin@big-fuzz ~/phink/phink (main)> 
```

Agenda

1. Overview
2. Background
3. **Challenges**
4. Solutions
5. Success

Several challenges have been identified during the creation of phink

	Challenge	Details
1	**Execution and instrumentation barriers**	▪ ink! contracts run in a VM, preventing direct instrumentation ▪ Standard fuzzers struggle to track execution paths in sandboxed environment
2	**Initial seed generation**	▪ Fuzzing campaigns need initial seeds so that they do not solely rely on random chance ▪ Creating initial seeds automatically is desirable
3	**Stateful execution and on-chain dependencies**	▪ Smart contracts interact with on-chain data and previous state ▪ Ensuring meaningful multi-call transactions during fuzzing is complex
4	**Coverage and feedback limitations**	▪ Generating coverage reports is difficult but crucial for optimizing fuzzing campaigns ▪ Limited visibility into how much of the contract is being tested

1 Coverage-guided fuzzing on VMs is challenging due to execution abstraction

I

Instrumenting ink!

Ink! compiler has constraints. Typically, code is instrumented by compilers (e.g., *afl-clang*). Ink! uses its custom compiler, which lacks native instrumentation

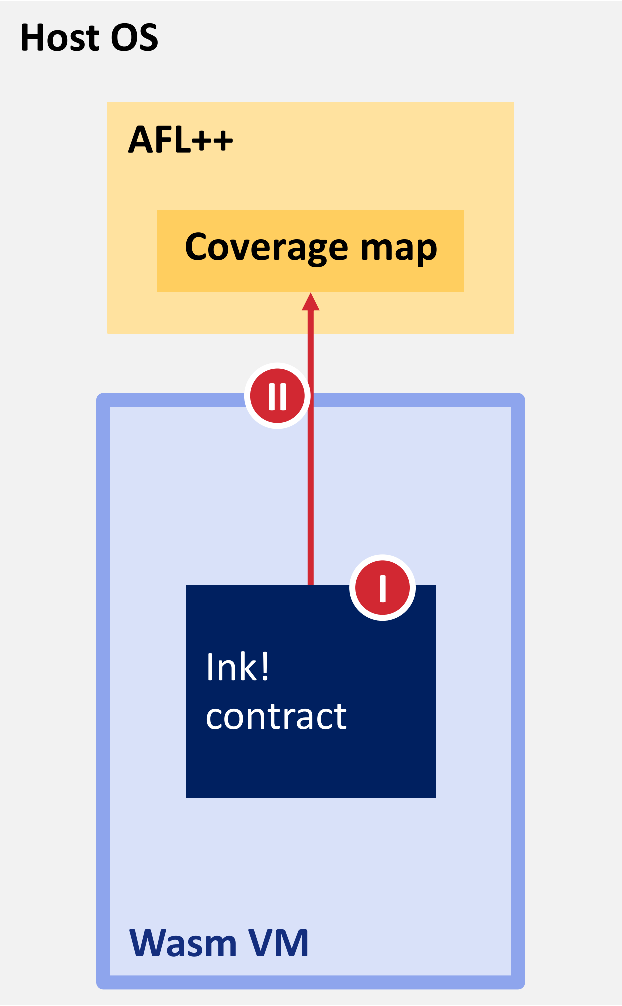
Instrumentation requires support. A version of the ink! compiler must be forked and maintained, or a PR submitted to the ink! compiler, both requiring ongoing maintenance

II

Passing through VM Sandbox

Sandbox restrictions. The Wasm Virtual Machine operates in a sandbox, making it challenging to pass information outside of the VM

Escape the sandbox to transmit coverage to AFL++. We need a way to transmit coverage beyond the sandbox and store it in AFL++'s coverage map



2 Creating effective initial seeds for a fuzzing campaign is difficult

I Impact of Initial Seeds

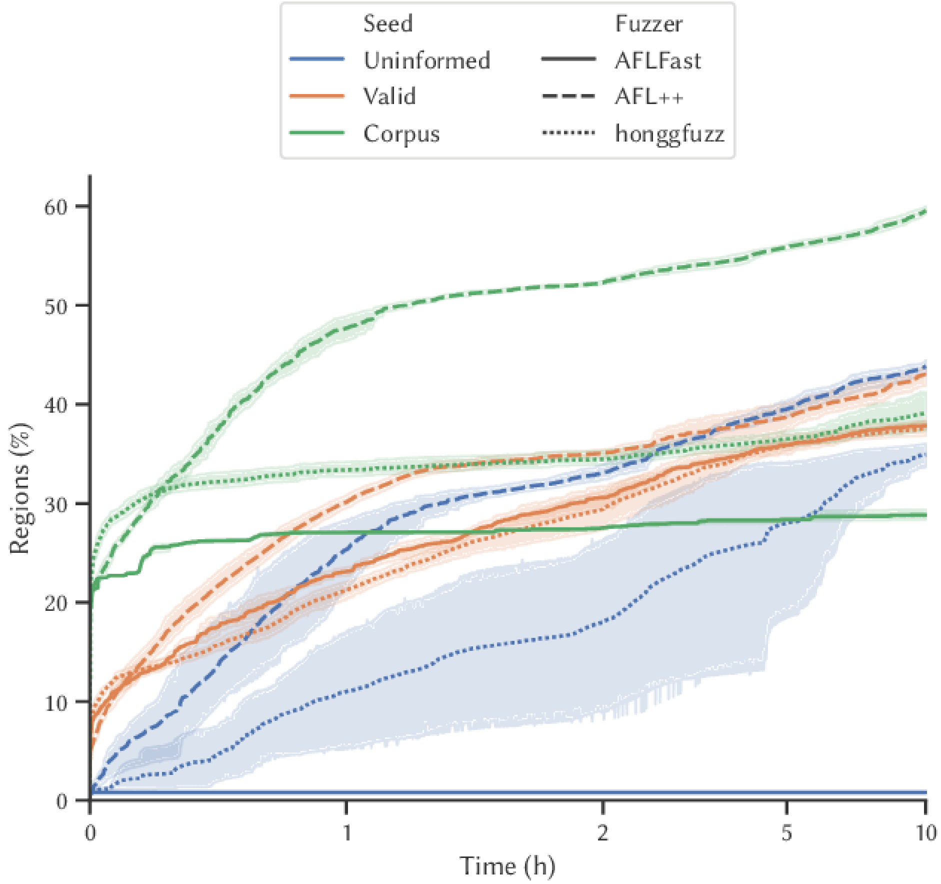
Academia has shown that **selecting appropriate initial** seeds can significantly impact the success of a fuzzing campaign

II Manual Seed Generation

The manual creation of seeds is a **time-consuming process**. Therefore, automating this procedure is favourable

III Seeds Harnessing Source

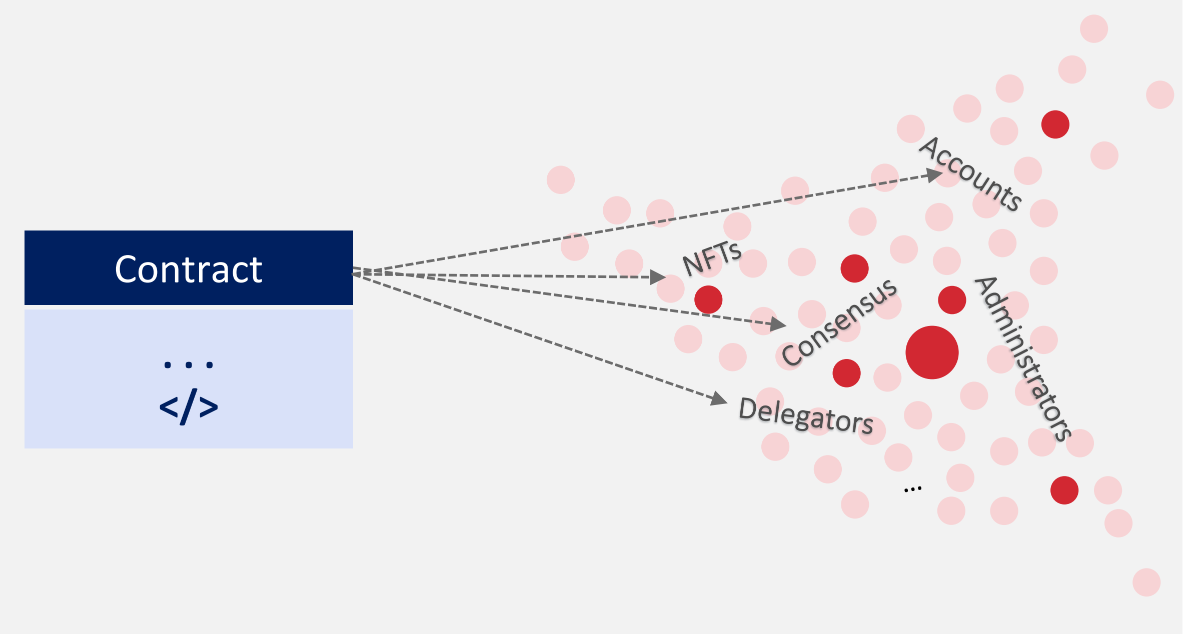
To automatically generate fuzzing seeds, we need to harness them from **a reliable source**



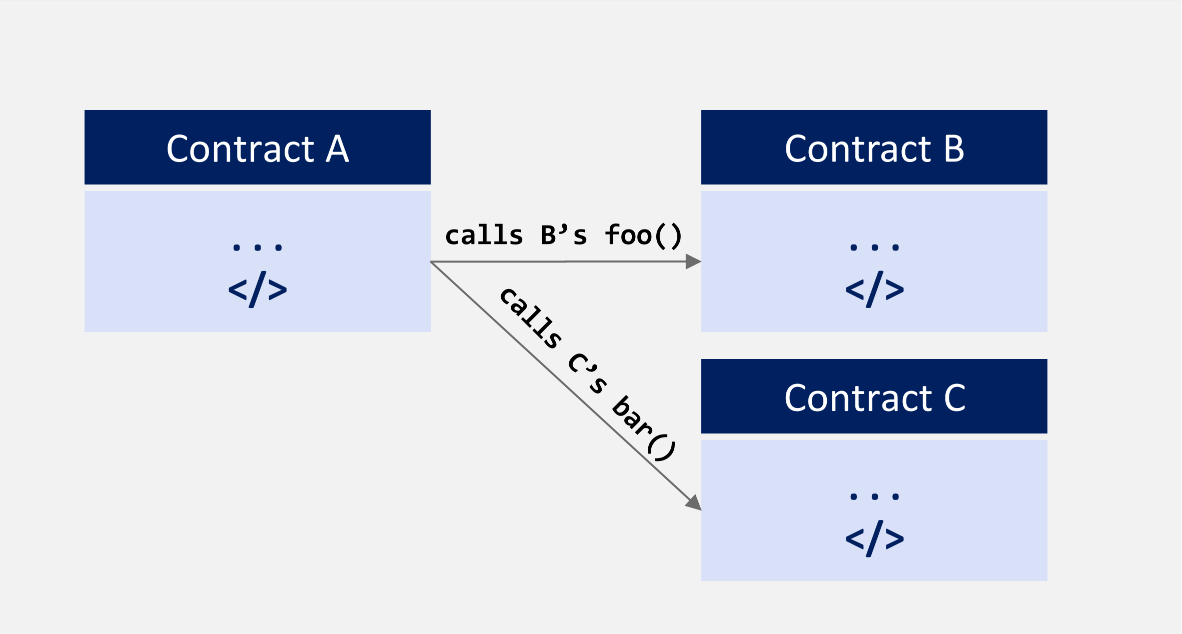
! An initial corpus covering more regions yields higher coverage over time

3 Bootstrapping Smart Contract with stateful execution and on-chain dependencies

I Contract requires on-chain state to work properly



II Contract might interact other deployed contracts



? How can we supply **real-world state** data to ensure the contract functions properly?

? How do we ensure that the contract can **interact** with its **contract dependencies**?

4 Generating coverage reports is crucial to have a successful fuzzing campaign

Coverage analysis is essential for discovering gaps in your fuzzing campaign, for example, through a misconfiguration

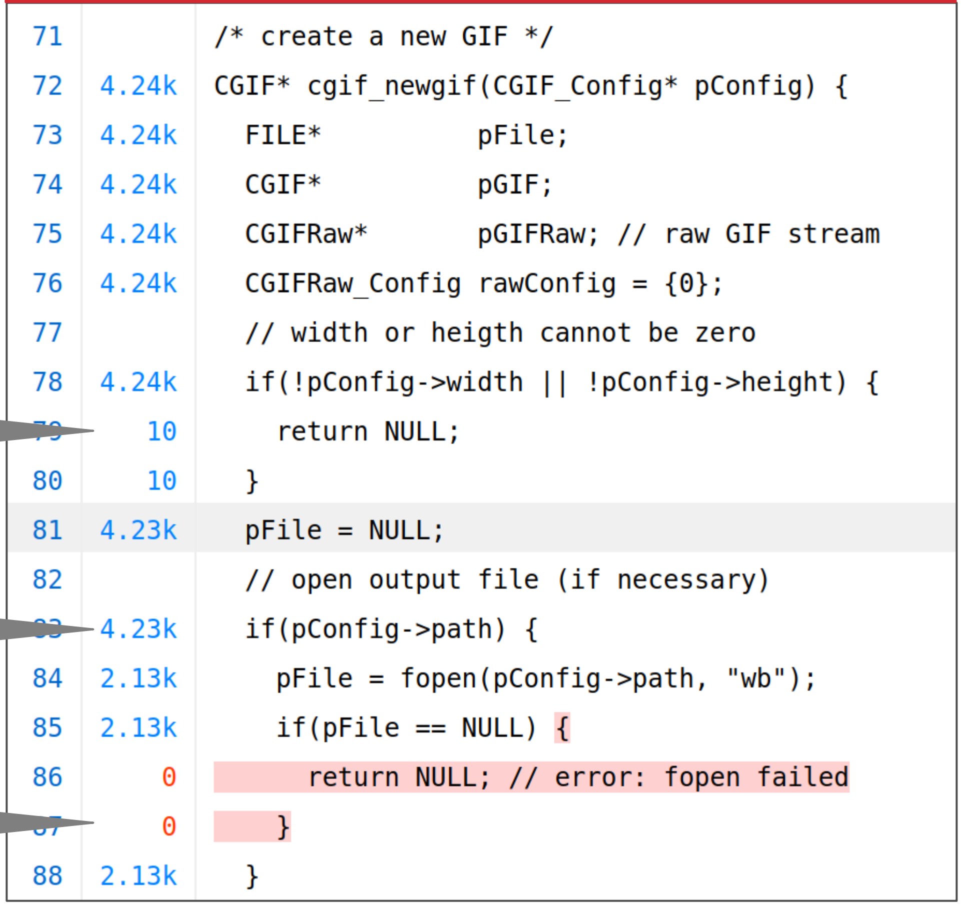
There are tools for generating coverage reports, but what if you have your own coverage system?

I The fuzzer hits this line only a **few times**; therefore, some edge cases may still be untested

II Has been hit almost as much as the function itself and can be considered **well covered**

III This line has **not been hit**, and either custom seeds or adaptation to the corpus might be required

OSS-Fuzz Coverage Report for cgif



```
71         /* create a new GIF */
72  4.24k  CGIF* cgif_newgif(CGIF_Config* pConfig) {
73  4.24k    FILE*          pFile;
74  4.24k    CGIF*          pGIF;
75  4.24k    CGIFRaw*       pGIFRaw; // raw GIF stream
76  4.24k    CGIFRaw_Config rawConfig = {0};
77           // width or heigth cannot be zero
78  4.24k    if(!pConfig->width || !pConfig->height) {
79     10      return NULL;
80     10    }
81  4.23k    pFile = NULL;
82           // open output file (if necessary)
83  4.23k    if(pConfig->path) {
84  2.13k      pFile = fopen(pConfig->path, "wb");
85  2.13k      if(pFile == NULL) {
86      0        return NULL; // error: fopen failed
87      0      }
88  2.13k    }
```

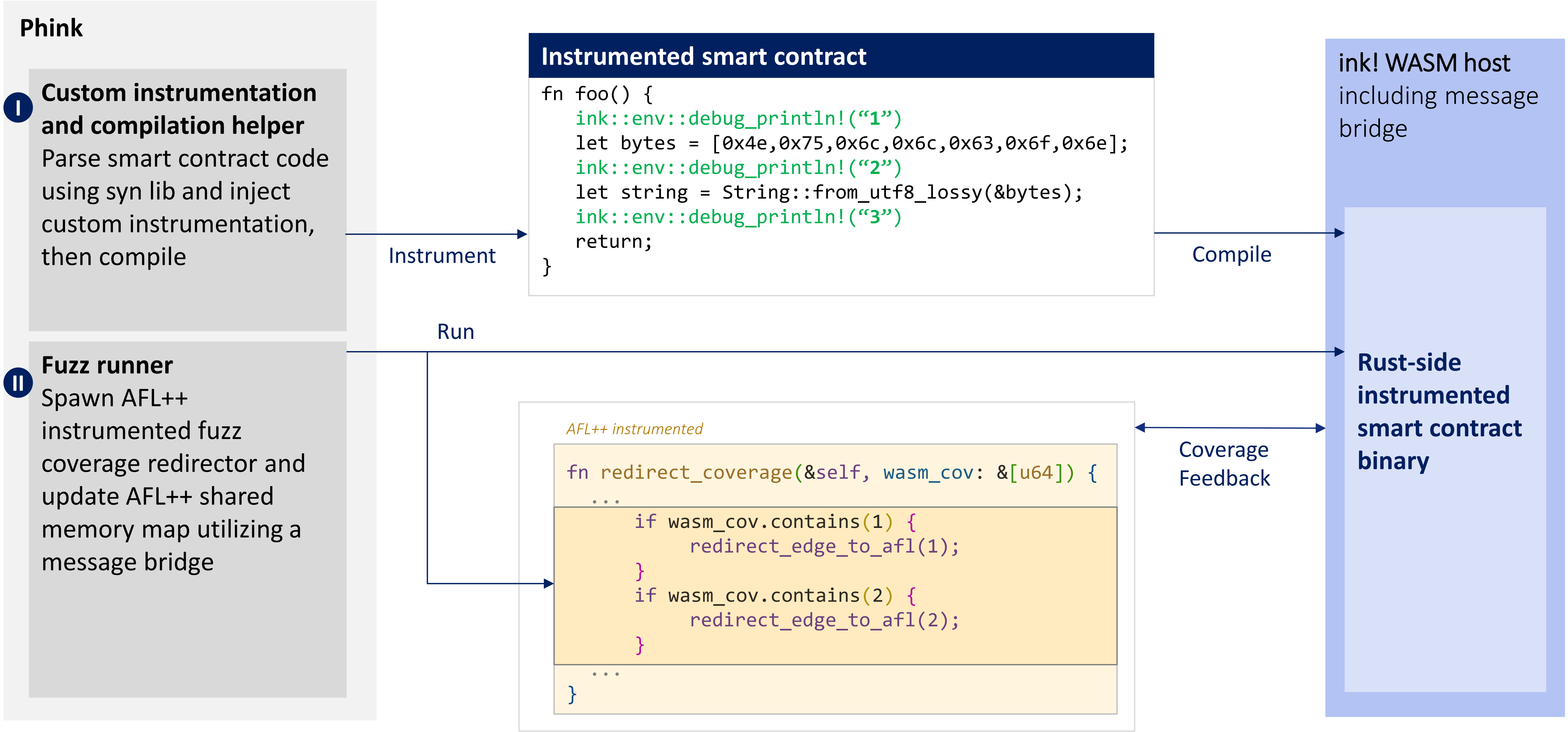
Agenda

1. Overview
2. Background
3. Challenges
4. **Solutions**
5. Success

Each challenge has been addressed with solutions that will now be detailed

Challenge	Solution	Details
1 Execution and instrumentation barriers	A Custom instrumentation and coverage mapping	▪ Phink solves instrumentation by injecting callbacks into contracts. Those callbacks serve as a direct communication from Phink to AFL++
2 Generating initial seeds	B Leverage tests for seed generation	▪ Contract tests are leveraged to generate initial seeds ▪ This provides a fully automatic and reliable method for generating initial seeds for the fuzzing campaign
3 Stateful execution and on-chain dependencies	C On-chain contract emulation and genesis state	▪ Phink enables developers to integrate ready-to-fuzz contract dependencies and define a genesis state, creating a rich execution environment for stateful fuzzing
4 Coverage and feedback limitations	D Coverage reports	▪ Phink's tracking of every executed statement allows to generate coverage reports which improve with monitoring and assessment of fuzzing campaigns

A Phink solves instrumentation via coverage remapping and message bridging



Phink

I **Custom instrumentation and compilation helper**
Parse smart contract code using syn lib and inject custom instrumentation, then compile

II **Fuzz runner**
Spawn AFL++ instrumented fuzz coverage redirector and update AFL++ shared memory map utilizing a message bridge

Instrument

Instrumented smart contract

```
fn foo() {
   ink::env::debug_println!(“1”)
   let bytes = [0x4e,0x75,0x6c,0x6c,0x63,0x6f,0x6e];
   ink::env::debug_println!(“2”)
   let string = String::from_utf8_lossy(&bytes);
   ink::env::debug_println!(“3”)
   return;
}
```

Compile

Run

AFL++ instrumented

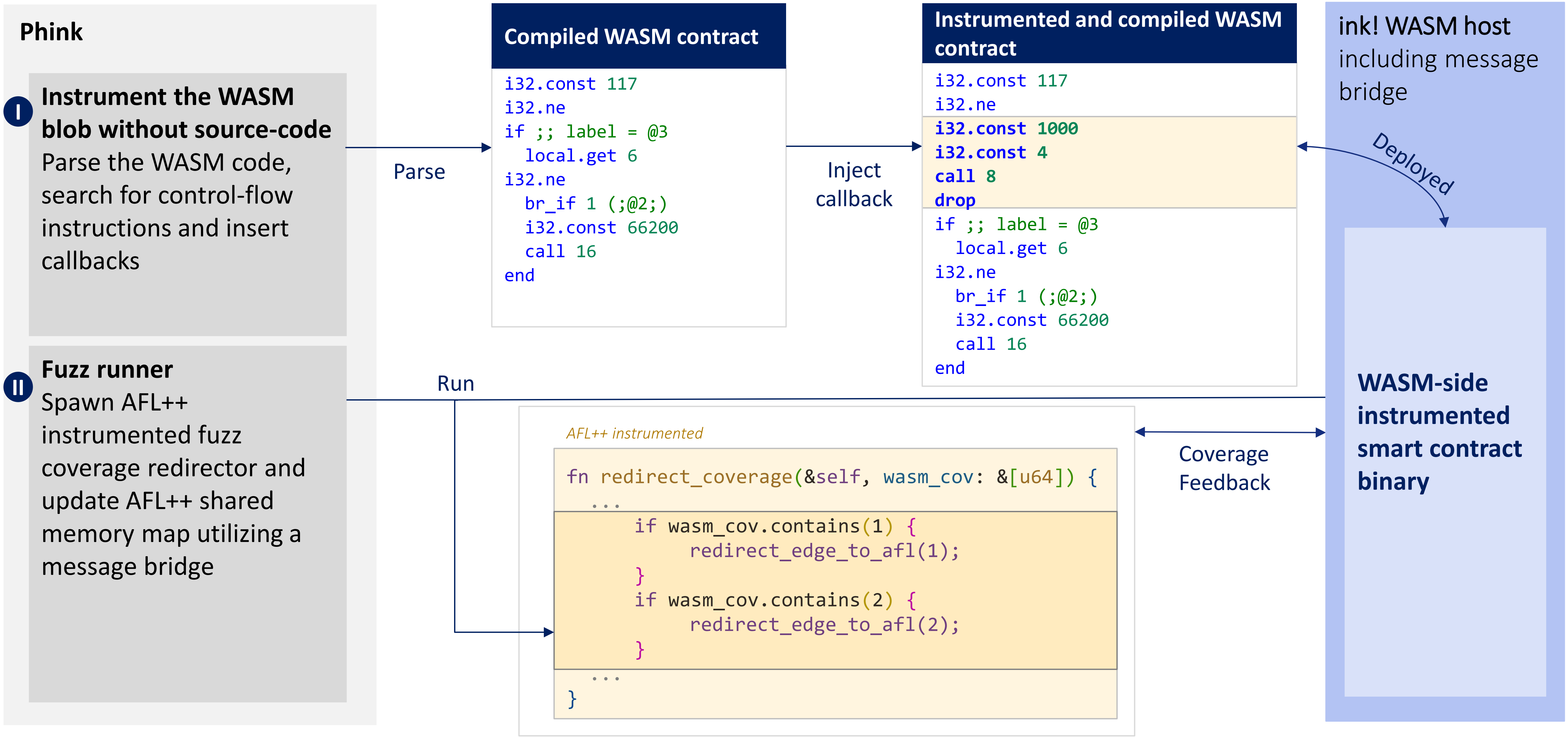
```
fn redirect_coverage(&self, wasm_cov: &[u64]) {
  ...
     if wasm_cov.contains(1) {
         redirect_edge_to_afl(1);
     }
     if wasm_cov.contains(2) {
         redirect_edge_to_afl(2);
     }
  ...
}
```

Coverage Feedback

ink! WASM host including message bridge

Rust-side instrumented smart contract binary

A Alternative approach: WASM blob instrumentation for resolving partial coverage & enabling black-box fuzzing



Phink

I Instrument the WASM blob without source-code

Parse the WASM code, search for control-flow instructions and insert callbacks

II Fuzz runner

Spawn AFL++ instrumented fuzz coverage redirector and update AFL++ shared memory map utilizing a message bridge

Compiled WASM contract

```
i32.const 117
i32.ne
if ;; label = @3
  local.get 6
i32.ne
  br_if 1 (;@2;)
  i32.const 66200
  call 16
end
```

Instrumented and compiled WASM contract

```
i32.const 117
i32.ne
i32.const 1000
i32.const 4
call 8
drop
if ;; label = @3
  local.get 6
i32.ne
  br_if 1 (;@2;)
  i32.const 66200
  call 16
end
```

AFL++ instrumented

```
fn redirect_coverage(&self, wasm_cov: &[u64]) {
  ...
      if wasm_cov.contains(1) {
          redirect_edge_to_afl(1);
      }
      if wasm_cov.contains(2) {
          redirect_edge_to_afl(2);
      }
  ...
}
```

ink! WASM host including message bridge

WASM-side instrumented smart contract binary

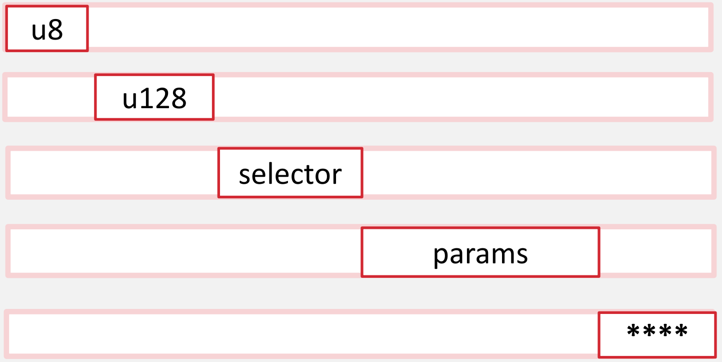
B Phink defines a structured format for inputs, designed specifically for message execution

Fuzzing Input Structure

- **Receive** fuzzing input as **bytes array** from AFL++
- **Parse** the bytes into a Message struct
- **Iterate** over next Message if the seed has a delimiter
- **Results** in a Vector of Message

Message structure

```
#[derive(Debug, Clone, Serialize)]
pub struct Message {
    origin: Origin,
    value_token: BalanceOf<Runtime>,
    payload: Vec<u8>,
}
```



Multi-message structure example

Example Message 1

Bob	231	send_to	{ account: 5D35…x}	****

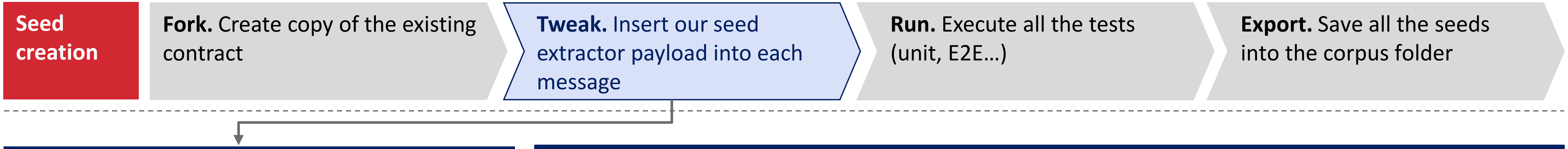
● ● ●

Example Message N

Alice	420	send_to	{ account: 5D35…x}	****

B

Contract tests are leveraged to generate initial seeds



Seed creation

Fork. Create copy of the existing contract

Tweak. Insert our seed extractor payload into each message

Run. Execute all the tests (unit, E2E…)

Export. Save all the seeds into the corpus folder

Tweak phase

1 For each `#[ink(message)]`

2 **Grab** arguments + function name

3 Prepare a snippet that **SCALE-encode** the `message` selector + parameters

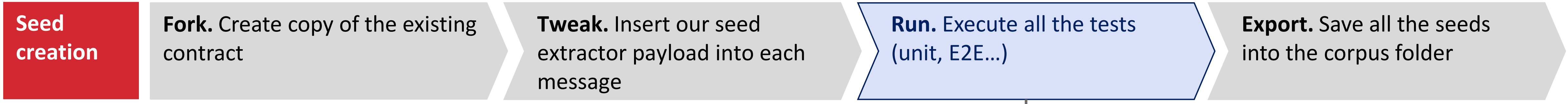
4 **Insert** that snippet at the beginning of the `message`

Message is tweaked to output the encoded seed when called

```
#[ink(message)]
pub fn crash_with_invariant(&mut self, data: String) -> Result<()> {
{
    let sel = ExecutionInput::new(selector_bytes!("crash_with_invariant"))
                  .push_arg(&data);

    let encoded = scale::Encode::encode(&sel);
    ink::env::debug_println!("ENCODED_SEED={}", encoded.iter()
        .map(|byte| format!("{:02x}", byte))
        .collect::<String>());
}
// Actual message logic below
if data.len() < 7 && data.len() > 3 {
     ...
    }
}
Ok(())
}
```

Inserted snippet

B Contract tests are leveraged to generate initial seeds



Seed creation

Fork. Create copy of the existing contract

Tweak. Insert our seed extractor payload into each message

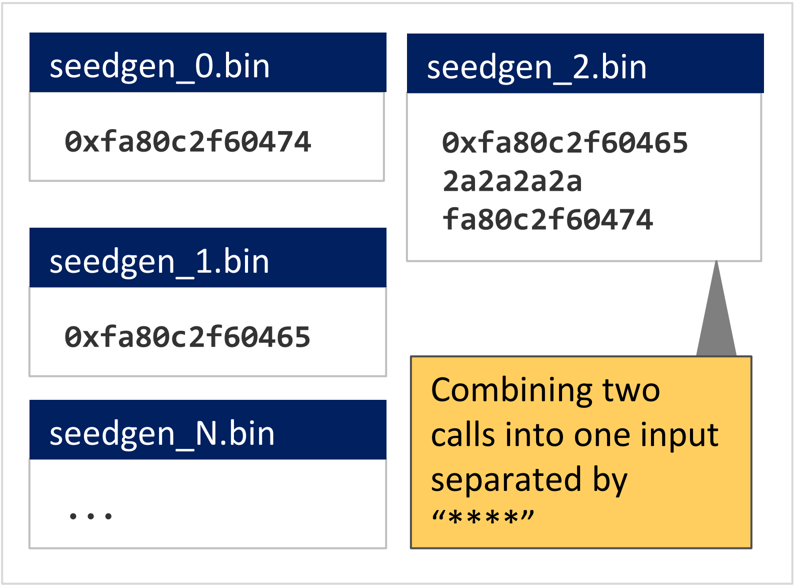
Run. Execute all the tests (unit, E2E…)

Export. Save all the seeds into the corpus folder

Tests are executed and saved as valid seeds

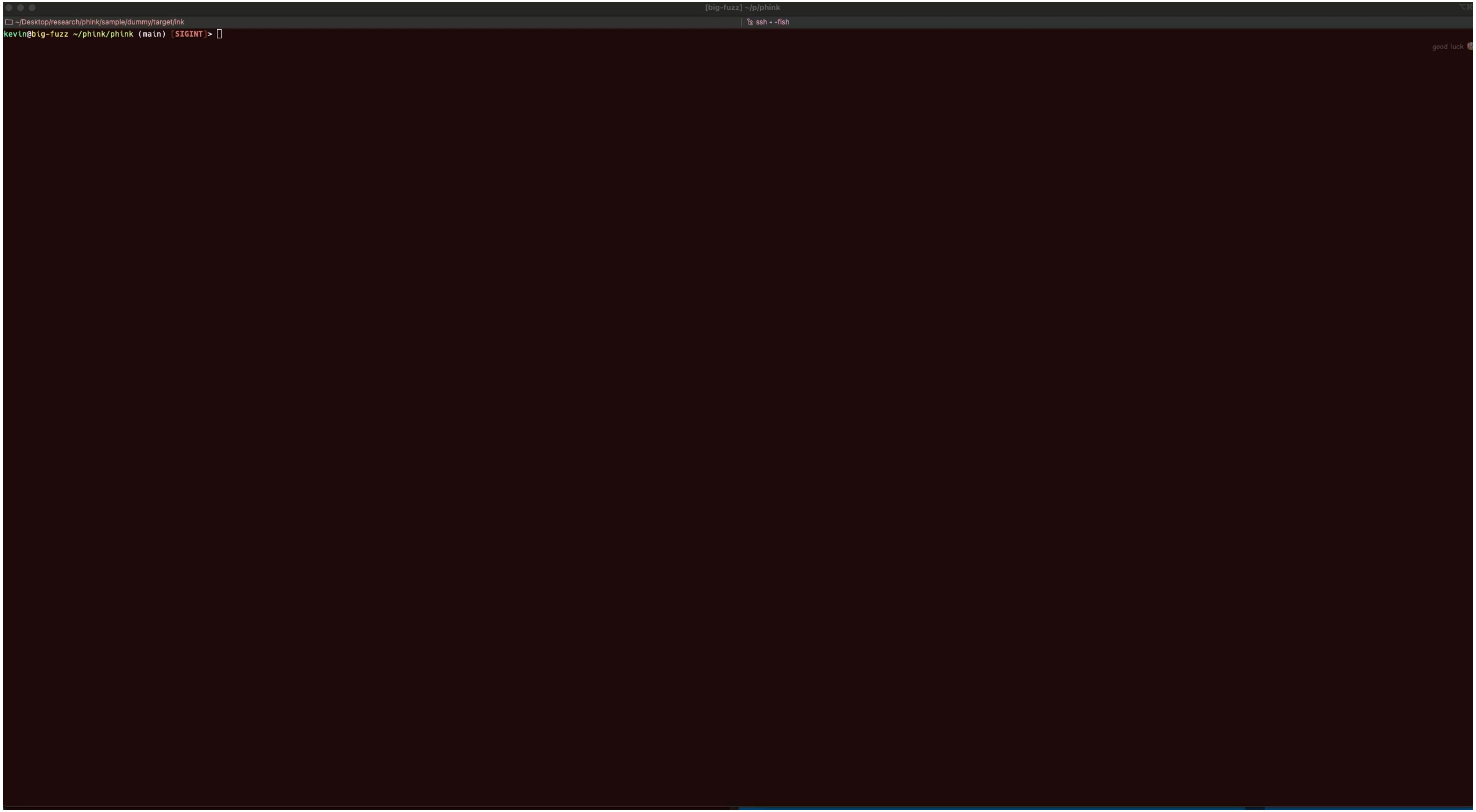
```
$ cargo run -- generate-seed sample/dummy/

running 3 tests
test dummy::e2e_tests::it_works ... ok
test dummy::tests::for_seedgen ... ok
test dummy::tests::new_works ... ok

ENCODED_SEED=fa80c2f60474
ENCODED_SEED=fa80c2f60465

Writing bytes 0xfa80c2f60474 to
`output/phink/corpus/seedgen_0.bin`
Writing bytes 0xfa80c2f60465 to
`output/phink/corpus/seedgen_1.bin`
```

Corpus directory



Combining two calls into one input separated by "****"

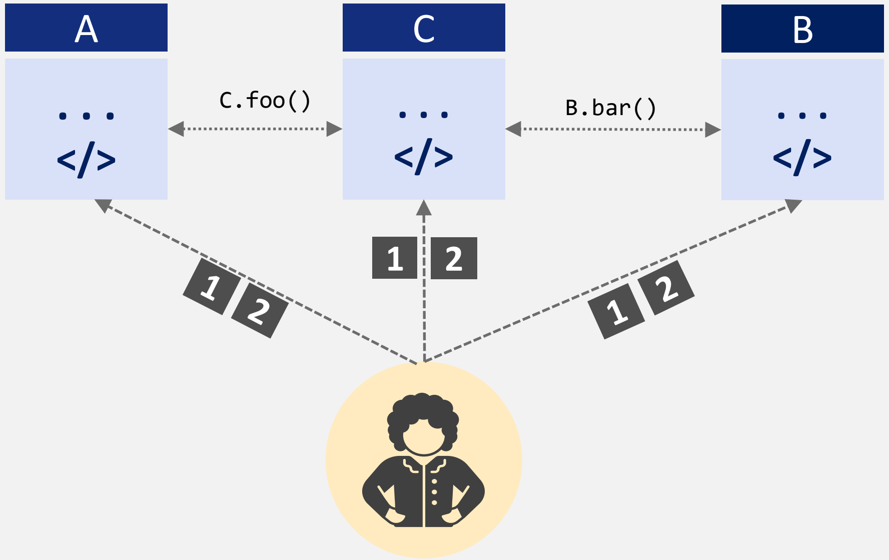
B Contract tests are leveraged to generate initial seeds



```
kevin@big-fuzz ~/phink/phink (main) [SIGINT]> 
```

C Integrating other contracts and defining ”genesis” state can solve emulation constraints

Cross-contract calls



1 **Instrument** every contract

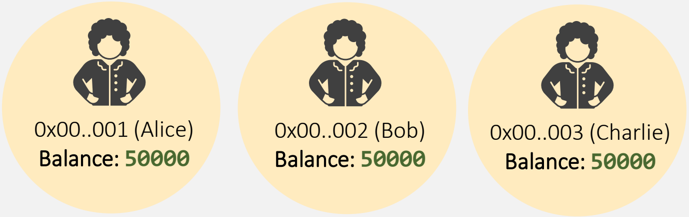
2 **Deploy** every contract

3 **Start** fuzzing, the coverage of A, B and C will be collected

Mocking on-chain state

Developers can insert a mocked environment

```
impl DevelopperPreferences for Preferences {
    fn runtime_storage() -> Storage {
        let storage = RuntimeGenesisConfig {
            balances: BalancesConfig {
                // Lot of money for Alice, Bob..
                balances: (0..u8::MAX)
                    .map(|i| [i; 32].into())
                    .map(|k| (k, 50000))
                    .collect(),
            },
        }.build_storage()
    }
    ...
}
```



0x00..001 (Alice)
Balance: 50000

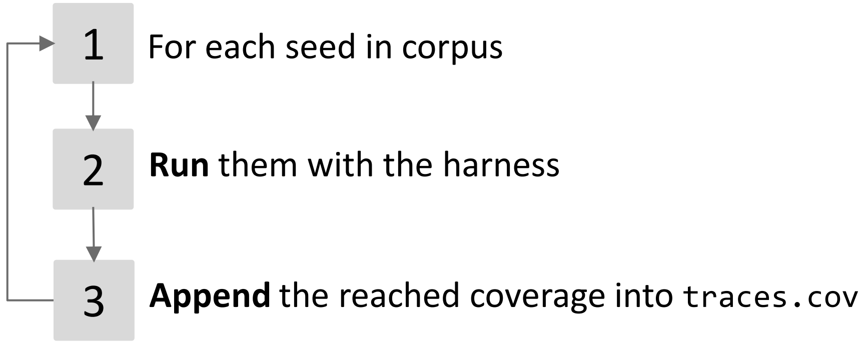
0x00..002 (Bob)
Balance: 50000

0x00..003 (Charlie)
Balance: 50000

D Users can create coverage reports for their contract

Phink tracks every executed statement and allows to generate coverage reports

I Generating a .cov file

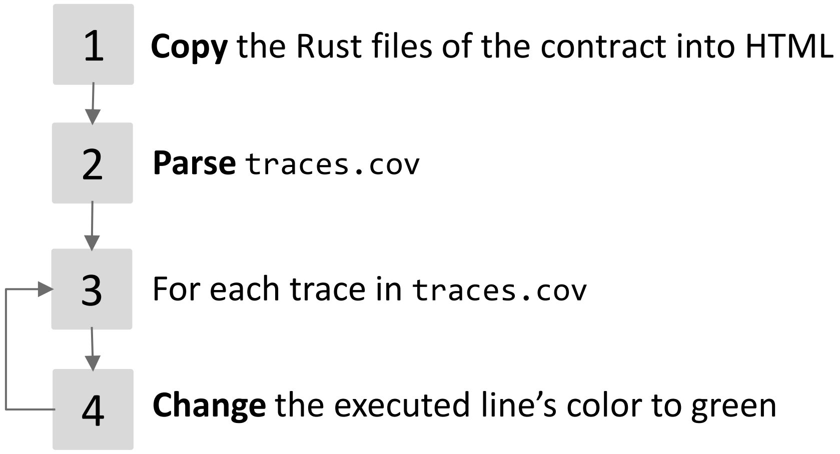


1 For each seed in corpus

2 **Run** them with the harness

3 **Append** the reached coverage into `traces.cov`

II Parsing and generating HTML



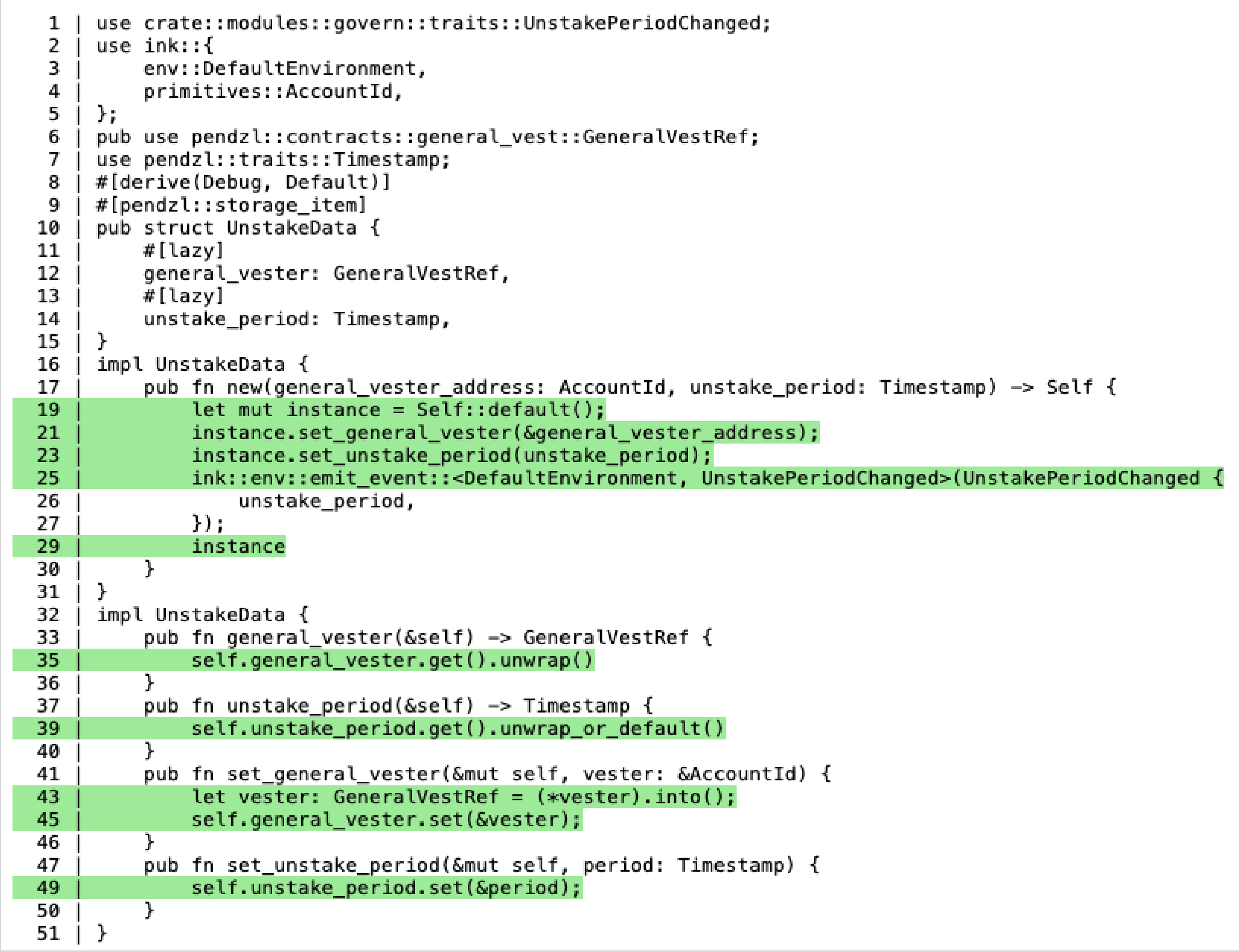
1 **Copy** the Rust files of the contract into HTML

2 **Parse** `traces.cov`

3 For each trace in `traces.cov`

4 **Change** the executed line's color to green

Coverage report sample

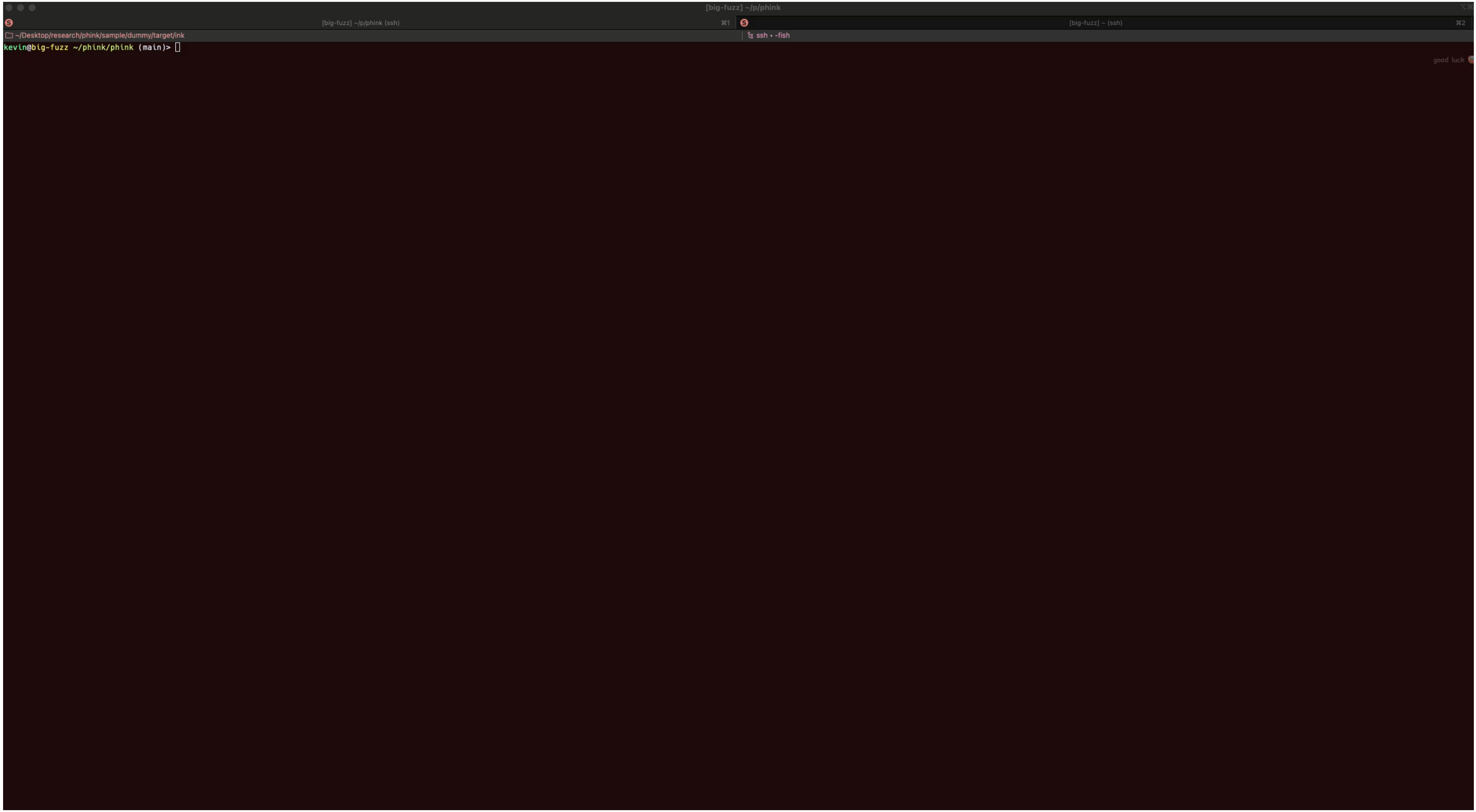


```
use crate::modules::govern::traits::UnstakePeriodChanged;
use ink::{
    env::DefaultEnvironment,
    primitives::AccountId,
};
pub use pendzl::contracts::general_vest::GeneralVestRef;
use pendzl::traits::Timestamp;
#[derive(Debug, Default)]
#[pendzl::storage_item]
pub struct UnstakeData {
    #[lazy]
    general_vester: GeneralVestRef,
    #[lazy]
    unstake_period: Timestamp,
}
impl UnstakeData {
    pub fn new(general_vester_address: AccountId, unstake_period: Timestamp) -> Self {
        let mut instance = Self::default();
        instance.set_general_vester(&general_vester_address);
        instance.set_unstake_period(unstake_period);
        ink::env::emit_event::<DefaultEnvironment, UnstakePeriodChanged>(UnstakePeriodChanged {
            unstake_period,
        });
        instance
    }
}
impl UnstakeData {
    pub fn general_vester(&self) -> GeneralVestRef {
        self.general_vester.get().unwrap()
    }
    pub fn unstake_period(&self) -> Timestamp {
        self.unstake_period.get().unwrap_or_default()
    }
    pub fn set_general_vester(&mut self, vester: &AccountId) {
        let vester: GeneralVestRef = (*vester).into();
        self.general_vester.set(&vester);
    }
    pub fn set_unstake_period(&mut self, period: Timestamp) {
        self.unstake_period.set(&period);
    }
}
```

Security Research Labs

D

Users can create coverage reports for their contract



[big-fuzz] ~/p/phink
[big-fuzz] ~/p/phink (ssh)
⌘1
[big-fuzz] ~ (ssh)
⌘2
~/Desktop/research/phink/sample/dummy/target/ink
ssh ◂ -fish
good luck

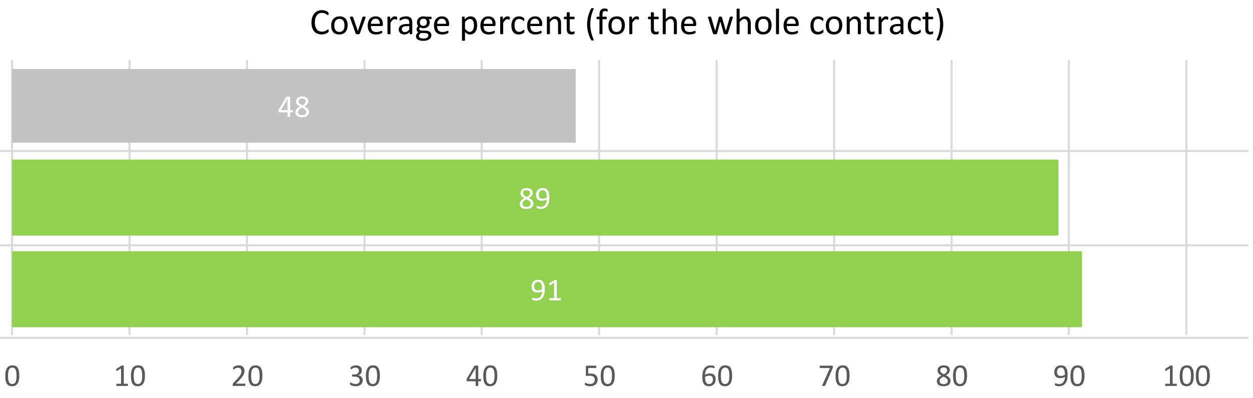
```
kevin@big-fuzz ~/phink/phink (main)> 
```

Agenda

1. Overview
2. Background
3. Challenges
4. Solutions
5. **Success**

Phink is now the industry standard fuzzer for ink! smart contracts

	Seedgen	Avg speed*	Corpus entries
AbaxFinance/dao-contracts	✗	1500/100	1639
ink-examples/erc1155	✓	1300/140	949
ink-examples/multisig	✓	1400/113	1524



* First value is early phase; second value is late phase. Measured in executions per second, for 1 core

Forkless fuzzer

Phink doesn't need a fork of ink, pallet_contract, substrate, polkadot or cargo-contract

VM-agnostic

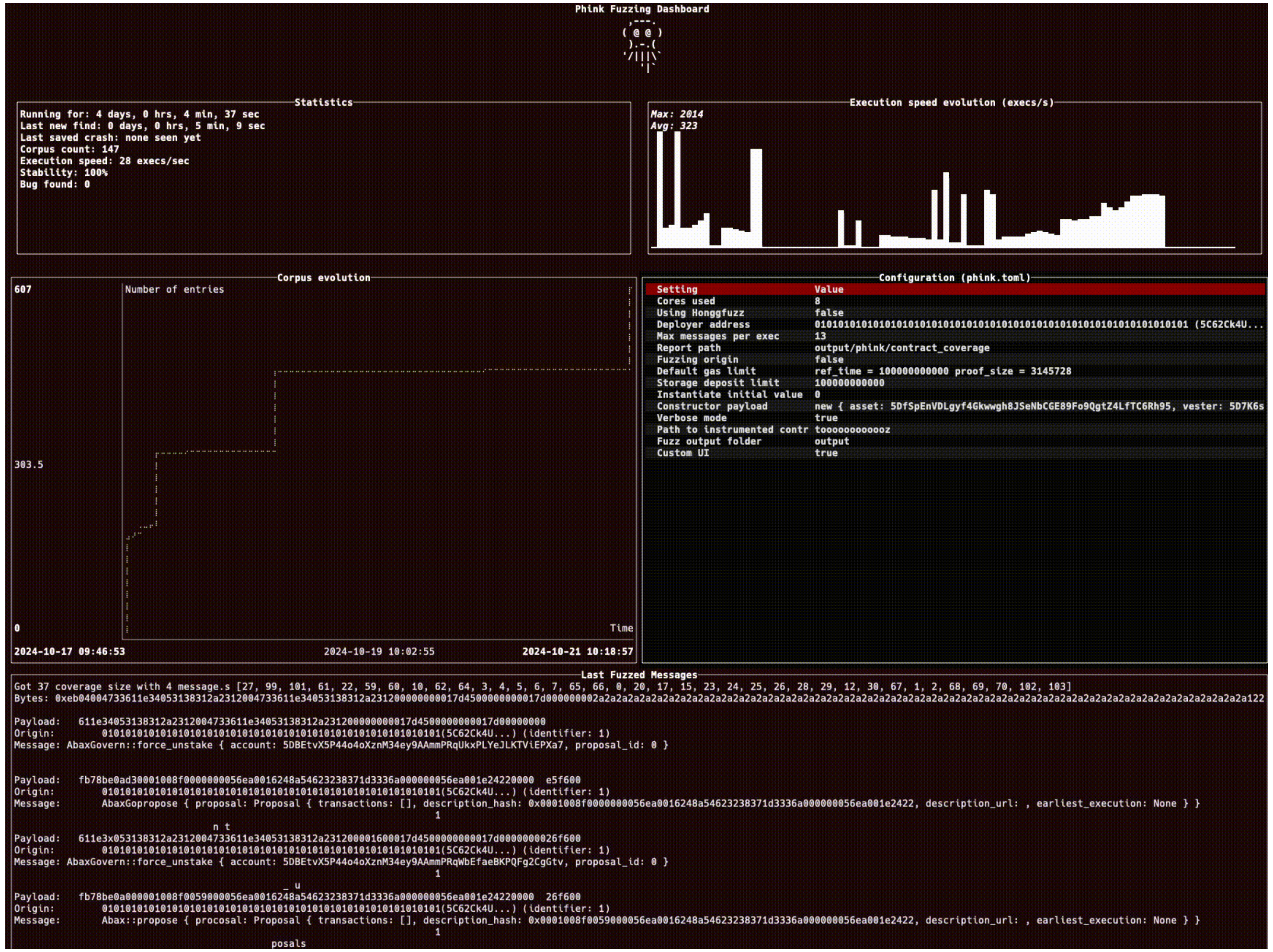
ink! contract compiled into WASM or RISC-V (newly supported) can be fuzzed by Phink

Fully coverage guided

With in-WASM instrumentation, the contract is fully instrumented on every control-flow

Blackbox fuzzing

Since Phink can instrument compiled WASM blobs, source-code is not required



```
Phink Fuzzing Dashboard
 ,---.
( @ @ )
 ).-.(
'/|||\`
  '|`

Statistics
Running for: 4 days, 0 hrs, 4 min, 37 sec
Last new find: 0 days, 0 hrs, 5 min, 9 sec
Last saved crash: none seen yet
Corpus count: 147
Execution speed: 28 execs/sec
Stability: 100%
Bug found: 0

Execution speed evolution (execs/s)
Max: 2014
Avg: 323

Corpus evolution
607        Number of entries
303.5
0                                                            Time
2024-10-17 09:46:53        2024-10-19 10:02:55        2024-10-21 10:18:57

Configuration (phink.toml)
Setting                        Value
Cores used                     8
Using Honggfuzz                false
Deployer address               0101010101010101010101010101010101010101010101010101010101010101 (5C62Ck4U...
Max messages per exec          13
Report path                    output/phink/contract_coverage
Fuzzing origin                 false
Default gas limit              ref_time = 100000000000 proof_size = 3145728
Storage deposit limit          100000000000
Instantiate initial value      0
Constructor payload            new { asset: 5DfSpEnVDLgyf4Gkwwgh8JSeNbCGE89Fo9QgtZ4LfTC6Rh95, vester: 5D7K6s
Verbose mode                   true
Path to instrumented contr     tooooooooooz
Fuzz output folder             output
Custom UI                      true

Last Fuzzed Messages
Got 37 coverage size with 4 message.s [27, 99, 101, 61, 22, 59, 60, 10, 62, 64, 3, 4, 5, 6, 7, 65, 66, 0, 20, 17, 15, 23, 24, 25, 26, 28, 29, 12, 30, 67, 1, 2, 68, 69, 70, 102, 103]
Bytes: 0xeb04004733611e34053138312a2312004733611e34053138312a231200000000017d4500000000017d000000002a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a122

Payload:   611e34053138312a2312004733611e34053138312a231200000000017d4500000000017d00000000
Origin:        0101010101010101010101010101010101010101010101010101010101010101(5C62Ck4U...) (identifier: 1)
Message: AbaxGovern::force_unstake { account: 5DBEtvX5P44o4oXznM34ey9AAmmPRqUkxPLYeJLKTViEPXa7, proposal_id: 0 }


Payload:   fb78be0ad30001008f0000000056ea0016248a54623238371d3336a000000056ea001e24220000  e5f600
Origin:        0101010101010101010101010101010101010101010101010101010101010101(5C62Ck4U...) (identifier: 1)
Message:       AbaxGopropose { proposal: Proposal { transactions: [], description_hash: 0x0001008f0000000056ea0016248a54623238371d3336a000000056ea001e2422, description_url: , earliest_execution: None } }
                                                    1
                      n t
Payload:   611e3x053138312a2312004733611e34053138312a23120000160017d4500000000017d0000000026f600
Origin:        0101010101010101010101010101010101010101010101010101010101010101(5C62Ck4U...) (identifier: 1)
Message: AbaxGovern::force_unstake { account: 5DBEtvX5P44o4oXznM34ey9AAmmPRqWbEfaeBKPQFg2CgGtv, proposal_id: 0 }
                                                    1
                            _ u
Payload:   fb78be0a000001008f0059000056ea0016248a54623238371d3336a000000056ea001e24220000  26f600
Origin:        0101010101010101010101010101010101010101010101010101010101010101(5C62Ck4U...) (identifier: 1)
Message:       Abax::propose { procosal: Proposal { transactions: [], description_hash: 0x0001008f0059000056ea0016248a54623238371d3336a000000056ea001e2422, description_url: , earliest_execution: None } }
                                                    1
                            posals
```

Thanks! 🐙

https://github.com/srlabs/phink

https://srlabs.github.io/phink/