Bugs in Blocks

HITB Phuket, Aug 24, 2023

Karsten Nohl <nohl@srlabs.de>

Security
Research
Labs

Nice to meet you :)

The SRLabs heroes behind this research

Karsten Nohl

- Studied cryptography, not crypto! =)
- Founder of SRLabs
- Research track record in mobile network security
- Former CISO at large telcos
- Lives in Bangkok

Louis Merlin

- Security researcher, focusing on blockchain security and threat analysis
- Develops fuzz testing tools

Gabriel Arnautu

- Security researcher, conducting blockchain security reviews
- Focuses on tool development for audit automation

Blockchain technology, love it or hate it, is continuously evolving; Researchers are hardly keeping up

The New York Times

The Crypto World Is on Edge After a String of Hacks

More than $2 billion in digital currency has been stolen in hacks this year, shaking faith in the experimental field of decentralized finance, known as DeFi.

Research question – How do we proactively find bugs in large blockchain ecosystems?

This talk discusses five types of common blockchain bugs, and how to find them.

Bugs are explosive in crypto: Single-line integer overflow caused cryptocurrency to implode

Background

- YAM launched in 2020 and quickly attracted >$500 million in assets
- The project founders warned about its immaturity and the lack of security auditing
- A bug caused the coin to lose control of its on-chain governance feature

Vulnerable code snippet in YAM Finance rebase logic

```
...
    totalSupply = initSupply.mul(yamsScalingFactor);
    emit Rebase(epoch, prevYamsScalingFactor, yamsScalingFactor);
    return totalSupply;
}
...
```

The rebase function aims to maintain token price stability. However, **due to an integer overflow it incorrectly calculates the totalSupply**, resulting in an excessive reserve of minted tokens

Impact

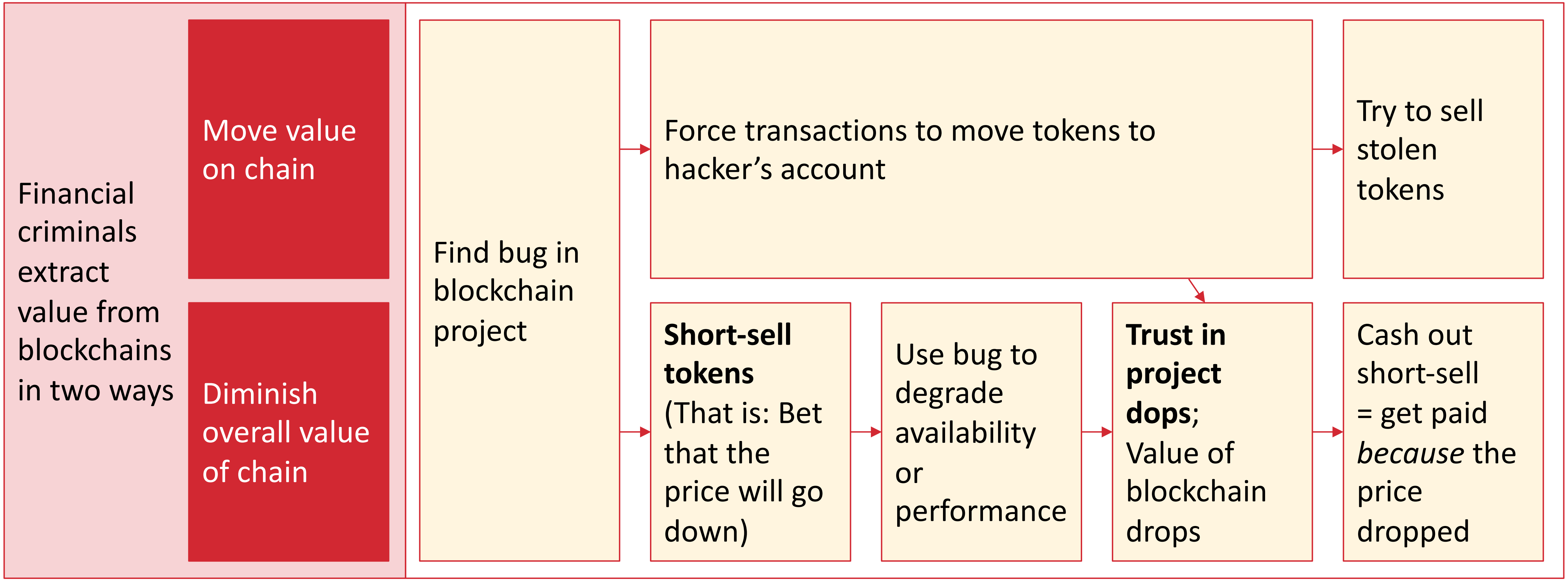
- Efforts to regain control of the YAM treasury failed
- YAM's total market capitalization dropped from $65 million to **zero** in a few hours
- This event shows how a single vulnerability in a single line of code can compromise a whole project and the consumer funds behind it

brock
@brockjelmore · **Follow**

i'm sorry everyone. i've failed. thank you for the insane support today. i'm sick with grief

10:01 AM · Aug 13, 2020

Criminals cash-out blockchain programming bugs in two ways



Financial criminals extract value from blockchains in two ways

Move value on chain

Diminish overall value of chain

Find bug in blockchain project

Force transactions to move tokens to hacker's account

Try to sell stolen tokens

Short-sell tokens (That is: Bet that the price will go down)

Use bug to degrade availability or performance

Trust in project dops; Value of blockchain drops

Cash out short-sell = get paid *because* the price dropped

Out of scope for this presentation: Other methods criminals use to defraud blockchains

- Abuse of a chain's business logic ('economic attacks')
- Hacking the underlying IT infrastructure or web/mobile apps
- Social engineering and other attacks on blockchain endusers
- Financial scams

Agenda

- **Intro to third-generation blockchains**
- Five types of blockchain hacking
- Fuzzing blockchains effectively

Third-generation blockchains set out to solve scalability and interoperability

1st generation blockchains

- Technology to transact with one another (at a peer-to-peer level)
- Users do not need to rely on centralized entities such as banks
- The design only allows you to send, receive and trade assets

2nd generation blockchains

- Introduces smart contracts, self-executing agreements made between parties
- Allows executing agreements without relying on an expensive intermediary to manage it
- Behaves as digital ecosystem that other crypto projects operate on
- Does not scale well, slowing down transactions

Our focus today:

3rd generation blockchains

- Solves 2nd gen issue of scalability by creating more parallel transaction and more storage
- Introduces interoperability, allowing blockchains to interact with one another
- Adds more flexibility towards networking, node, and runtime configuration, empowering custom-purpose blockchains

Distributed Ledger (Bitcoin)
2008

Smart Contracts (Ethereum)
2013

Interoperability (e.g. Polkadot)
2017

Substrate is a framework to program "third-gen" chains

Substrate is a programming framework to build blockchains …

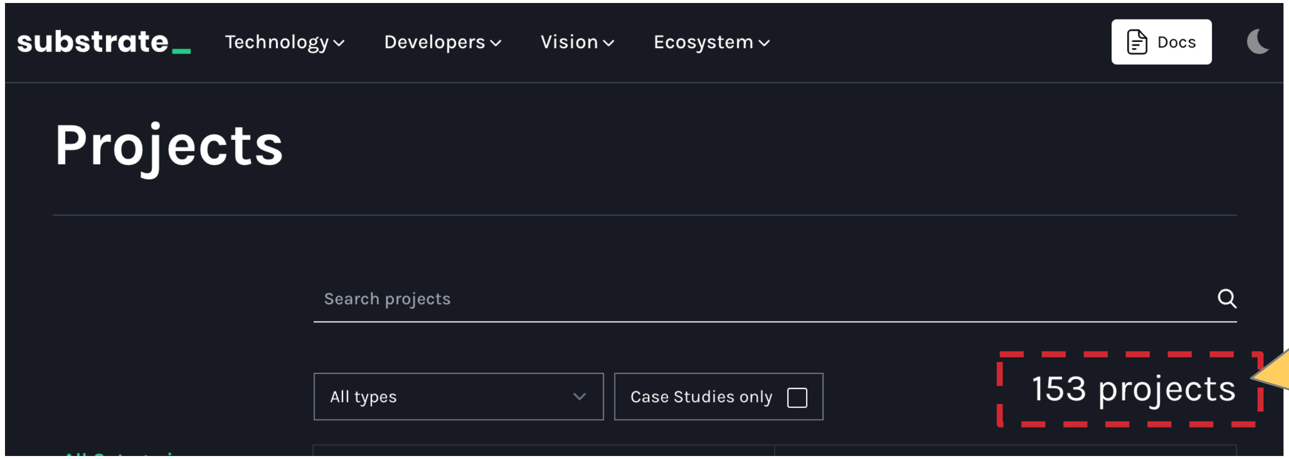
Engineering challenge

Developers found they were recreating much of the same functionality but with different limitations around **scale**, **governance**, **forks**, **interoperability**, and **upgrades**

Their solution

Substrate in a nutshell provides
- Tooling for development, deployment, debugging
- Blockchains to upgrade forkless
- Hot-swap components (pallets) such as the network stack, consensus, finality engine

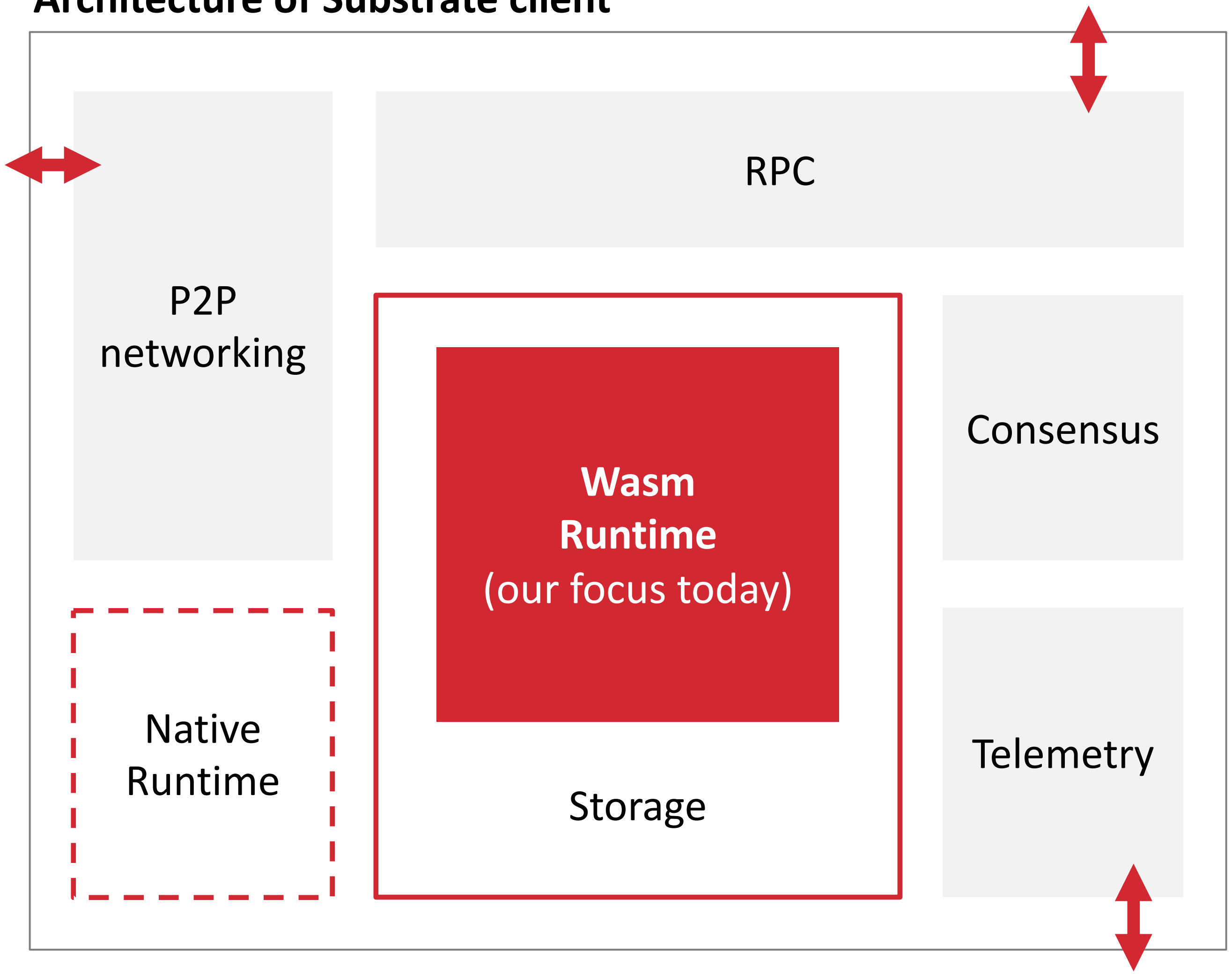
… and has been adopted for over a hundred blockchain projects



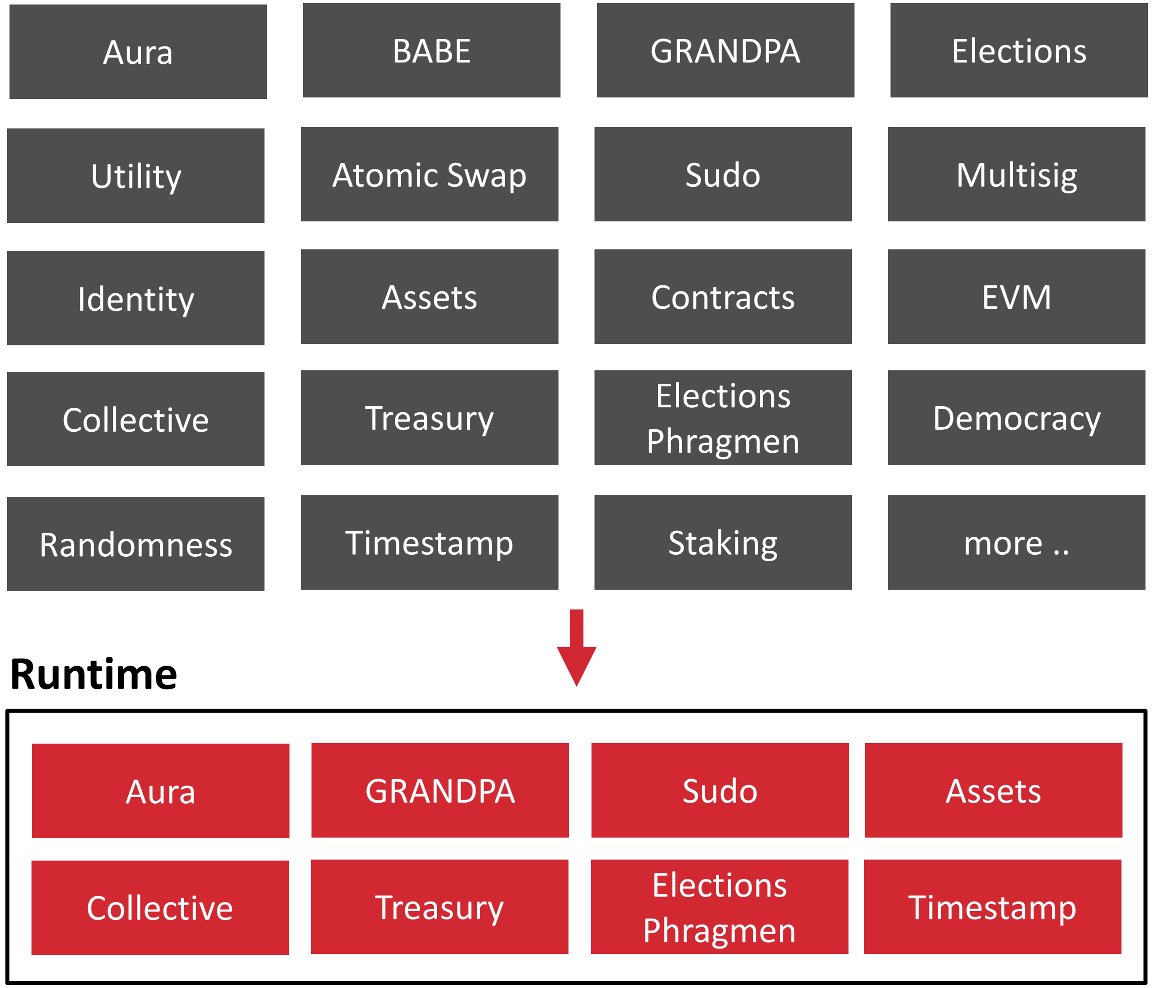
Substrate is actively used by **153 teams** building blockchain projects, making it a **relevant security research target**

Substrate is the foundation of different blockchain projects; Scalable methods and toolchain needed for vulnerability testing

Architecture of Substrate client



FRAME pallets



Agenda

- Intro to third-generation blockchains
- **Five types of blockchain hacking**
- Fuzzing blockchains effectively

Five types of hacking attacks are commonly possible against third-gen blockchains

Example hacking goal	Bug type	Bug impact	Availa-bility	Integrity	Attack scope
Exhaust resources	A Wrongly-priced transactions	Spam a wrongly-priced transaction can cause a DoS	■	□	These hacks may be leveraged against q wide range of blockchain projects, as they do not require any secret information from the victim; and most configurations and program source code is open
Manipulate program flow	B Unsafe arithmetic Logic bugs	Abuse an operation to edit values to your advantage	■	■	
DoS nodes	C Reachable panics	Cause every available node to panic and cause a DoS	■	□	
Abuse business logic	D Incorrect usage of standard patterns	Abuse a misconfiguration to gain a financial advantage or cause DoS	■	■	
Slow down chain	E Storage bloating	Reduce chain useability by filling its storage	■	□	

A Underpriced function calls enable resource exhaustion

Hacking goal	Bug type	Bug impact	Avail.	Integrity
Exhaust resources	Wrongly-priced transactions	Spam a wrongly-priced transaction can cause a DoS	■	

Example:
Hacker short-sells tokens and drives down chain value

1. The cost ("weight") of some blockchain function was estimated too low
2. Hacker drives chain into loop by calling underpriced function repeatedly
3. The blockchain is frozen, loses credibitility and value

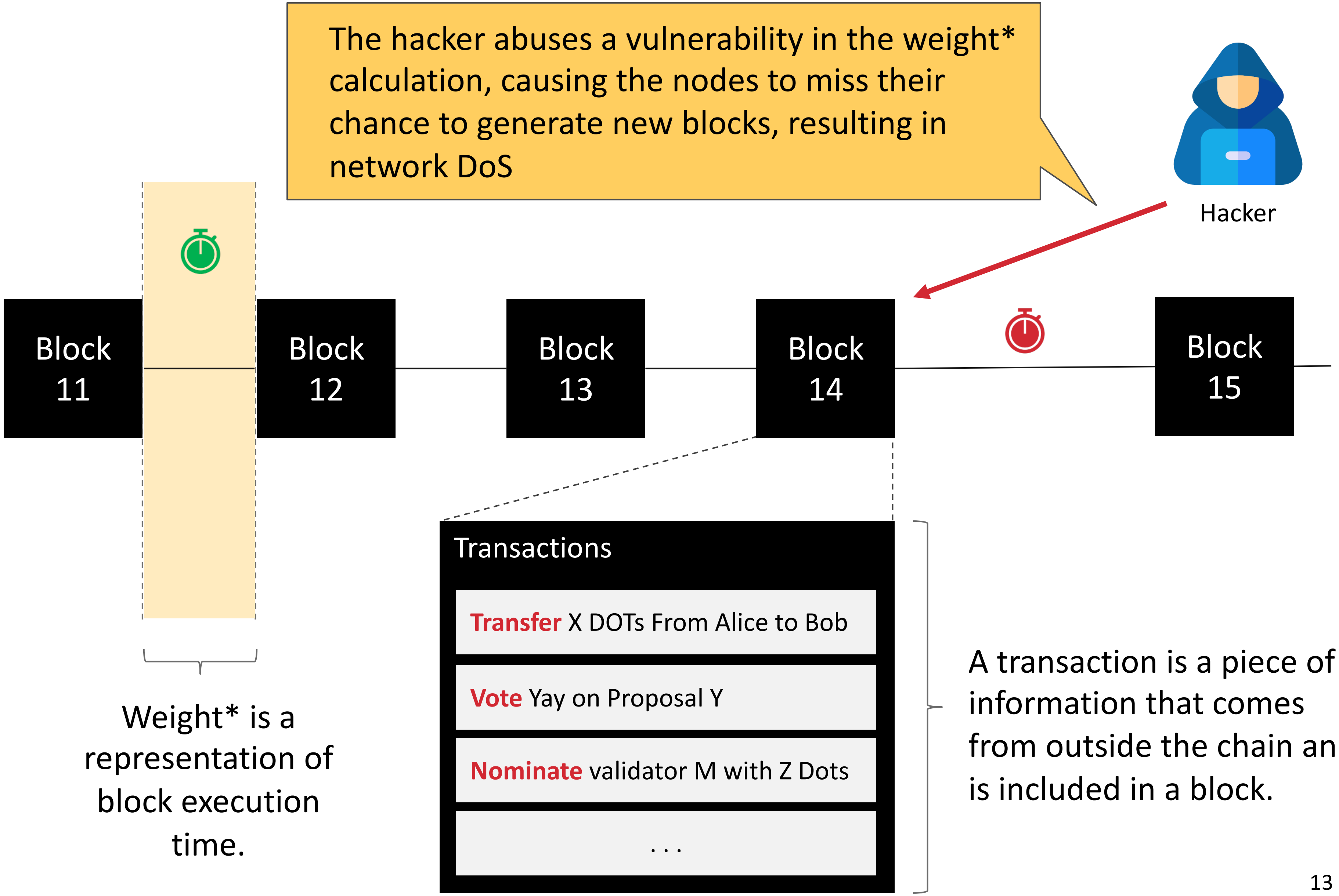
A 1 Miscalculation of block execution time can cause network DoS

Scenario 1: **Exhaust resources**

Background info.
Resources available to blockchains are limited. These resources include memory usage, storage I/O, computation, transaction/block size and state database size.

Attack.
The hacker sends a malicious transaction to cause block execution taking too long

*One unit of weight is one picosecond of execution time, that is 10**12 weight = 1 second, or 1,000 weight = 1 nanosecond, on fixed reference hardware



The hacker abuses a vulnerability in the weight* calculation, causing the nodes to miss their chance to generate new blocks, resulting in network DoS

Weight* is a representation of block execution time.

Transactions

- **Transfer** X DOTs From Alice to Bob
- **Vote** Yay on Proposal Y
- **Nominate** validator M with Z Dots
- . . .

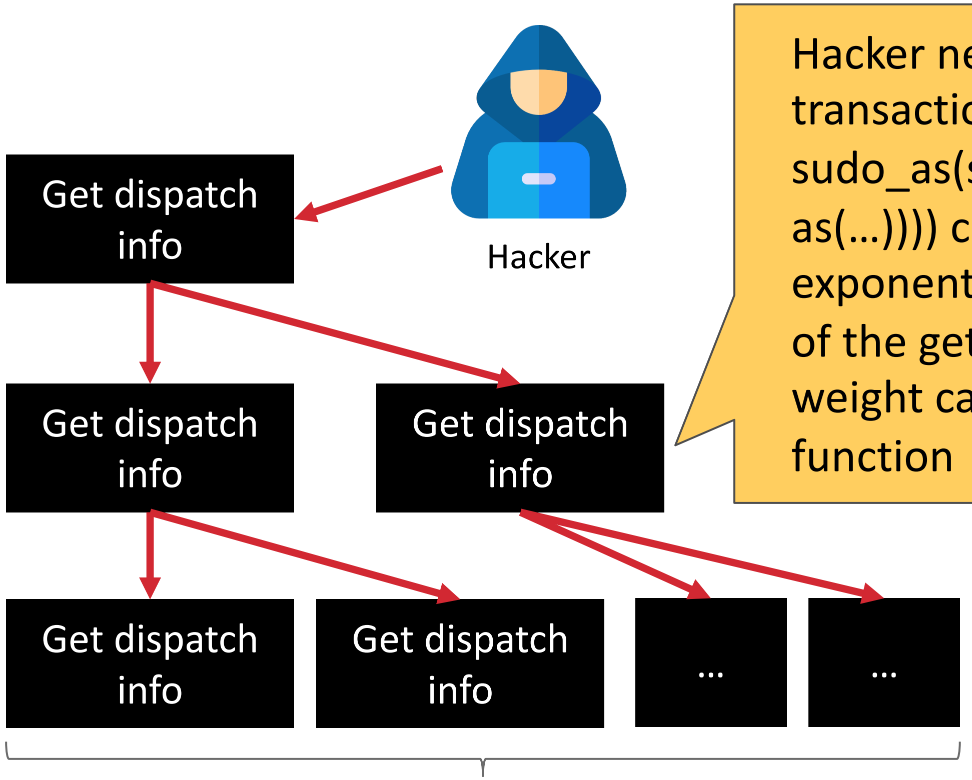
A transaction is a piece of information that comes from outside the chain and is included in a block.

Ⓐ ② Hackers can craft and gossip a nested transaction causing nodes to miss their slots and fail at block production, and potentially halting the blockchain

Scenario 1: **Vulnerable Code in sudo_as transaction**

```
#[weight = (
    call.get_dispatch_info().weight
    .saturating_add(10_000)
    // AccountData for inner call origin accountdata.
    .saturating_add(T::DbWeight::get().reads_writes(1, 1)),
    call.get_dispatch_info().class
)]
fn sudo_as(origin,
           who: <T::Lookup as StaticLookup>::Source,
```

To receive the sudo_as transaction call dispatch class and weight, the getter function to receive the dispatch information that holds both weight and class is called twice



Hacker nests transaction sudo_as(sudo_as(sudo_as(…)))) causing exponential complexity of the get dispatch info weight calculation function

Example: nesting sudo_as 41 times results in a call tree with 2^40 = **1’099’511’627’776 leaves**

B An arithmetic overflow in the weight calculation allows hacker to exhaust chain resources

Hacking goal	Bug type	Bug impact	Avail.	Integrity
Manipulate program flow	Unsafe arithmetic Logic bug	Abuse an operation to edit values to your advantage	■	■

Example: **Competitor sabotages credibility of the chain**

1. The cost ("weight") of some blockchain function is calculated based on the input's length; an arithmetic overflow will trigger a "wrap around" behavior
2. Competitor calls function with large input parameter, causing a huge discrepancy between the calculated weight and the computational work required
3. The blockchain acts contrary to its intended programming, loses credibility and value

B 1 Integer overflows can lead to financial loss or denial of service

Scenario 1: **Vulnerable Code**

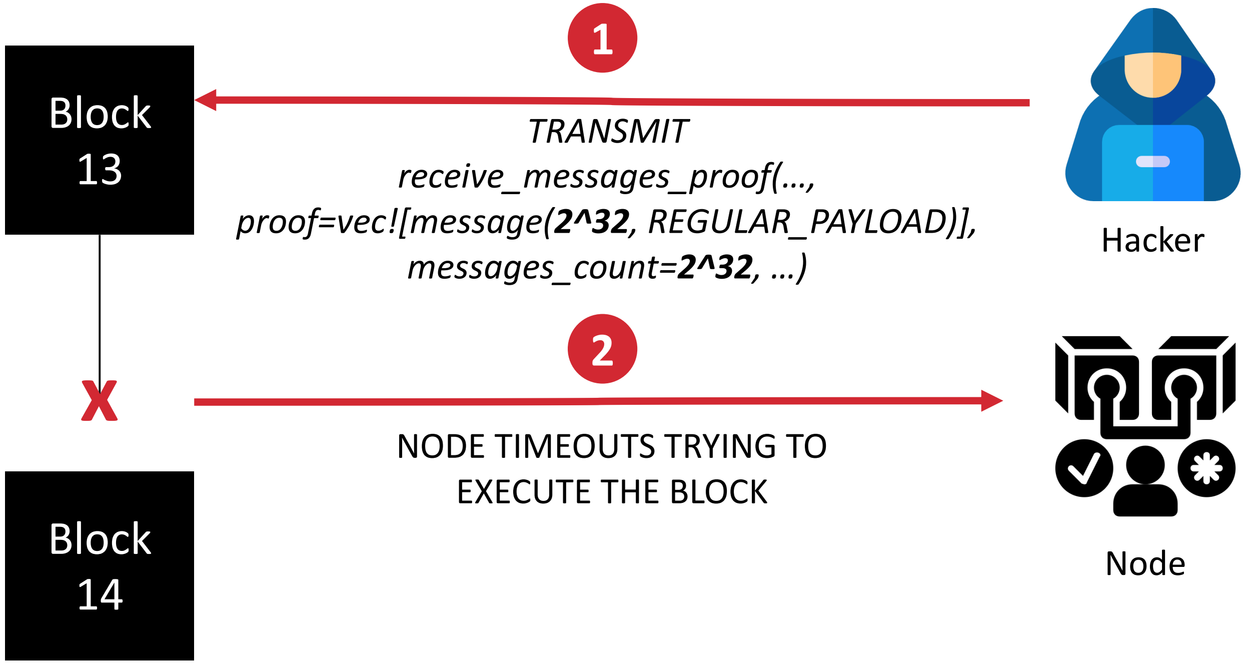
```
// verify that relayer is paying actual dispatch weight
let actual_dispatch_weight: Weight = messages
    .values()
    .map(|lane_messages| lane_messages
        .messages
        .iter()
        .map(T::MessageDispatch::dispatch_weight)
        .sum::<Weight>()
    )
   .sum();
if al  wed_dispatch_weight < actual_dispatch_weight {
    re
Err(Err       >::InvalidMessagesDispatchWeight.into());
}
```

The hacker sends a malicious transaction with a large proof in one of these queues, **causing a denial of service because of the “wrap around” behavior when sum() is executed.**

Attack sequence

① Hacker transmits a ***receive_messages_proof*** transaction containing a large ***proof*** struct and a high value for ***messages_count***

② Node that executes the block will timeout, missing their production slot



B 2 Bonus vulnerability: Arithmetic overflow prevention code leads to logic bug

Scenario 1: **Vulnerable Code**

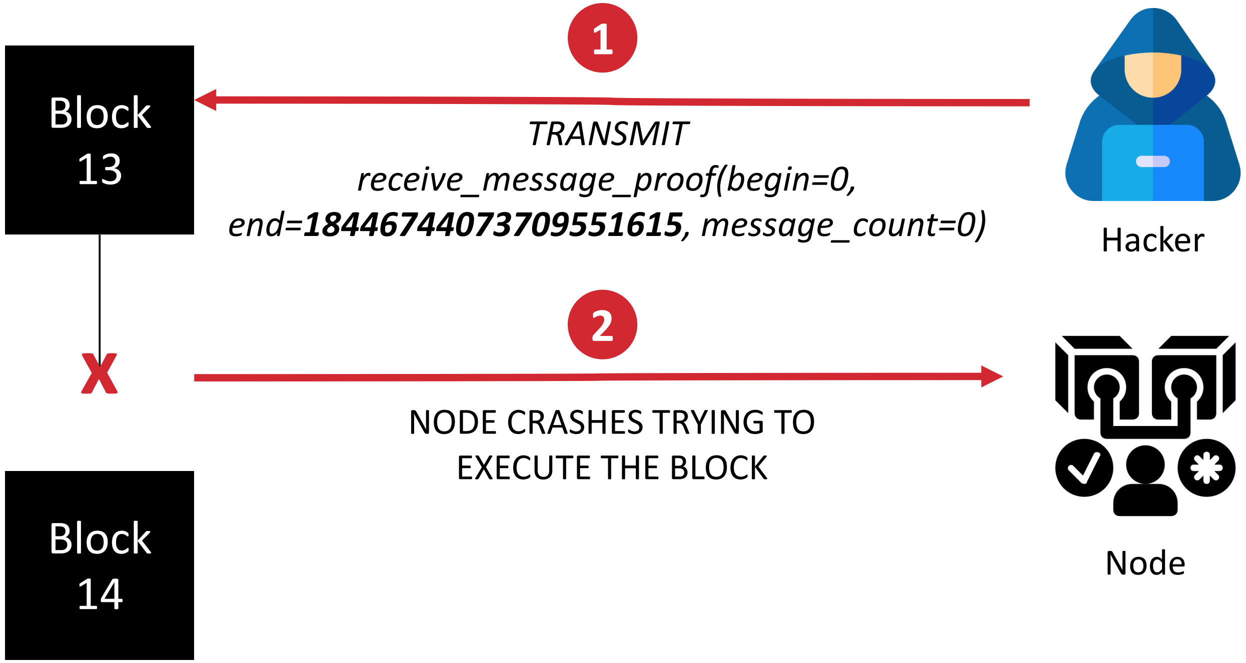
```
let messages_in_the_proof = end.checked_sub(begin)
    .and_then(|diff| diff.checked_add(1))
    .unwrap_or(0);
if messages_in_the_proof != messages_count {
    return Err(MessageProofError::MessagesCountMismatch);
}
let mut messages = Vec::with_capacity(end);
```

To circumvent overflows, developers will use "safe mathematic operations", such as **checked_add** or **saturating_add**. These add a new layer of complexity: the code must handle edge cases properly

Attack sequence

① Hacker transmits a ***receive_message_proof*** transaction containing an ***end*** value of u64::MAX and ***begin*** and ***message_count*** of 0, causing a vector to be allocated that has size u64::MAX

② Node that executes the block panics on trying to allocate the vector



*TRANSMIT receive_message_proof(begin=0, end=**18446744073709551615**, message_count=0)*

NODE CRASHES TRYING TO EXECUTE THE BLOCK

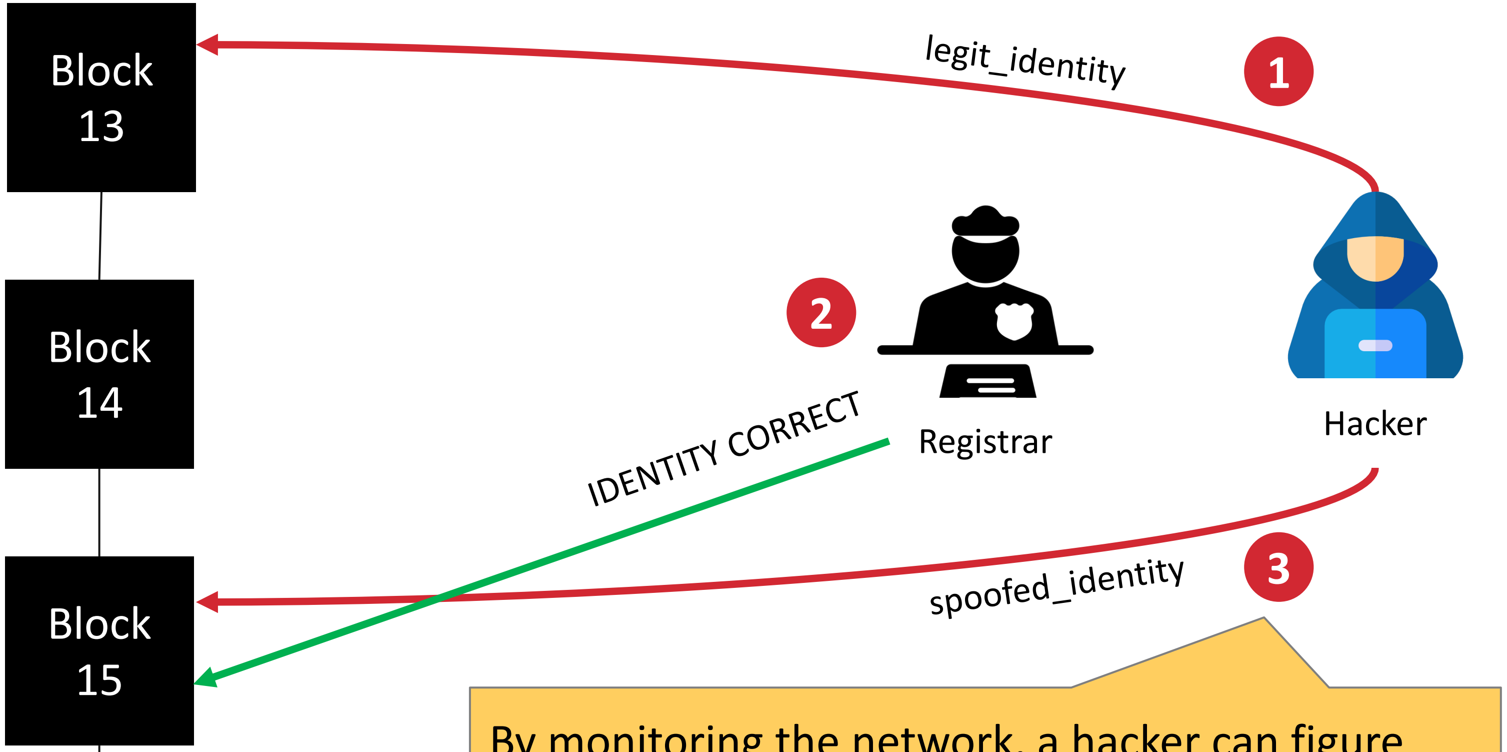
B Logic bugs can enable hackers to gain unfair advantages and rewards

Hacking goal	Bug type	Bug impact	Avail.	Integrity
Manipulate program flow	Unsafe arithmetic Logic bug	Abuse the logic of the chain against itself, gaining some reward in the process	■	■

Example:
Hacker exploits logic bug present in source code to spoof their identity

1. Hacker claims a legitimate identity on the blockchain and monitors the network to figure out when the registrar will hand out its sentence
2. Hacker requests a spoofed identity immediately between the time when the registrar submits their judgment and when it is accepted on-chain
3. The hacker now controls a validated spoofed identity because the code did not check which identity was being validated

B 3 Hackers can spoof their identity by re-setting it right before the judgement is given



By monitoring the network, a hacker can figure out when a registrar will provide judgement on their identity and rush to include a transaction modifying their requested identity, with a higher tip so it gets executed before the judgement.

Attack sequence

1. Hacker requests for legitimate identity to be judged “hello, I am legit_identity and here is my proof”
2. Registrar provides judgement on identity “hello User, your identity is correct”
3. Hacker requests for spoofed identity “hello, I am spoofed_identity” with high tip, thereby running ahead of transaction (2)

C Unhandled return values can cause the nodes to panic, allowing a hacker to DoS the chain

Hacking goal	Bug type	Bug impact	Avail.	Integrity
DoS nodes	Reachable panics	Cause every available node to panic and cause a DoS	■	□

Example:
Hacker launches DoS attack against chain for supply-chain attack

1. Non-explicit handling of a function’s result assumes that it cannot return a “None” value
2. Hacker causes all the nodes to panic by calling a transaction with a high value as parameter
3. The blockchain is frozen, functionality is halted for projects using the chain, and the ecosystem loses credibitility and value

C Triggering Rust panic conditions can compromise chain availability

Scenario 1: **Vulnerable code**

```
Call::Bids(Call::create_bid {
    BidDetails {
        currency: CENTS,
        dot_amount: 5734123568823053662,
        quantity_of_data_in_bytes: 31768
    }
})
```

create_bid INVOKES *get_dot_to_token*, PASSING *dot_amount*

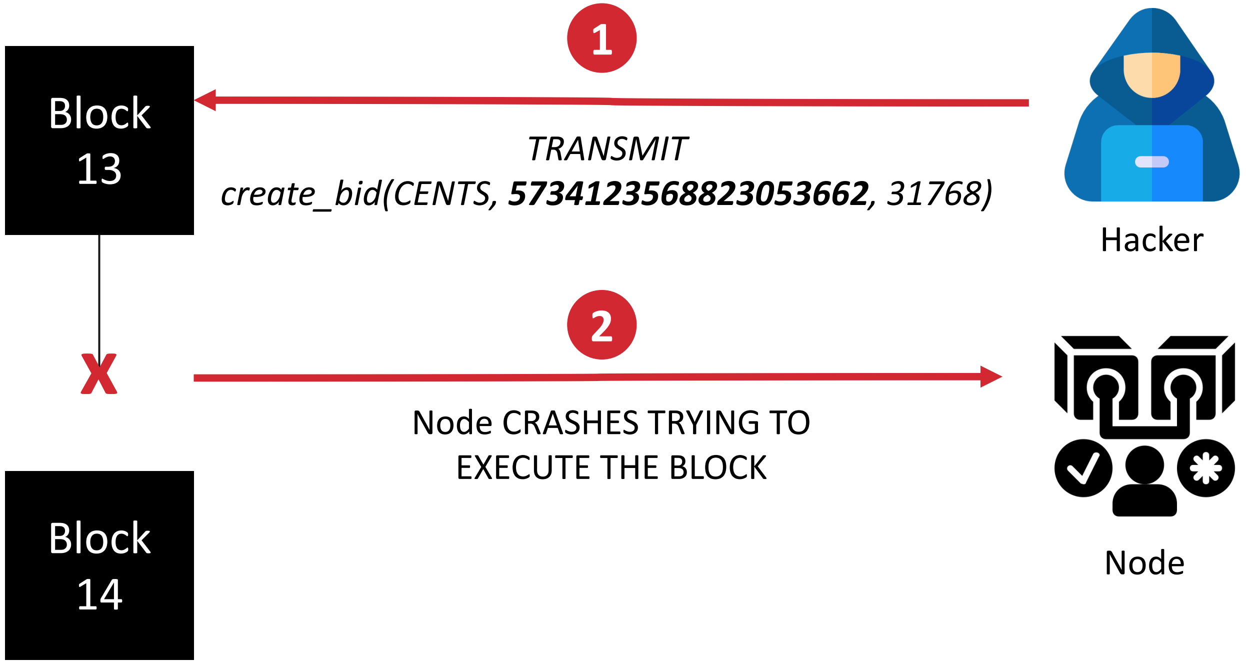
```
fn get_dot_to_token (dot_amount: u128) -> u128 {
    dot_amount.checked_mul(T::DOTToTokenRate::get()).unwrap();
}
```

Rust chooses to panic when *None* is returned in order to avoid any unexpected behavior.
In such cases, the runtime assumes that it is **better to stop the program instead of using an unexpected value.**

Attack sequence

① Hacker transmits a ***create_bid*** transaction containing an unusually high ***dot_amount*** (close to u128::MAX)

② Node that executes the block crashes



D Wrong configuration in runtime allows hacker to fill up blockchain’s storage

Hacking goal	Bug type	Bug impact	Avail.	Integrity
Abuse business logic	Incorrect usage of standard patterns	Abuse a misconfiguration allows a hacker to gain a financial advantage or cause DoS	■	■

Example:
Hacktivist drives down credibility of chain

1. Wrongly configured runtime parameter allows the creation of an account for a derisory amount of money
2. Hacktivist creates a large number of accounts for a small amount of money, cluttering the storage of the chain
3. The blockchain’s storage size increases, causing longer transaction times; it loses credibitility and value

D A bad runtime configuration can open vulnerabilities in the blockchain

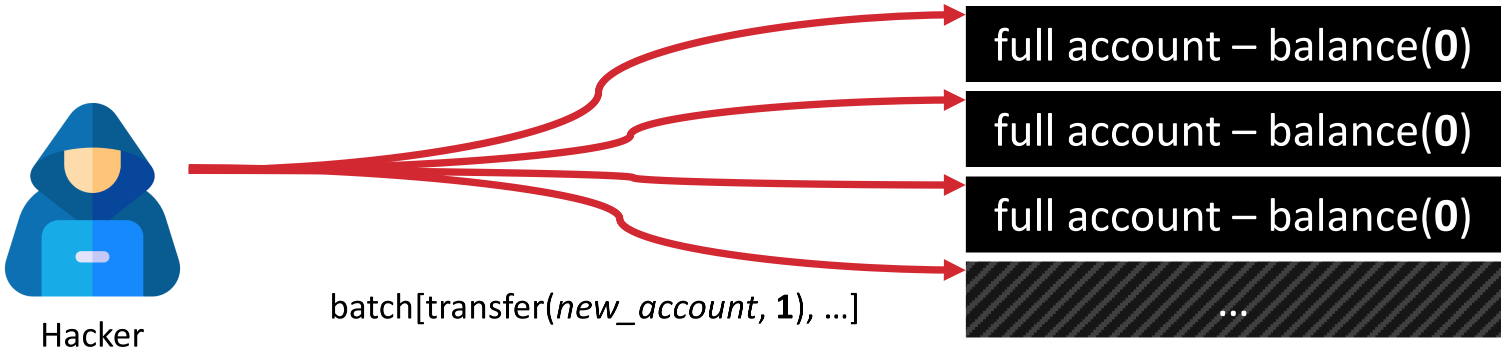
Scenario 1: **Vulnerable Code**

```
pub const UNITS: Balance = 1_000_000_000_000;
pub const CENTS: Balance = UNITS / 100;

parameter_types! {
    pub const ExistentialDeposit: Balance = 0;
    pub const MaxLocks: u32 = 50;
    pub const MaxReserves: u32 = 50;
}
```

Each account is represented by an **Account** structure which keeps track of user's balance and Substrate specific reference counters, but it can also be enhanced with project specific parameters. This data structure lives in the storage of the blockchain if the account has a balance of at least **ExistentialDeposit**.

Attack

If **ExistentialDeposit** is set to a small value, a hacker could create a lot of accounts which will fill up the storage of the blockchain, using only a small amount of money for transaction costs.



Hacker

batch[transfer(*new_account*, **1**), …]

full account – balance(**0**)

full account – balance(**0**)

full account – balance(**0**)

…

Setting an existential deposit of **1** means setting an existential deposit of 0.000000000001 **UNITS**, which is not enough to prevent spamming the creating of new accounts.

project	**ED**	**in €**
polkadot	1 DOT	5 €
kusama	0.003 KSM	0.01 €

E Non existing storage deposits allows hacker to fill up blockchain’s storage

Hacking goal	Bug type	Bug impact	Avail.	Integrity
Slow down chain	Storage bloating	Reduce chain useability by filling its storage	■	□

Example: **Disgruntled insider drives down credibility of chain**

1. Missing storage deposit for the creation of large database items allows spamming this process
2. Disgruntled insider creates many storage items for a small amount of money, cluttering chain storage
3. Chain storage increases, causing longer transaction times and harder operability; it loses credibitility and value

E Insufficient storage deposits can allow a hacker to cheaply fill the blockchain storage

Storage bloating refers to the phenomenon of excessive accumulation of data within a blockchain network, **leading to increased storage requirements and potential operational inefficiencies**

Scenario 1: **Vulnerable Code**

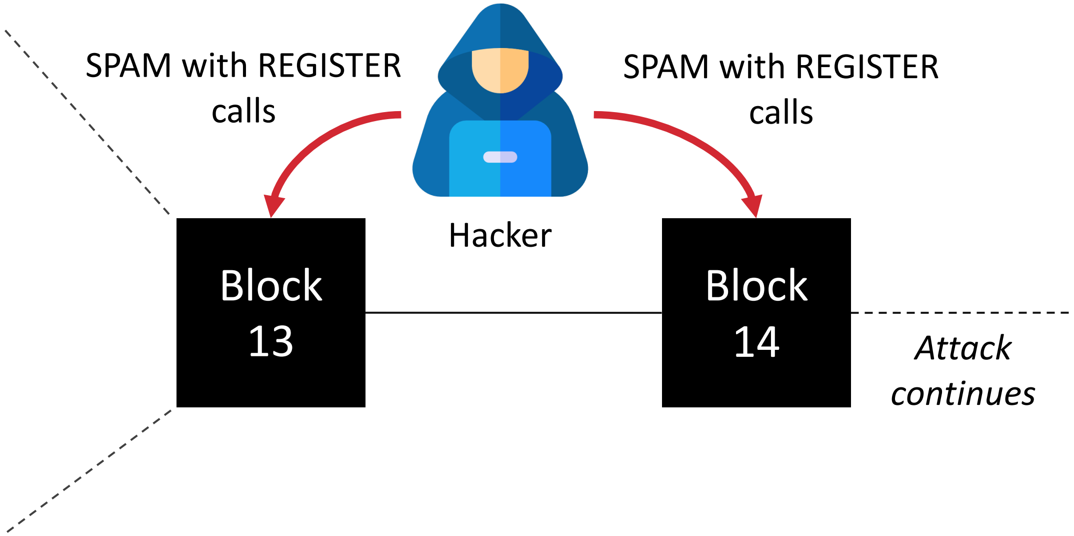
```
pub storage RequiredStakeForStakeAndSlash: Balance = 1_000_000;
...
pub fn register(origin: OriginFor<T>, valid_till: T::BlockNumber) -> DispatchResult {
    let relayer = ensure_signed(origin)?;
    ...
    RegisteredRelayers::<T>::try_mutate(&relayer, |maybe_registration| -> DispatchResult {
        let mut registration = maybe_registration
            .unwrap_or_else(|| Registration { valid_till, stake: Zero::zero() });
    ...
```

Attack

Spamming millions of ***Bridges::register()*** calls could result in **1GB of storage** filled for only ~USD 25’000 (compared to tens of millions of $ in other blockchains).

Transactions

- **Register** bridge relayer
- **Register** bridge relayer
- **Register** bridge relayer
- . . .



Agenda

- Intro to third-generation blockchains
- Five types of blockchain hacking
- **Fuzzing blockchains effectively**

Three analysis techniques to find blockchain bugs

	I Static analysis	II Fuzzing	III Manual review
A Wrongly-priced transactions	✓	✓	✓
B Unsafe arithmetic		✓	✓
C Reachable panics	✓	✓	✓
D Incorrect usage of standard patterns	✓		✓
E Storage bloating		Working on it	✓

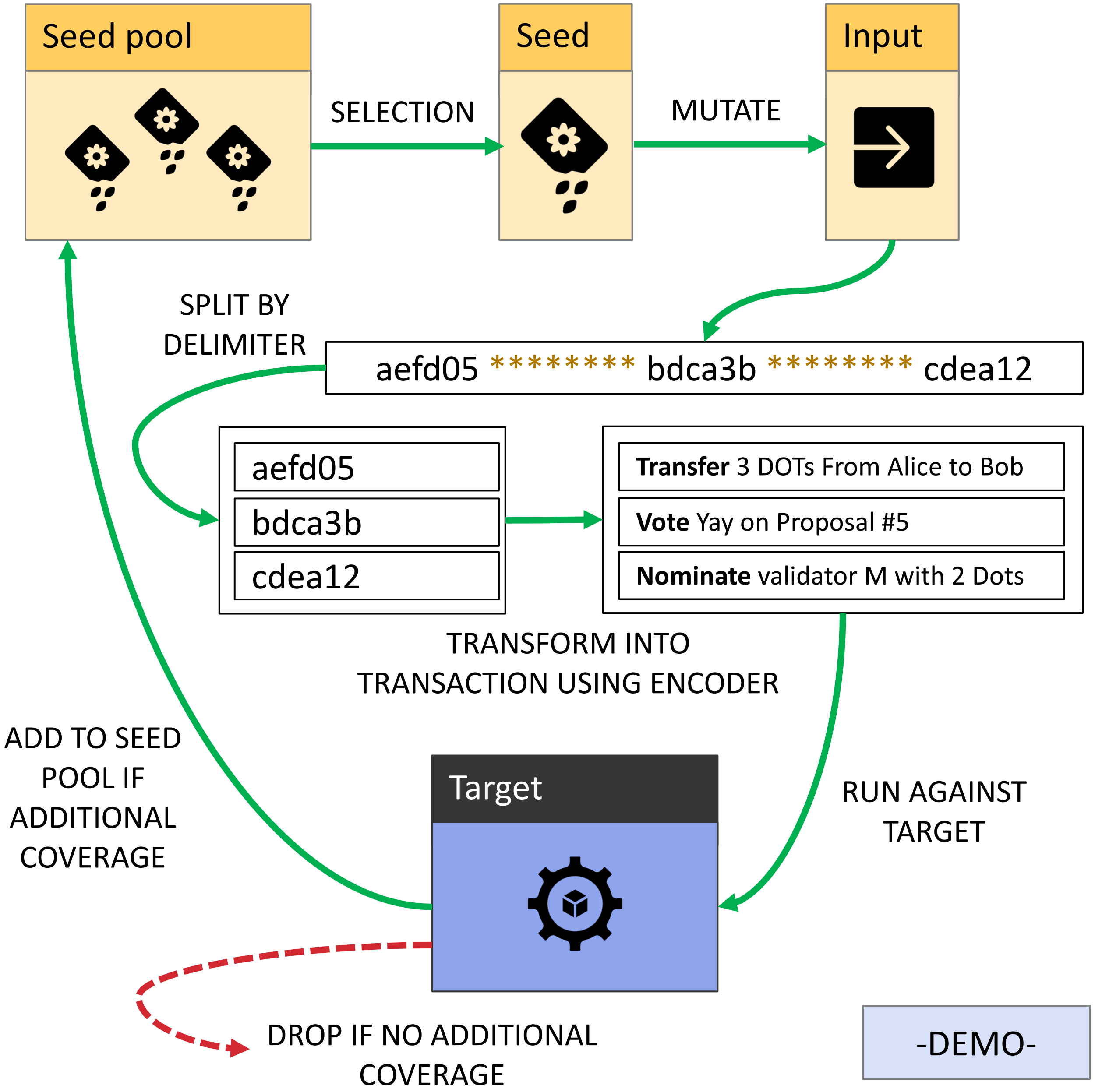
Take aways

- Static analysis should be done as part of development process, using tools such as *semgrep* and *dylint*
- Security testing then typically starts with fuzz testing, which is particularly strong in finding availability bugs
- Before an economic launch, every project should also go through security auditing including manual review

We created software to find bugs in all these categories, this is our fuzz engine

Custom parser-based fuzzing: Transform binary stream of data into list of runtime calls and run against target program

AFL++ and honggfuzz based coverage guided fuzz engine	①	Mutate binary input and transform into transactions (=API calls)
Powerful mutator	②	Fuzzer applies random permutation to given binary string
Substrate-based target	③	Run extracted transactions/API calls against the target program on a predefined state



Seed pool

SELECTION

Seed

MUTATE

Input

SPLIT BY DELIMITER

aefd05 ******** bdca3b ******** cdea12

aefd05
bdca3b
cdea12

Transfer 3 DOTs From Alice to Bob
Vote Yay on Proposal #5
Nominate validator M with 2 Dots

TRANSFORM INTO TRANSACTION USING ENCODER

RUN AGAINST TARGET

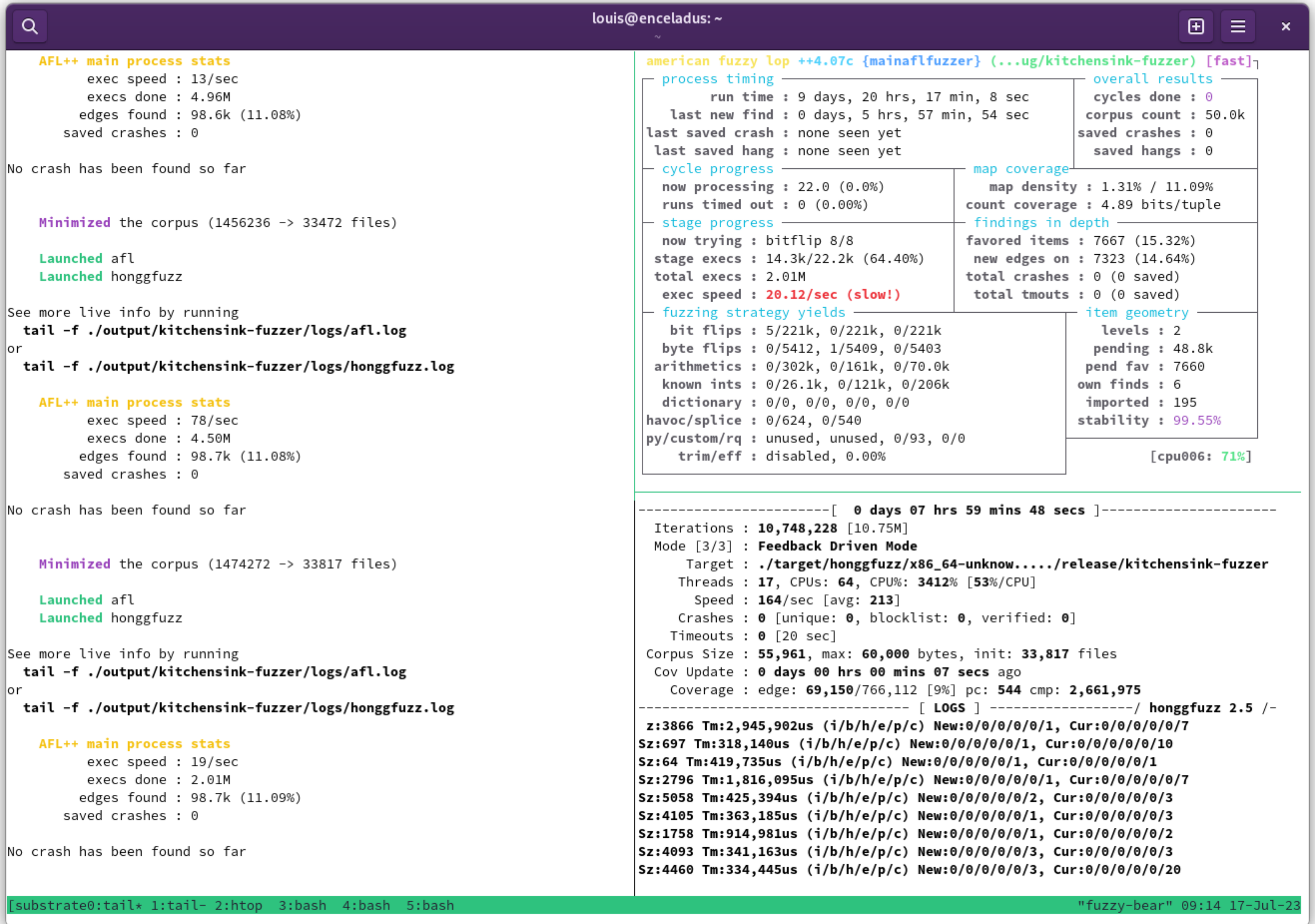
Target

ADD TO SEED POOL IF ADDITIONAL COVERAGE

DROP IF NO ADDITIONAL COVERAGE

-DEMO-

- Demo -

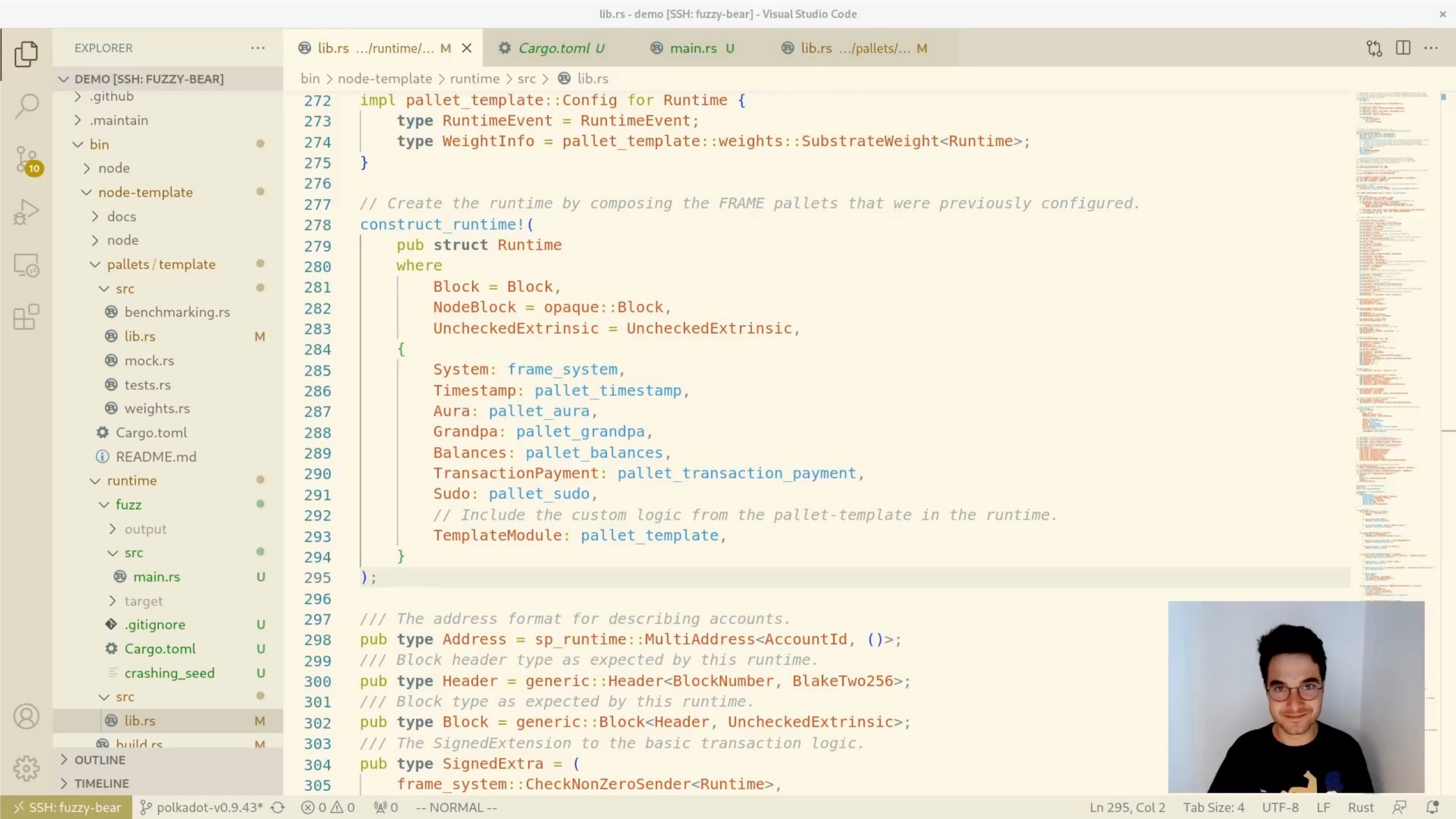


```
    AFL++ main process stats
          exec speed : 13/sec
          execs done : 4.96M
         edges found : 98.6k (11.08%)
      saved crashes : 0

No crash has been found so far


    Minimized the corpus (1456236 -> 33472 files)

    Launched afl
    Launched honggfuzz

See more live info by running
  tail -f ./output/kitchensink-fuzzer/logs/afl.log
or
  tail -f ./output/kitchensink-fuzzer/logs/honggfuzz.log

    AFL++ main process stats
          exec speed : 78/sec
          execs done : 4.50M
         edges found : 98.7k (11.08%)
      saved crashes : 0

No crash has been found so far


    Minimized the corpus (1474272 -> 33817 files)

    Launched afl
    Launched honggfuzz

See more live info by running
  tail -f ./output/kitchensink-fuzzer/logs/afl.log
or
  tail -f ./output/kitchensink-fuzzer/logs/honggfuzz.log

    AFL++ main process stats
          exec speed : 19/sec
          execs done : 2.01M
         edges found : 98.7k (11.09%)
      saved crashes : 0

No crash has been found so far
```

```
american fuzzy lop ++4.07c {mainaflfuzzer} (...ug/kitchensink-fuzzer) [fast]┐
┌─ process timing ───────────────────────────────────┬─ overall results ──┐
│        run time : 9 days, 20 hrs, 17 min, 8 sec    │  cycles done : 0   │
│   last new find : 0 days, 5 hrs, 57 min, 54 sec    │ corpus count : 50.0k│
│last saved crash : none seen yet                    │saved crashes : 0   │
│ last saved hang : none seen yet                    │  saved hangs : 0   │
├─ cycle progress ─────────────────┬─ map coverage┴────────────────────┤
│  now processing : 22.0 (0.0%)    │    map density : 1.31% / 11.09%    │
│  runs timed out : 0 (0.00%)      │ count coverage : 4.89 bits/tuple   │
├─ stage progress ─────────────────┼─ findings in depth ──────────────┤
│  now trying : bitflip 8/8        │ favored items : 7667 (15.32%)      │
│ stage execs : 14.3k/22.2k (64.40%)│  new edges on : 7323 (14.64%)     │
│ total execs : 2.01M              │ total crashes : 0 (0 saved)        │
│  exec speed : 20.12/sec (slow!)  │  total tmouts : 0 (0 saved)        │
├─ fuzzing strategy yields ────────┴─────────────┬─ item geometry ───────┤
│   bit flips : 5/221k, 0/221k, 0/221k           │    levels : 2         │
│  byte flips : 0/5412, 1/5409, 0/5403           │   pending : 48.8k     │
│ arithmetics : 0/302k, 0/161k, 0/70.0k          │  pend fav : 7660      │
│  known ints : 0/26.1k, 0/121k, 0/206k          │ own finds : 6         │
│  dictionary : 0/0, 0/0, 0/0, 0/0               │  imported : 195       │
│havoc/splice : 0/624, 0/540                     │ stability : 99.55%    │
│py/custom/rq : unused, unused, 0/93, 0/0        ├───────────────────────┘
│    trim/eff : disabled, 0.00%                  │          [cpu006: 71%]
└────────────────────────────────────────────────┘
```

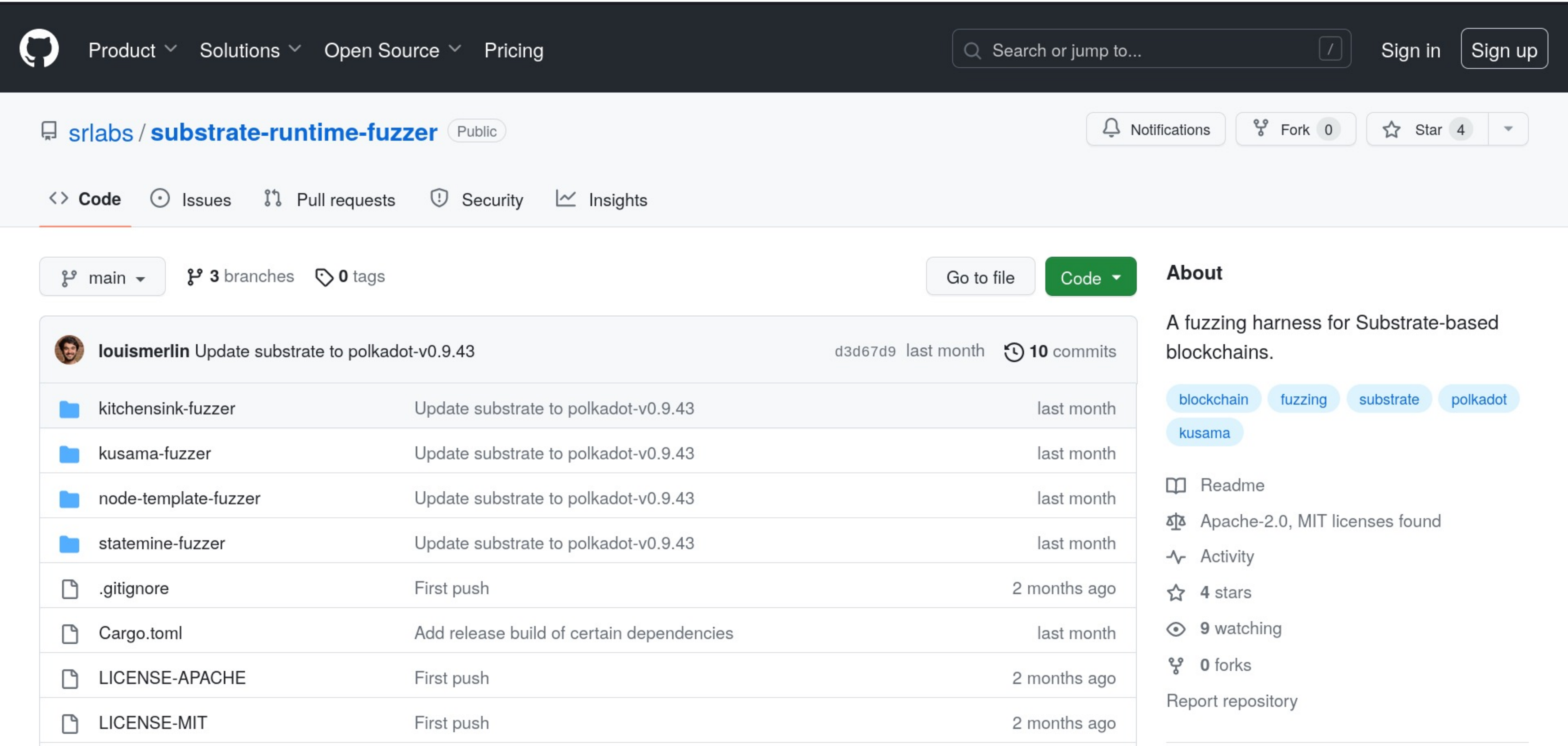
```
------------------------[  0 days 07 hrs 59 mins 48 secs ]----------------------
  Iterations : 10,748,228 [10.75M]
  Mode [3/3] : Feedback Driven Mode
      Target : ./target/honggfuzz/x86_64-unknow...../release/kitchensink-fuzzer
     Threads : 17, CPUs: 64, CPU%: 3412% [53%/CPU]
       Speed : 164/sec [avg: 213]
     Crashes : 0 [unique: 0, blocklist: 0, verified: 0]
    Timeouts : 0 [20 sec]
 Corpus Size : 55,961, max: 60,000 bytes, init: 33,817 files
  Cov Update : 0 days 00 hrs 00 mins 07 secs ago
    Coverage : edge: 69,150/766,112 [9%] pc: 544 cmp: 2,661,975
---------------------------------- [ LOGS ] ------------------/ honggfuzz 2.5 /-
 z:3866 Tm:2,945,902us (i/b/h/e/p/c) New:0/0/0/0/0/1, Cur:0/0/0/0/0/7
Sz:697 Tm:318,140us (i/b/h/e/p/c) New:0/0/0/0/0/1, Cur:0/0/0/0/0/10
Sz:64 Tm:419,735us (i/b/h/e/p/c) New:0/0/0/0/0/1, Cur:0/0/0/0/0/1
Sz:2796 Tm:1,816,095us (i/b/h/e/p/c) New:0/0/0/0/0/1, Cur:0/0/0/0/0/7
Sz:5058 Tm:425,394us (i/b/h/e/p/c) New:0/0/0/0/0/2, Cur:0/0/0/0/0/3
Sz:4105 Tm:363,185us (i/b/h/e/p/c) New:0/0/0/0/0/1, Cur:0/0/0/0/0/3
Sz:1758 Tm:914,981us (i/b/h/e/p/c) New:0/0/0/0/0/1, Cur:0/0/0/0/0/2
Sz:4093 Tm:341,163us (i/b/h/e/p/c) New:0/0/0/0/0/3, Cur:0/0/0/0/0/3
Sz:4460 Tm:334,445us (i/b/h/e/p/c) New:0/0/0/0/0/3, Cur:0/0/0/0/0/20
```

```
[substrate0:tail* 1:tail- 2:htop  3:bash  4:bash  5:bash                "fuzzy-bear" 09:14 17-Jul-23
```



```rust
impl pallet_template::Config for Runtime {
    type RuntimeEvent = RuntimeEvent;
    type WeightInfo = pallet_template::weights::SubstrateWeight<Runtime>;
}

// Create the runtime by composing the FRAME pallets that were previously configured.
construct_runtime!(
    pub struct Runtime
    where
        Block = Block,
        NodeBlock = opaque::Block,
        UncheckedExtrinsic = UncheckedExtrinsic,
    {
        System: frame_system,
        Timestamp: pallet_timestamp,
        Aura: pallet_aura,
        Grandpa: pallet_grandpa,
        Balances: pallet_balances,
        TransactionPayment: pallet_transaction_payment,
        Sudo: pallet_sudo,
        // Include the custom logic from the pallet-template in the runtime.
        TemplateModule: pallet_template,
    }
);

/// The address format for describing accounts.
pub type Address = sp_runtime::MultiAddress<AccountId, ()>;
/// Block header type as expected by this runtime.
pub type Header = generic::Header<BlockNumber, BlakeTwo256>;
/// Block type as expected by this runtime.
pub type Block = generic::Block<Header, UncheckedExtrinsic>;
/// The SignedExtension to the basic transaction logic.
pub type SignedExtra = (
    frame_system::CheckNonZeroSender<Runtime>,
```

https://github.com/srlabs/substrate-runtime-fuzzer



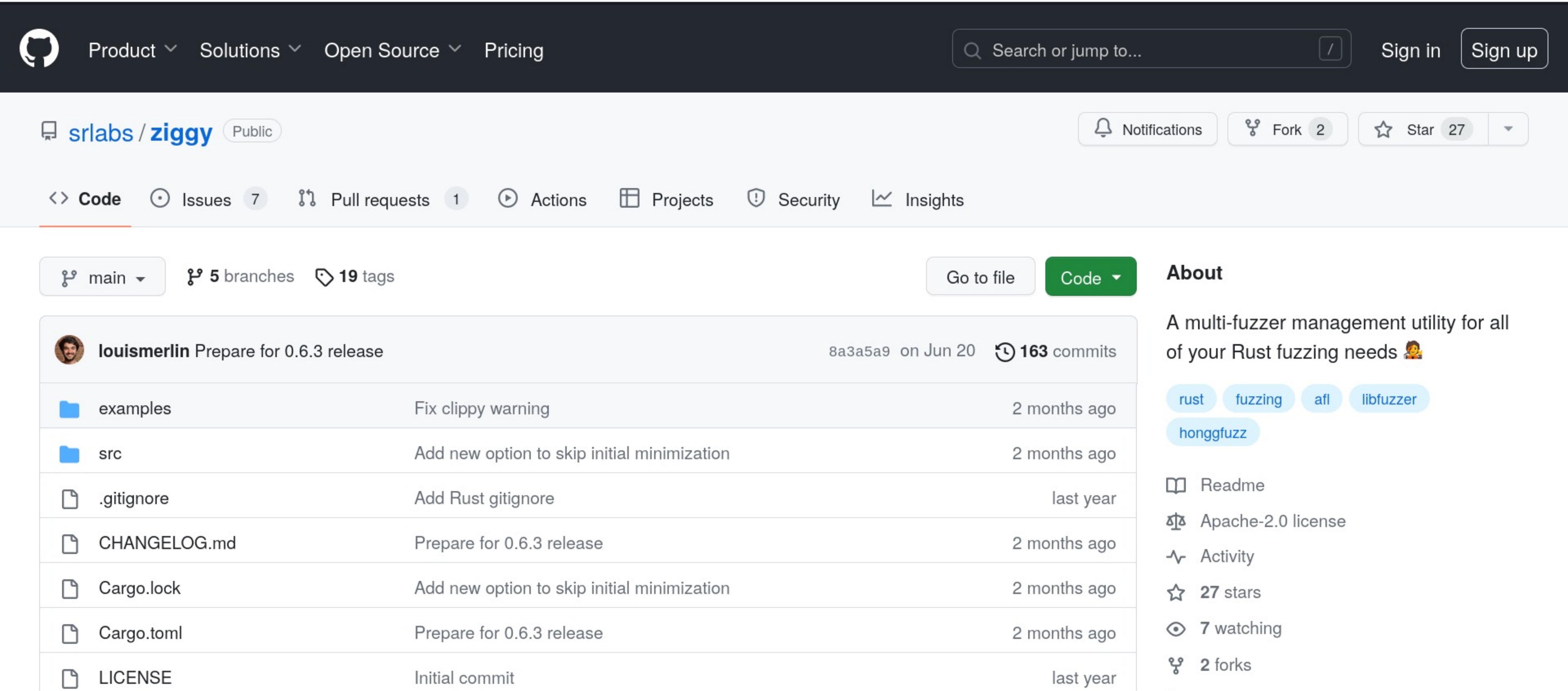
louismerlin Update substrate to polkadot-v0.9.43 d3d67d9 last month 10 commits

kitchensink-fuzzer	Update substrate to polkadot-v0.9.43	last month
kusama-fuzzer	Update substrate to polkadot-v0.9.43	last month
node-template-fuzzer	Update substrate to polkadot-v0.9.43	last month
statemine-fuzzer	Update substrate to polkadot-v0.9.43	last month
.gitignore	First push	2 months ago
Cargo.toml	Add release build of certain dependencies	last month
LICENSE-APACHE	First push	2 months ago
LICENSE-MIT	First push	2 months ago

About

A fuzzing harness for Substrate-based blockchains.

https://github.com/srlabs/ziggy



louismerlin Prepare for 0.6.3 release 8a3a5a9 on Jun 20 163 commits

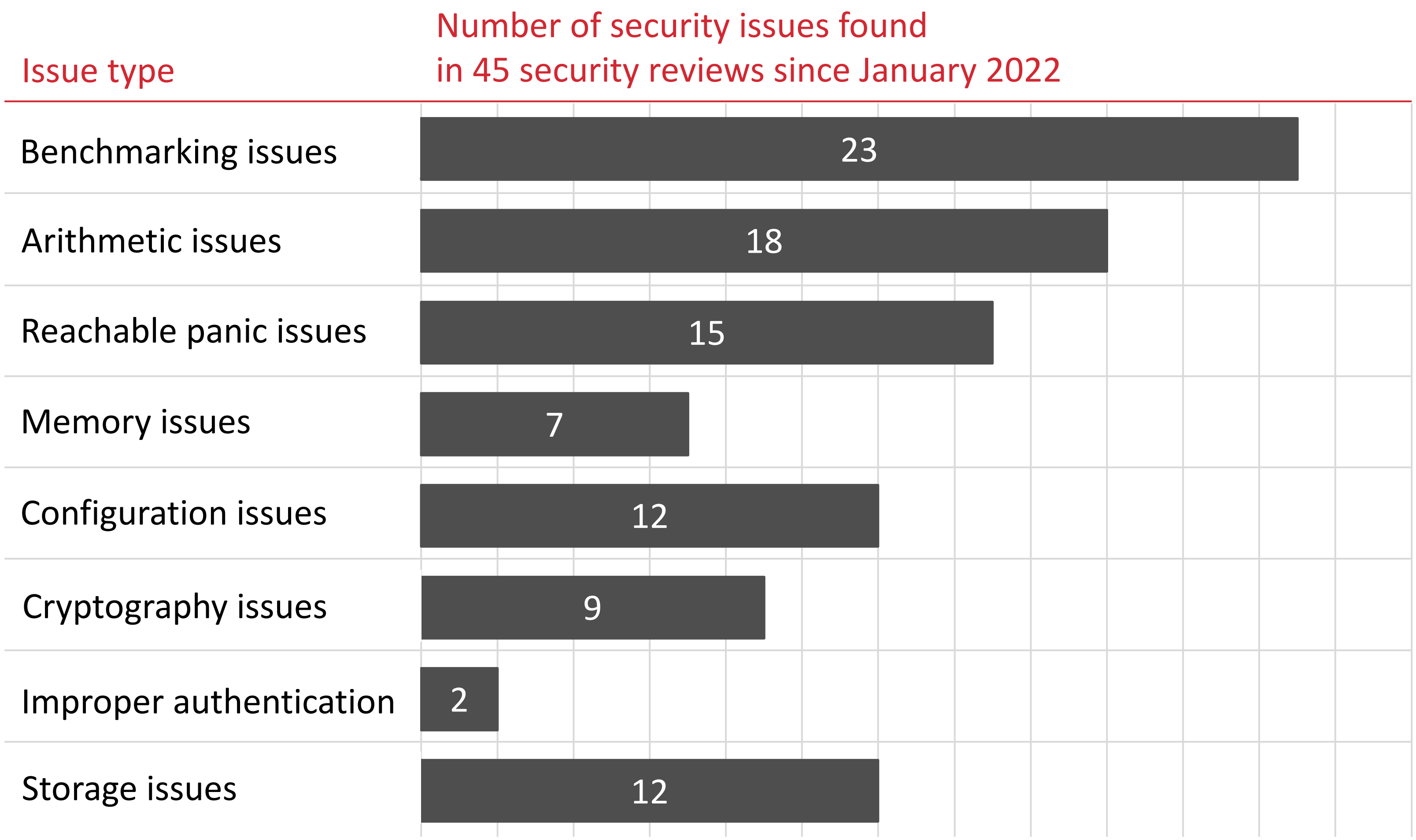
examples	Fix clippy warning	2 months ago
src	Add new option to skip initial minimization	2 months ago
.gitignore	Add Rust gitignore	last year
CHANGELOG.md	Prepare for 0.6.3 release	2 months ago
Cargo.lock	Add new option to skip initial minimization	2 months ago
Cargo.toml	Prepare for 0.6.3 release	2 months ago
LICENSE	Initial commit	last year

About

A multi-fuzzer management utility for all of your Rust fuzzing needs 🧑‍🎤

We continuously find bugs on a variety of chains

- **A** Wrongly-priced transactions
- **B** Unsafe arithmetic
- **C** Reachable panics
- **D** Incorrect usage of standard patterns
- **E** Storage bloating



Semi-automated testing is most effective in detecting
insufficient benchmarking, **unsafe arithmetic usage**, **reachable panics** and **configuration issues**

Takeaways

1	**Blockchains contain fascinating hacking puzzles**
2	**Most bugs fall into five categories, many are crashes**
3	**Open-source tools enable mostly-automated reviews**

Questions?

Karsten Nohl <nohl@srlabs.de>
Louis Merlin <louis@srlabs.de>
Gabriel Arnautu <gabriel@srlabs.de>