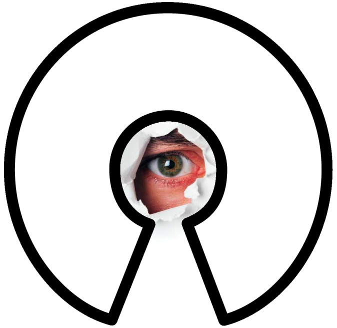
VeriOSS: using the Blockchain to Foster Bug Bounty Programs

Gabriele Costa, Andrea Canidio, Letterio Galletta
IMT School for Advanced Studies, Lucca
name.surname@imtlucca.it

Tokenomics 2020

The problem

- **Open source software** (OSS) is ubiquitous
  - Web browsers
  - Operating Systems
  - Libraries
- Many security-critical utilities
  - OpenSSL, KeePass, GnuPG, ...
- Vulnerabilities in OSS may spread out to many systems
  - Also closed source and proprietary software
- Security analysts can inspect the code
  - But vulnerability detection is hard and requires workforce and money



Bug bounty

- **Bounty issuers** offer rewards for each bug found
  - Rewards depend on type and impact
- **Bounty hunters** apply by submitting a bug report
  - I.e., they disclose a vulnerability to the bounty issuer
- **Hunters can federate** in large groups of ethical hackers
  - E.g., **HackerOne**
- For OSS, bug bounties can be offered by third parties
  - E.g., public institutions (see **EU-FOSSA** and **EU-FOSSA 2**)

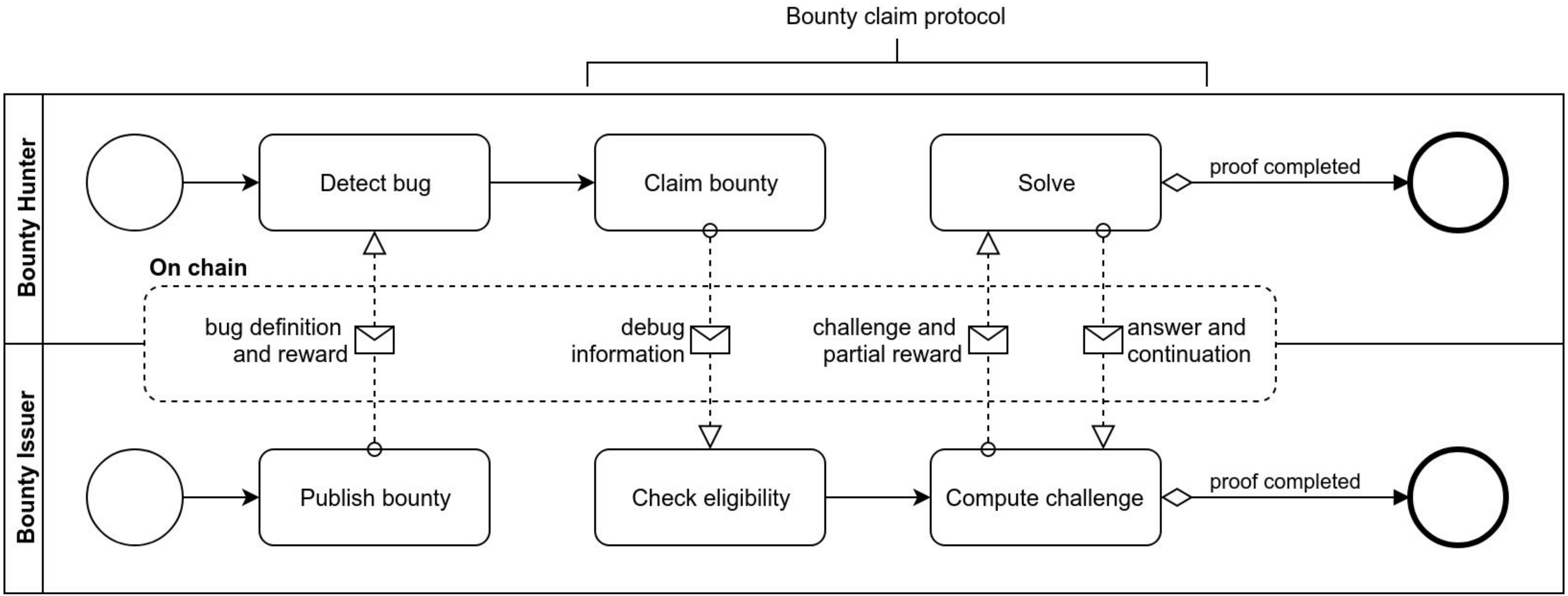
Market performance

- Two markets: **bug bounties** vs. **0-day exploits**
  - 0-day exploits can be sold on gray/black markets
  - E.g., zerodium
- If a bug can be turned into an exploit, the value increases significantly
  - **Google Chrome: from 5000$ to 300000$**
  - source: https://tinyurl.com/vlroo69
- Many bounty issuers also require an **evidence** of the bug
  - Typically an exploit
- Bug reporting may even require extra effort
  - E.g., providing a remediation plan
- Bug **eligibility** is decided by the bounty issuer **after** disclosure
  - **Limited bargaining power for the bounty hunter**

Fair information trading

- Hörner and Skrzypacz (2016)
  - Information sellers sustain several tests aiming at confirming their knowledge
  - Every time a test is passed, the buyer sends the seller a partial payment
  - Main problem: total lack of (mutual) commitment
- Avoid the hold up problem with smart contracts
  - BH commits bug info without revealing it
  - Smart contracts automate test execution and partial payments
  - Initial disclosure follows the same logic of Hörner and Skrzypacz (2016)

VeriOSS

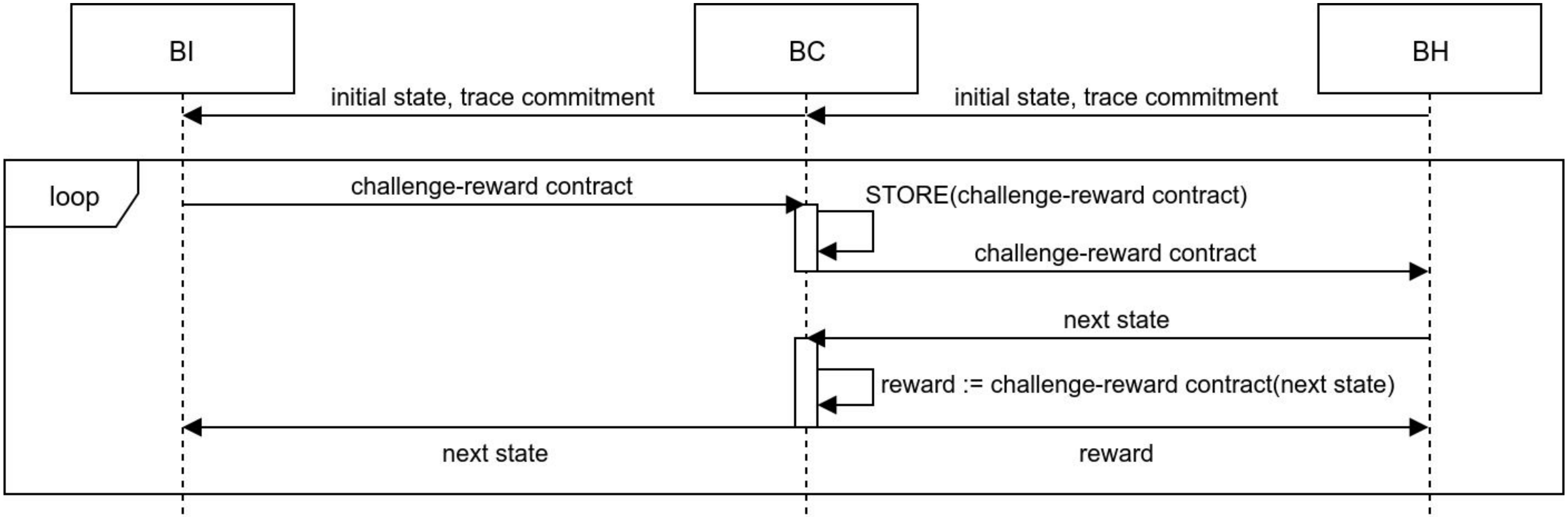


Bounty claim protocol
Bounty Hunter
Bounty Issuer
Detect bug
Claim bounty
Solve
proof completed
On chain
bug definition and reward
debug information
challenge and partial reward
answer and continuation
Publish bounty
Check eligibility
Compute challenge
proof completed

Bounty claim protocol

- On chain (BC), fair exchange protocol between issuer (BI) and hunter (BH)
  - Pay-per-Knowledge (P2K)
- BH starts by sharing **trace commitment** and **initial (failure) state**
  - The execution trace cannot be changed later on
- BI checks the **bug eligibility** on the failure state
  - Can decide if bug is relevant, but cannot rebuild the entire trace
- BI and BH can **negotiate protocol parameters**
  - Total reward and partial rewards computation method

Bounty claim protocol: MSC



Program trace & commitment

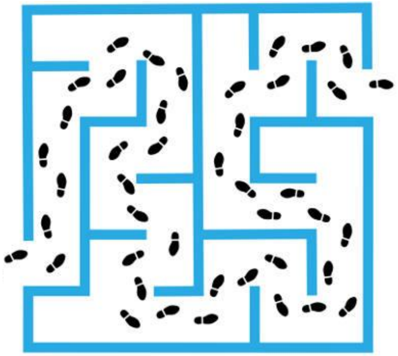
- A program trace is a **sequence of states** computed at runtime
- Roughly, each state is a **mapping from variables to values**
  - E.g., { x ↦ 42, y ↦ true, z ↦ ‘A’}
- A trace **uniquely identifies an execution** and **can be repeated**
  - E.g., it starts with the program input
- **Faulty states** can also appear in the trace
  - E.g., “segmentation fault”
- Commitment amounts to computing and **sharing cryptographic hash of states**
  - Decommitment occurs when states are revealed

Program trace & commitment

- A program trace is a **sequence of states** computed at runtime
- Roughly, each state is a **mapping from variables to values**
  - E.g., { x ↦ 42, y ↦ true, z ↦ ‘A’}
- A trace **uniquely identifies an execution** and **can be repeated**
  - E.g., it starts with the program input
- **Faulty states** can also appear in the trace
  - E.g., “segmentation fault”
- Commitment amounts to computing and **sharing cryptographic hash of states**
  - Decommitment occurs when states are revealed

Challenge-response

- The loop in the bounty claim protocol is a **proof-of-knowledge** (PoK)
  - BH proves she knows the next segment of the execution trace
- BI publishes a smart contract with a **challenge**
  - The contract pays its balance to BH only if she can solve the challenge
- The challenge is a **NP-hard** problem if BH does not know the trace
  - E.g., providing a model for a satisfiable SMT formula
- **Backward symbolic execution** can support it!
  - BI and BH run a remote, backward, symbolic debug session



Challenge computation example

```
char foo(unsigned char c) {
 if(c > 0) { c = c + 1; }
 unsigned char d = c + 1;
 char x = 42/d;
 return x;
}
```

Challenge computation example

```
char foo(unsigned char c) {
 if(c > 0) { c = c + 1; }
 unsigned char d = c + 1;
 char x = 42/d;            <-- division by 0!
 return x;
}
```

Challenge computation example

```
char foo(unsigned char c) {
 if(c > 0) { c = c + 1; }
 unsigned char d = c + 1;    // d≡0
 char x = 42/d;              <-- division by 0!
 return x;
}
```

Challenge computation example

```
char foo(unsigned char c) {
 if(c > 0) { c = c + 1; }    // c+1≡0
 unsigned char d = c + 1;    // d≡0
 char x = 42/d;              <-- division by 0!
 return x;
}
```

Challenge computation example

```
char foo(unsigned char c) {  // (c>0 ∧ c+2≡0) ∨ (c≤0 ∧ c+1≡0)
 if(c > 0) { c = c + 1; }    // c+1≡0
 unsigned char d = c + 1;    // d≡0
 char x = 42/d;              <-- division by 0!
 return x;
}
```

Challenge computation example

```
char foo(unsigned char c) {  // (c>0 ∧ c+2≡0) ∨ (c≤0 ∧ c+1≡0)
 if(c > 0) { c = c + 1; }    // c+1≡0
 unsigned char d = c + 1;    // d≡0
 char x = 42/d;              <-- division by 0!
 return x;
}
```

Challenge: return valid `c` satisfying `(c>0 ∧ c+2≡0) ∨ (c≤0 ∧ c+1≡0)`

Challenge computation example

```
char foo(unsigned char c) {  // (c>0 ∧ c+2≡0) ∨ (c≤0 ∧ c+1≡0)
 if(c > 0) { c = c + 1; }    // c+1≡0
 unsigned char d = c + 1;    // d≡0
 char x = 42/d;              <-- division by 0!
 return x;
}
```

Challenge: return valid c satisfying $(c>0 \land c+2\equiv 0) \lor (c\leq 0 \land c+1\equiv 0)$

Answer: 254

Conclusion and future work

- VeriOSS aims to support **fair**, **reliable** and **open** market for bug bounty programs
- Solidity contracts are under development

NEXT STEPS

- Equilibria for partial rewards
- Alternative challenge-response implementations
- Formal verification of the protocols

Thank you

Extra: Implementation

- Challenge takes a state as a byte vector
  - Decommits and solve the challenge
  - Transfer in case of success
- Solve returns true only if the provided state is a valid model for the symbolic constraints
- Decommit checks the state hash
- BI can revoke the contract after a while

```
contract PartialReward {

  address public hunter = /* ... */;
  uint    public reward = /* ... */;
  uint    public expire = /* ... */;

  function challenge(bytes4[] state) public {
    if(decommit(state) && solve(state))
      hunter.transfer(reward);
  }

  function solve(bytes4[] state) private
  returns (bool)
  {
    if(state[0] <= 255) // not (a > 255)
      return false;
    return true;
  }

  function decommit(bytes4[] state) private
    returns (bool) { /* check state hash */ }

  function timeout() public {
    require(now >= expire);
    selfdestruct(this);
  }
}
```

Extra: Why should BH and BI follow the protocol?

Calculating the initial precondition is costly

The initial disclosure by the BH should motivate the BI to pay this cost

Hold up: the BH can ask for a higher payment after the BI computes the weakest precondition

Total payment is established before the BI computes the weakest preconditions

BI could intentionally interrupt the protocol once he learns enough

The BH still erns the partial payments (which should be properly designed)