A Blockchain-Based Bug Bounty Program for Genomics

Yuhang Cui, Computer Science & Math
Advisors: Prof. Fan Zhang , Dept. of Computer Science; Prof. Hoon Cho, School of Medicine
Math Reader: Prof. Yair Minsky, Dept. of Mathematics, Yale University

Introduction

The rising availability of genome data services for bioinformatics research leads to severe concerns about individual genome privacy, since certain exploits allow extraction of individual genomes without consent.

This project aims to allow proactive discovery and remedy of such exploits with a bug bounty program, where exploit source code can be traded for monetary rewards, satisfying the following conditions:
> **Fair Exchange:** Service provider gets exploit code if and only if exploit submitter gets bounty payment
> **Verifiable:** Both party can verify the integrity of the protocol workflow.
> **Privacy-Preserving:** Reference data and exploit code is only accessible to submitter and service provider.

Results Summary

> Implemented a **verifiable and privacy-preserving** bug bounty protocol that allows **fair exchange** of exploit code and payment.

> Low on-chain computation cost: around 2 million gas (~$90) for each protocol cycle.

> Low computational overhead for proofs using **Intel SGX**.

> Fast verification time using **locality-sensitive hashing**.

> Simulation runtime about **4 seconds** for 10,000 samples of length 10,000.

Background

> **Blockchain:** Distributed append only ledge of transaction history.
> **Smart Contract:** Programs stored and executed on blockchain. Code and internal states fully transparent.
> **Trusted Execution Environment/Intel SGX:** Hardware support for trusted computing. Generates verifiable attestation to execution and output integrity.
> **Scale of Human Genome Data:** Single sequences up to tens of GB. Database size typically on orders of TB.
> **Locality-Sensitive Hashing:** Allows fast approximate nearest-neighbor search.

Protocol Overview

The general idea for the protocol is to have the submitter generate an attestation for a valid exploit in SGX, upload the attestation and encrypted code to a smart contract for verification, and claim the reward by uploading the decryption key once verification finishes.

The service provider could then retrieve the encrypted code and decryption key to get the plain source code. The process also use asymmetric encryption to keep exploit code hidden.

1. Verifier Retrieval: Submitter retrieves verifier program, asymmetric encryption key, and SGX manifest files.
2. Exploit Verification: Submitter executes verifier program in SGX to generate proof for a working exploit.
3. Upload Commitment: Submitter uploads SGX attestation and commitment for exploit code and encrypted key
4. Exploit Code: Submitter sends encrypted exploit code to service host, who checks its integrity against the commitments and confirm code reception.
5. Exploit Verification: Submitter executes verifier program in SGX to generate proof for a working exploit.
6. Atomic Swap: Submitter uploads the decryption key for exploit code and gets bounty reward in an atomic operation.
7. Exploit Recovery: Service host recovers exploit code with encrypted code from step 4 and decryption key from step 6.

All information uploaded by submitter is public key encrypted, so only the host can decrypt them with the private key.

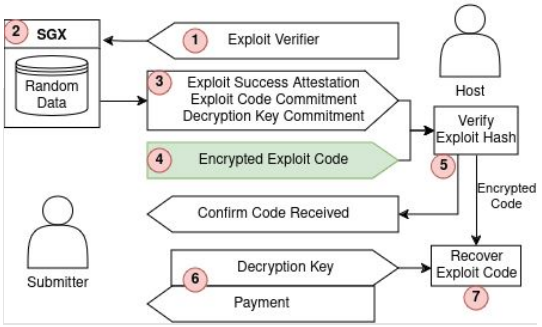

Diagram of protocol workflow. White transactions are made through smart contract, green transaction is off-chain.

SGX Verifier

$verifier(k_{submitter}, \mathsf{pk}, exploit, interface)$

1: $exploit\_encrypted \leftarrow \mathsf{Enc}(k_{submitter}, \mathsf{Enc}(\mathsf{pk}, exploit))$
2: **output** $exploit\_encrypted$
3: $exploit\_hash \leftarrow SHA256(exploit\_encrypted)$
4: **output** $exploit\_encrypted$
5: $key\_hash \leftarrow SHA256(\mathsf{Enc}(\mathsf{pk}, k_{submitter}))$
6: **output** key_hash
7: $data \leftarrow\$ data\_generator()$
8: $outputs \leftarrow exploit(interface(data))$
9: **if** $\exists d \in outputs : d \in data$ **then**
10: **output** $True$
11: **else**
12: **output** $False$
13: **fi**

Smart Contract

$user\_registration(address)$

1: **assert** $msg.sender = owner$
2: $verified.users[address] \leftarrow True$

$code\_received(address)$

1: **assert** $msg.sender = owner$
2: $code\_received[address] \leftarrow True$

$submission\_handler(attestation, key\_hash, exploit\_encrypted)$

1: **assert** $verified.users[msg.sender] = True$
2: **assert** $verify\_attestation(attestation, key\_hash, exploit\_encrypted) = True$
3: $key\_hashes[msg.sender] \leftarrow key\_hash$
4: $verification\_success[msg.sender] \leftarrow True$

$reward\_handler(k.encrypted_{submitter})$

1: **assert** $verified.users[msg.sender] = True$
2: **assert** $verification\_success[msg.sender] = True$
3: **assert** $key\_hashes[msg.sender] = SHA256(k.encrypted_{submitter})$
4: **assert** $code\_received[msg.sender]$
5: $token.transfer(msg.sender)$

Analysis

> **Fair Exchange:** The submitter can retrieve the bounty if and only if the service host have both the encrypted code and decryption key, and thus can recover the exploit code.
> **Verifiable:** SGX attestation and smart contract source code fully verifiable.
> **Privacy-Preserving:** Reference data never used. Exploit code only decryptable by host due to asymmetric encryption.
> **Cost Efficiency:** On-chain operations are low constant cost. Expensive exploit verification and data exchange is off-chain.
> **Time Efficiency:** SGX and random data generation introduce little overhead to exploit verification process, compared to traditional zero-knowledge proofs.
> **Security:** Resistant against common security issues on blockchain protocols (replay, front-running, etc)

Future Work

> **Large Scale Speed Testing:** Test performance with large datasets under hardware SGX.
> **Reference Data Testing:** Modify the protocol to allow testing on reference data rather than random data.
> **Local Verifier Security:** Ensure verifier is protected against exploits within TEE enclave.