A Blockchain-based Framework for Bug Bounty Programs in Genomics

Yuhang Cui
yuhang.cui@yale.edu

Computer Science Advisor: Fan Zhang
f.zhang@yale.edu

Bioinformatics Advisor: Hoon Cho
hoon.cho@yale.edu

Math Reader: Yair Minsky
yair.minsky@yale.edu

A Senior Thesis as a partial fulfillment of requirements for the Bachelor of Science in Computer Science

Department of Computer Science
Yale University
May 2, 2024

Acknowledgements

I would like to express my gratitude to Professor Fan Zhang for providing guidance on the blockchain and cryptography aspects of this project, and Professor Hoon Cho for providing insight into various bioinformatics tools and protocols, this project would not have been possible without the generous help from my advisors. I would also like to thank my math reader Professor Yair Minsky for providing guidance on meeting the mathematical content requirements, as well as Professor Sohee Park and DUS of both the Math and Computer Science departments for managing and overseeing the senior project course program.

Special thanks to the Yale Department of Computer Science and Department of Mathematics for providing a fantastic environment for academic growth. I'm grateful for the opportunities and resources that have enriched my time here.

Contents

A Blockchain-based Framework for Bug Bounty Programs in Genomics

Yuhang Cui

Abstract

The goal of this project is to design and implement a program to proactively identify exploits in genome data services, taking advantage of the transparency properties of blockchain to hold the service provider accountable for identifying and handling such exploits. The sensitive nature of genome data and their large data size posed interesting challenges to the protocol design process due to the expensive on-chain computation cost. The implemented protocol allows the secure exchange of verified exploit codes for bounty tokens without revealing any sensitive information, with verifiable guarantees of fair exchange. This project demonstrates the feasibility of such protocols and provides a sample implementation for future work.

Chapter 1

Introduction

1.1 Motivation

With the rising popularity of genomics research and publicly available genome data services, protecting the privacy of individuals in genome datasets is of utmost importance. Many such services provide only summary-level data, restrict the scope of supported queries, or use access-control mechanisms to prevent data leakage. However, such mechanisms could fail due to exploits and inference attacks. For example, genotype imputation services, which use statistical inference to fill in missing genotypes, are vulnerable to data reconstruction attacks with malicious queries [1]. This attack allows the genomes of individuals to be extracted accurately, which is a severe breach of privacy as the data can be used to discriminate and identify individuals based on their genetic information.

Considering the severe privacy concerns of genome data leaks, meticulous measures should be taken to proactively identify and prevent such attacks. One way of discovering such exploits is through *bug bounty programs*, where attackers who manage to exfiltrate secret data can claim a bounty (i.e., monetary rewards) and thus are incentivized to alert the service provider, who can then fix the exploit and prevent further damage.

However, implementing trustworthy bug bounty programs can be challenging, since the bug submitter has an incentive to get the reward with unhelpful exploits, and the service provider has an incentive to censor the existence of exploits and not provide the reward. Using sensitive data also poses another challenge, since the exploit verification needs to prove the correctness of the exploit without exposing any private information.

1.2 Background

1.2.1 Human Genome Data

The size of modern human genome datasets used in bioinformatics is rapidly increasing, often involving terabytes of data. For large databases with hundreds of thousands of genome samples, operations on the dataset are computationally intensive in both time and memory, usually taking many hours, making it impossible to use certain technologies with slow computation time and limited memory.

1.2.2 Blockchain and Smart Contracts

Blockchain is a distributed, append-only ledger of verified transaction history, maintained by many participating nodes. For each new block containing transactions like currency transfers, the nodes will use some distributed consensus protocol to decide which transactions are included, so that every node can agree on the next block to add to the transaction history ledger. This ever-growing list of transaction histories provides a convenient platform for hosting bug bounty programs, since it guarantees the immutability and availability of verified transactions, and also provides pseudo-anonymous payment methods through cryptocurrency and tokens.

Many blockchains also allow transactions to include function calls to programs stored on the blockchain, called smart contracts, which provide a verifiable platform for code execution. The source code, internal state updates, and transaction input information of smart contracts are visible to everyone on the blockchain, making it easy to verify the integrity of these programs. A thorough exploration of the applications of smart contracts was done in [2].

Some desirable properties of smart contracts for the bug bounty program are:

- **Availability**: The blockchain network should always be accessible as long as some node is active, in contrast to private servers that are prone to downtime due to failures and attacks.

- **Transparency**: Verified transactions should be visible to anyone with access to the blockchain, preventing exploit submissions from being censored by the service provider. This also allows the integrity of any smart contract to be verified since its source code is visible.

- **Immutability**: It is not possible to change verified transactions on a block, guaranteeing previously verified transactions to always be visible.

- **Verifiability**: Many blockchain systems are designed to be verifiable on a consumer computer, so the transaction history can be independently verified without trusting a single party.

These properties can keep both the submitter and the service provider accountable during the transaction and do not require any trust since everything is verifiable. However, these properties also pose some challenges, since the transparency property will expose any private data or exploit code to every node on the blockchain.

The distributed nature of blockchains also makes smart contracts very expensive to run and store, since many consensus nodes need to execute and store the same program, costing computing resources. Due to the large size of genome data, it is prohibitively expensive to perform the program verification on-chain, so some method to test the program off-chain and provide verifiable proof is needed.

1.2.3 Encryption

Encryption allows the conversion of a readable plaintext into an encrypted ciphertext using pieces of data called keys. There are two types of common encryption schemes: symmetric and asymmetric.

Symmetric encryption uses a single key k to encrypt and decrypt data. We get a random key k with $\mathsf{KGen}()$, encrypt $text$ with k using $\mathsf{Enc}(k, text)$ for the encrypted text, and decrypt $text\_encrypted$ with k using $\mathsf{Dec}(k, text\_encrypted)$ to get the plaintext back. A sample use is shown below:

```
1 : k ←$ KGen()
2 : text ← "sample text"
3 : text_encrypted ← Enc(k, text)
4 : text_decrypted ← Dec(k, text_encrypted)
5 : text = text_decrypted
```

Asymmetric encryption uses a public key pk to encrypt the data, and a private/secret key sk to decrypt the data. We get a pair of keys pk and sk using $\mathsf{KGen}()$, encrypt $text$ with pk, and decrypt the cipher text with sk to get the plain text back. A sample use is shown below:

```
1 : (pk, sk) ←$ KGen()
2 : text ← "sample text"
3 : text_encrypted ← Enc(pk, text)
4 : text_decrypted ← Dec(sk, text_encrypted)
5 : text = text_decrypted
```

There are many encryption algorithms with varying speed and security. We will assume secure encryption algorithms are provided.

1.2.4 Trusted Execution Environments

One commonly used method to provide verifiable proof of knowledge is zk-SNARK, which constructs small and verifiable proof that the submitter has a piece of information, like a valid exploit source code, without revealing the information itself [3]. However, generating such proofs is computationally expensive, especially when it involves large data. Instead, we will use Trusted Execution Environment (TEE), which relies on hardware features like Intel SGX to generate verifiable attestation of valid executions and protect data privacy. However, due to the history of side channel attacks on SGX that extracts private data, we will assume that TEE attestations guarantee execution integrity but does not protect against data leaks. This assumption still allows verifiable bug bounty programs to be implemented, as described in the Sealed-Glass Proofs paper [4].

A common way to execute programs in Intel SGX is using Gramine [5], which uses a manifest file to determine which system files are needed to execute the program, and checks the validity of these files. Gramine can execute programs in the host system environment, as long as the necessary files are specified in the manifest file.

1.2.5 Locality Sensitive Hashing

Verifying the output sequence of an exploit against a large database of very long genome sequences is very time-consuming, and the verifier should also allow a small amount of error in the output of a large genome sequence. One way to achieve this is through locality-sensitive hashing, which has a high probability of hashing similar sequences to the same hash buckets, allowing fast identification of similar sequences. Its effectiveness has been shown for relative identification in the SF-Relate protocol [6]. Locality-sensitive hashing also provides some reverse-image resistance, which can provide additional security compared to certain dimensional reduction techniques.

1.3 Problem Description

The goal of this project is to design and implement a bug bounty program that guarantees the following properties:

1. **Fair Exchange**: The service provider gets access to the exploit code if and only if the submitter gets bounty payment

2. **Verifiable**: Both parties can verify the integrity of the protocol from available information.

3. **Privacy-Preserving**: Only the host and submitter can access the exploit code, and only the host has access to any genome data.

This allows the protocol to guarantee a fair exchange of exploit code and payment without revealing information to third parties. The protocol should also be cost-effective by not performing too much on-chain computation and have good performance even with large data sets.

The bug bounty is intended to be used once per service update: the service provider will need to modify or re-create the bug bounty for a newer version of the service interface, after fixing the previously submitted exploit.

The protocol also makes the following assumptions:

1. Intel SGX attestation guarantees the integrity of code execution and validity of outputs.

2. Exploits that work on a randomly generated database with the same server interface are acceptable.

3. The submitter and service provider always act in their best interest: the submitter always wants the bounty, and the service provider always wants the exploit code.

The second assumption may exclude some exploits that depend on specific data patterns in the original database but will accept any data analysis exploits and exploits in server interface.

In the rest of this report, we will go over some related work in chapter 2, give an intuitive overview of the protocol in chapter 3, analyze the implementation details in chapter 4, test and benchmark a sample implementation in chapter 5, and conclude with some discussion of future work in chapters 6 and 7.

1.4 Contributions

- Designed a verifiable and practical bug bounty protocol for genome data under the Sealed-Glass Proof assumptions.
- Provides a sample protocol implementation in Python and Solidity for reference and testing.

Chapter 2

Related Work

- **Sealed-Glass Proofs**: The design of this protocol is based on the Sealed-Glass Proofs paper [4], which described a bug bounty protocol using a leaky TEE and randomly generated data. This project implements a simpler protocol by taking advantage of random number generation inside TEE and flexibility of smart contracts compared to Bitcoin. The protocol is also an implementation of zero-knowledge contingent payment [7], which implements fair exchange of information on Bitcoin, although the flexibility of smart contracts makes the protocol even simpler.

- **Secure Genome Computation**: The bug bounty program implemented could not verify exploits that works on the original data but not random data. Several protocols have been designed to securely handle genome analysis tasks through multiparty computation [8], homomorphic encryption [9], and locality-sensitive hashing [6]. Protocols like these might be used to create a bug bounty that can securely test on the original data, but careful security analysis is needed to determine the risk of exposing these information.

- **Similarity Search**: This project uses the locality sensitive hashing implementation from Faiss for nearest neighbor search, but Faiss also includes other GPU-accelerated nearest neighbor search algorithms that could be faster or more memory efficient depending on the use case [10]. There are also specialized fast similarity analysis for genome data like RAFFI [11]. An implementer may want to test the performance of different algorithms when deploying their own protocols.

- **Privacy-Preserving Smart Contracts**: Under the Sealed-Glass Proof assumptions, TEE based on Intel SGX cannot be trusted to keep data private, so many privacy-preserving smart contract implementations based on TEE cannot be trusted with private data. However, these privacy-preserving smart contract implementations can provide verifiable execution of smart contracts and can be more cost-effective than on-chain computation, so the can be desirable for cost-effective public data handling and function implementation [12].

Chapter 3

Design Overview

3.1 Protocol Description

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.

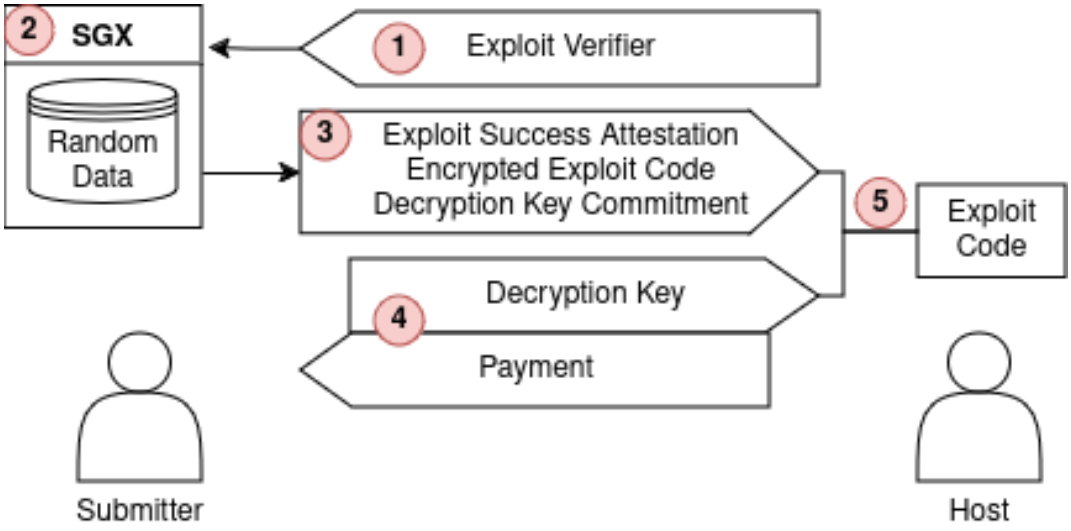

Figure 3.1: Bug Bounty Protocol

The general outline of an execution flow, as shown in Figure 2.1, is:

0. **Setup**: The service provider deploys a smart contract containing an exploit verifier for the service they want to check.

1. **Verifier Retrieval**: Submitter retrieves the verifier files from the smart contract.

2. **Exploit Verification**: Submitter use exploit verifier to test their exploit code with randomly generated data in SGX, generating an attestation.
    - This provides verifiable proof of the validity of the exploit without revealing private data.
    - SGX attestation also verifies the commitment data.
3. **Upload Commitment**: Submitter sends the encrypted exploit code, commitment of decryption key, and the SGX attestation to the service provider, who can then verify the validity of the exploit
4. **Claim Reward**: Once the exploit is verified, the submitter can send the decryption key to the service provider, and get the bounty if the key hash matches in one atomic operation.
    - This guarantees fair exchange: The Submitter gets a reward if and only if the service provider gets the encrypted exploit code and the key to decrypt it.
5. **Recover Exploit**: Since the service provider have the encrypted code and decryption key, they can decrypt the code with the key to get the original exploit code.

The transactions in the protocol can be implemented as functions a smart contract, which allows verifiable transactions, can automatically perform validity checks, and allow atomic operations.

In addition to this, we encrypt everything the submitter uploads with public key encryption, so only the service provider can decrypt the exploit code, this guarantees the privacy of the exploit code. This is based on the bug bounty protocol in the Sealed-Glass-Proof paper, with asymmetric encryption to protect data privacy.

3.2 Goal Satisfaction

This protocol satisfies the following design goals:

1. **Fair Exchange**: The submitter can retrieve the bounty if and only if the encrypted code and decryption key are both uploaded to the smart contract, and the service provider can retrieve them to decrypt the verified exploit code.
2. **Verifiable**: The source code of the smart contract and verification files for SGX are all viewable to the submitter before submitting a verification, so the submitter can verify the integrity of the protocol. The submitted exploit code and hashes are verified by SGX, and the validity of the attestation is verified by the smart contract for the service provider.
3. **Privacy Preserving**: The submitter can only see randomly generated data from a side channel attack on TEE, and the encrypted source code uploaded to the smart contract can only be decrypted with the service provider's private key, so no private information is leaked to third parties.

4. **Cost Efficiency**: Since the computationally-intensive exploit verification is performed off-chain, the on-chain operations only consist of short operations like checking hashes and verifying simple conditions, thus keeping the on-chain cost low.

5. **Time Efficiency**: Random data generation is relatively fast for the data size, and the verifier does not introduce too much overhead to the exploit execution. The service provider can adjust the data size if execution time is a problem.

Chapter 4

Implementation and Analysis

4.1 Workflow Details

4.1.1 Protocol Setup and Definitions

To set up the protocol, the service provider first generates a pair of keys $(\mathsf{pk}, \mathsf{sk})$ for asymmetric encryption:

$$1: \quad (\mathsf{pk}, \mathsf{sk}) \leftarrow\!\!\$\ \mathsf{KGen}()$$

Before using the protocol, the submitter generates a key $k_{submitter}$ for symmetric encryption, and have all exploit code stored in the $exploit$ file:

$$\begin{array}{ll} 1: & k_{submitter} \leftarrow\!\!\$\ \mathsf{KGen}() \\ 2: & exploit \leftarrow \text{"Exploit code"} \end{array}$$

The exploit should take in an interaction interface to simulate server operations, and returns a list of extracted genome sequences.

Smart Contract Data

Then, the service provider initializes a smart contract with the following data to be publically retrievable by a submitter:

- **Manifest File** $manifest$: Specifies any system file needed for program execution under SGX, such as Python, library files, or data files.
- **Public Key** pk: Encrypts the exploit code, so only the service provider can decrypt it.
- **Interface Program** $interface$: Simulates the API interface for the service given a random database. For example, taking an http API request string as input and returns the requested data as JSON string.

- **Verifier Program** $verifier$: Takes exploit program $exploit$, public key pk, submitter key $k_{submitter}$, and interface program $interface$ as input. Performs the following tasks in SGX with $manifest$ and generates an SGX attestation $attestation$:

  1. Encrypt the exploit code with public key and the submitter key, the outputs it:

     $exploit\_encrypted \leftarrow \mathsf{Enc}(k_{submitter}, \mathsf{Enc}(\mathsf{pk}, exploit))$
     output $exploit\_encrypted$

  2. Encrypt the submitter key with the public key, hashes it with $SHA256$, and outputs it:

     $key\_hash \leftarrow SHA256(\mathsf{Enc}(\mathsf{pk}, k_{submitter}))$
     output key_hash

  3. Generate a randomized database with interaction interface defined in $interface$, calls the $exploit$ program with $interface$, and verify the exploit outputs against the database. Output $True$ if some sequence in the output is in the original data, otherwise output $False$.

     $data \leftarrow\!\!\$\ data\_generator()$
     $outputs \leftarrow exploit(interface(data))$
     if $\exists d \in outputs : d \in data$ **then**
         output $True$
     else
         output $False$
     fi

     Due to the large size of genome sequences, the service provider may wish to allow a small amount of error. In that case, the condition can be changed to checking if some output sequence have a similar sequence in the database, with similarity measured by Hamming Distance.

Smart Contract Functions

The smart contract also contains the following functions, note that all arguments of the functions are stored in the calldata of the transaction and is retrievable by the service provider:

- $user\_registration$: Allows owner to update dictonary of registered users, the owner should register a list of verified users (know organizations, trusted researchers, etc) after the contract is created:

$user\_registration(address)$

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

- $submission\_handler$: Takes $attestation, key\_hash, exploit\_encrypted$ as inputs. Verifies if the exploit outputs $True$ with the corresponding key_hash and $exploit\_encrypted$ using the $attestation$. Records the key hash and verification passing in dictionaries:

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

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 : $verification\_success[msg.sender] \leftarrow True$

- $reward\_handler$: Takes $k\_encrypted_{submitter} = \mathsf{Enc}(\mathsf{pk}, k_{submitter})$ as input, gives reward if the hash matches to the submission

$reward\_handler(k\_encrypted_{submitter})$

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 : $token.transfer(msg.sender)$

4.1.2 Exploit submission

The procedure and function calls for submitting an exploit after setup is shown in Figure 3.1.

The exact steps are described below:

1. **Verifier Retrieval**: User retrieves public key pk, verifier program $verifier$, interface program $interface$, and manifest file $manifest$.

2. **Exploit Verification**: The submitter executes $verifier$ in Intel SGX using $manifest$ with the following inputs and outputs:

   - **Input**: $exploit$, pk, $k_{submitter}$, $interface$
   - **Output**: $exploit\_encrypted$, key_hash, $attestation$

3. **Upload Commitment**: The submitter calls the $submission\_handler$ function in the smart contract with the following arguments to verify the exploit and store inputs to calldata:

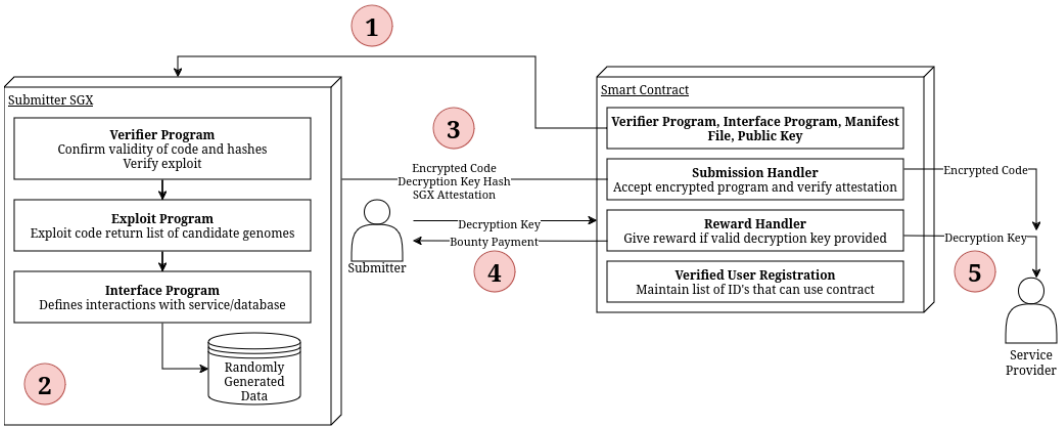

Figure 4.1: Exploit Submission Procedure

$$1: \quad submission\_hander(attestation, key\_hash, exploit\_encrypted)$$

4. **Claim Reward**: Once the exploit is verified, the submitter uploads the encrypted key to claim the bounty, with $k\_encrypted_{submitter} \leftarrow \mathsf{Enc}(\mathsf{pk}, k_{submitter})$ for the following call to $reward\_handler$:

$$1: \quad reward\_handler(k\_encrypted_{submitter})$$

5. **Recover Exploit**: The service provider retrieves $exploit\_encrypted$ and $k\_encrypted_{submitter}$ from the calldata inputs in steps 3 and 4, then use it to retrieve the original $exploit$:

$$\begin{aligned}
&1: \quad k_{submitter} \leftarrow \mathsf{Dec}(\mathsf{sk}, k_encrypted_{submitter}) \\
&2: \quad exploit \leftarrow \mathsf{Dec}(\mathsf{sk}, \mathsf{Dec}(k_{submitter}, exploit_encrypted)) \\
&3: \quad \textbf{return } exploit
\end{aligned}$$

4.2 Pseudocode

Putting it together, we get the following interface for the 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 : $key\_hash \leftarrow SHA256(\mathsf{Enc}(\mathsf{pk}, k_{submitter}))$
4 : **output** key_hash
5 : $data \leftarrow\!\!\$\ data\_generator()$
6 : $outputs \leftarrow exploit(interface(data))$
7 : **if** $\exists d \in outputs : d \in data$ **then**
8 : **output** $True$
9 : **else**
10 : **output** $False$
11 : **fi**

Which will be executed under SGX using the $manifest$ provided and have an attestation verifying the validity of all outputs.

For the smart contract, we get the following interface layout:

$user\_registration(address)$

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

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

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 : $token.transfer(msg.sender)$

4.3 Alternative Design: Off-Chain Messaging

For larger exploit programs, the protocol can be modified to send $exploit\_encryted$ through off-chain communication to save gas cost.

Rather than sending $exploit\_encrypted$ to $submission\_handler$, the submitter sends $exploit\_hash = SHA256(exploit\_encrypted)$, and sends the full $exploit\_encrypted$ to

the service provider through off-chain communication. The service provider will check $exploit\_encrypted$ against its hash to verify its integrity.

Once the service provider verified the integrity of $exploit\_encrypted$, they can use a map to mark a user's submission as received, and $reward\_handler$ will check that map before giving out the reward. The workflow of the new protocol is shown in Figure 3.2.

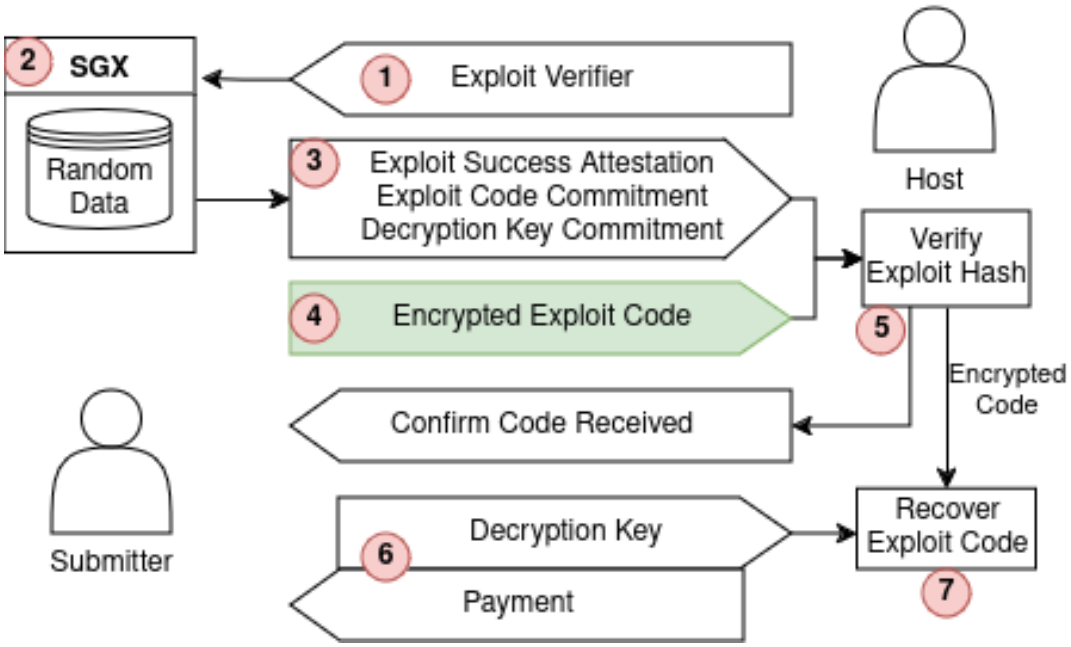

Figure 4.2: Off-Chain Messaging Protocol. Green transaction is off-chain.

The updated $verifier$ and smart contract code is below:

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

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** $exploit\_hash$
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**

$user\_registration(address)$

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

$code\_received(address)$

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

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

1 : **assert** $verified\_users[msg.sender] = True$
2 : **assert** $verify\_attestation(attestation, key\_hash, exploit\_hash) = True$
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** $code\_received[msg.sender] = True$
5 : $token.transfer(msg.sender)$

So the service provider will need to call $code\_received$ with the submitter's address once they receive $exploit\_encrypted$ offline to continue the protocol.

The Analysis section will show this protocol still give the same security guarantees under the assumptions, while making all smart contract calls constant cost.

4.4 Protocol Analysis

We will analyze how the protocol satisfies the design goals under the initial assumptions.

Lemma 1. *If the submitter executes $submission\_handler$ function successfully, then the service provider has access to the encrypted code of a verified functional exploit on the service, as well as a verifiable commitment to its decryption key.*

Proof. Under the Sealed-Glass Proof assumptions, Intel SGX attestation guarantees the validity of $verifier$'s outputs. So $verifier$ outputs $True$ if and only if $exploit$ successfully extracts some genome sequence d in the randomly generated data, thus showing that it is a valid exploit.

The integrity of commitment data $exploit\_encrypted$ and key_hash (and $exploit\_hash$ in the off-chain version) are also guaranteed by SGX, and verifiable with the attestation.

Since the inputs of smart contract function calls are visible to everyone, including the service provider, once the submitter calls $submission\_handler$, both key_hash and $exploit\_encrypted$ will be accessible by the service provider.

Since the outputs of $verifier$ are verifiable with $attestation$, the $submission\_handler$ can only pass the asserts if both key_hash and $exploit\_encrypted$ are valid SGX outputs. Thus successful execution of $submission\_handler$ guarantees that the service provider can access the valid encrypted exploit code and decryption key commitment. (For the off-chain version, the service provider gets a valid commitment to $exploit\_encrypted$) □

Theorem 1 (Fair Exchange)**.** *The service provider gets $exploit$ if and only if the submitter gets the payment.*

Proof. By the definition of the $reward\_handler$ function, the submitter gets the reward if and only if the last submitted $exploit\_encrypted$ and key_hash are both valid and $SHA256(k\_encrypted_{submitter}) = key\_hash$, with all these information visible to the service provider through the smart contract.

Using the standard assumption that we have access to a collision-resistant hash function $SHA256$, we get that $SHA256(k\_encrypted_{submitter}) = key\_hash$ if and only if $k\_encrypted_{submitter}$ is indeed the original key used to encrypt $exploit\_encrypted$ as guaranteed by SGX attestation, once we decrypt it with sk. Thus we get that $reward\_handler$ executes successfully if and only if the matching $k\_encrypted_{submitter}$ and $exploit\_encrypted$ are visible to the service provider. (In the off-chain version, the additional assert for $code\_received$ guarantees that $reward\_handler$ executes successfully if and only if the service provider received $exploit\_encrypted$ and confirmed it with a call to $code\_received$, the rest of the proof is the same)

Since $k\_encrypted_{submitter} = \mathsf{Enc}(\mathsf{pk}, k_{submitter})$, by definition of asymmetric encryption, since the service provider has sk, they can retrieve $k_{submitter}$ from $\mathsf{Dec}(\mathsf{sk}, k\_encrypted_{submitter})$. Similarly, since they have $exploit\_encrypted$ as $enc(k_{submitter}, \mathsf{Enc}(\mathsf{pk}, exploit))$, they can recover $exploit$ from $\mathsf{Dec}(\mathsf{sk}, \mathsf{Dec}(k_{submitter}, exploit\_encrypted))$. Therefore the submitter gets the reward if and only if $exploit$ is accessible to the service provider, and Fair Exchange is satisfied. □

Theorem 2 (Verifiable)**.** *Both parties can verify the integrity of the protocol and detect any discrepancies.*

Proof. Since the entire protocol is mediated by a smart contract, by the transparency property of blockchain transactions and smart contracts, the code and internal states of the smart contract are visible to both parties. Any output from Intel SGX is also verifiable since its associated $attestation$ is uploaded to the smart contract. Thus all aspects of the protocol are verifiable by both parties, and both parties can detect any violations to the protocol procedure. □

Theorem 3 (Privacy-Preserving)**.** *The private genome data is only visible to the service provider, and the exploit code is only visible to the service provider and the submitter throughout the protocol.*

Proof. The protocol does not require the service provider to use any private genome data, thus it is only visible to the service provider.

The exploit code is only visible on the smart contract as: $exploit\_encrypted$, which is $\mathsf{Enc}(k\_encrypted_{submitter}, \mathsf{Enc}(\mathsf{pk}, exploit))$. Under the standard assumption that asymmetric encryption is secure. $exploit$ cannot be recoverwed without sk, which only the service provider have access to. Thus $exploit$ is only visible to the service provider and the submitter who created it. □

4.4.1 Runtime of $verifier$

If the data generated have n genome sequences of length d, the data generation time is $O(nd)$ for independently generating each sequence position. The execution time of $exploit$ is some unknown function of n, d, and the max number of outputs k, say $f(n, d, k)$, assuming the effect of random variation is negligible. If we do the nearest neighbor search with locality-sensitive hashing, there exist implementations that take $O(nk)$ time to search all k results using locality-sensitive hashing[13], with some polylog factor overhead for initial indexing. Thus the overall time complexity is $O(n(d+k) + f(n, d, k))$ times some polylog factor.

4.4.2 Smart Contract Cost

Block-chains charge a fee for executing and storing smart contract functions, usually called gas.

The service provider incurs the cost of creating the contract, setting up the bounty, registering users, and marking received codes for the off-chain version. The contract creation cost can be high with large verifier files, so the service provider may want to look into off-chain storage like IPFS [14].

The submitter incurs the cost of calling $submission\_handler$ and $reward\_handler$. For the on-chain version, the cost of submitting a large exploit may be high, while the off-chain version has a constant cost for any file size.

The Results section will show gas usage on a sample implementation of the off-chain submission protocol.

4.5 Security

Although we have shown the validity of the protocol, there are still some security considerations in a practical application:

- **Replay Attacks:** It is very difficult to distinguish different implementations of the same exploit, which is why the protocol should only have enough balance for one bounty claim during each update cycle, so any exploit on that service version can only be used to claim one bounty.

- **Front-Running Attacks:** Due to the transparency of transactions, it is possible to manipulate the block mining order, delay the $reward\_handler$ call, and submit both the exploit and the decryption key of another person's exploit before the

original submitter transaction is mined. This can be mitigated by incorporating the submitter's identity into the submission. For example, adding the submitter's address as part of SGX's attested output.

The service provider also have incentive to perform the attack and recover the bounty cost, this can be mitigated by locking the bounty recipient to the last successful exploit submitter for a certain time, so only the original submitter can claim the reward during that time period, and can refuse to upload the decryption key if the lock targets someone else.

- **Brute Force Attacks**: Since the exploit verification is completely off-chain, it is possible for a submitter to attempt invalid exploits on many random datasets until it works. For very large sequence lengths, this may not be a concern due to the low probability of success and long runtime. To fully mitigate the issue, the verifier code can call a smart contract function to record each attempt's timestamp, which can detect suspicious attempt frequencies and incur a transaction cost for each attempt, thus discouraging brute-force attacks.

- **Verifier Security**: The security of the protocol does depend on the verifier program only allowing valid exploits to pass. However, the verifier program itself may be vulnerable to exploits during local execution. The Sealed-Glass Proof assumptions only allow oblivious adversaries that cannot change execution flow, but the service provider may want to carefully analyze the verifier program first to avoid exploits on the verifier itself.

- **Intel SGX Security**: The assumption that SGX attestation guarantees execution integrity may not always hold if the attestation system itself was compromised. zk-SNARK does not rely on hardware features for security guarantees, and its computational cost may be feasible for smaller programs. The original Sealed-Glass Proof paper provides a bug bounty protocol using zero-knowledge proofs, by having the service provider providing a random dataset after the code is committed [4].

Chapter 5

Results

5.1 Implemented Solution

A sample implementation of the protocol can be found on https://github.com/notyu-yu/senior_project

The sample implementation uses Python under Gramine for executing the verification program under simulated SGX. Refer to the README file for details.

A sample smart contract is included, using a template interface for payment token and SGX attestation verification. To fully implement on-chain SGX verification, an existing on-chain contract can be invoked, and some sample implementations are shown in [15] [16] [17].

5.2 Sample Usage

A sample test execution following the project README is:

0. **Initialization**: Create the verifier scripts and execute the key generation scripts, you should have the following files:

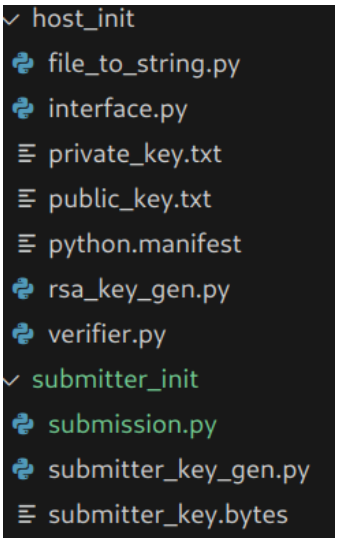

Figure 5.1: Initialization Files

0.c. **Deploy Contract**: Copy the relevant file strings into conracts/zkcp-interface.sol, then deploy the GenomeBounty smart contract in Remix [18], and add the verified users.

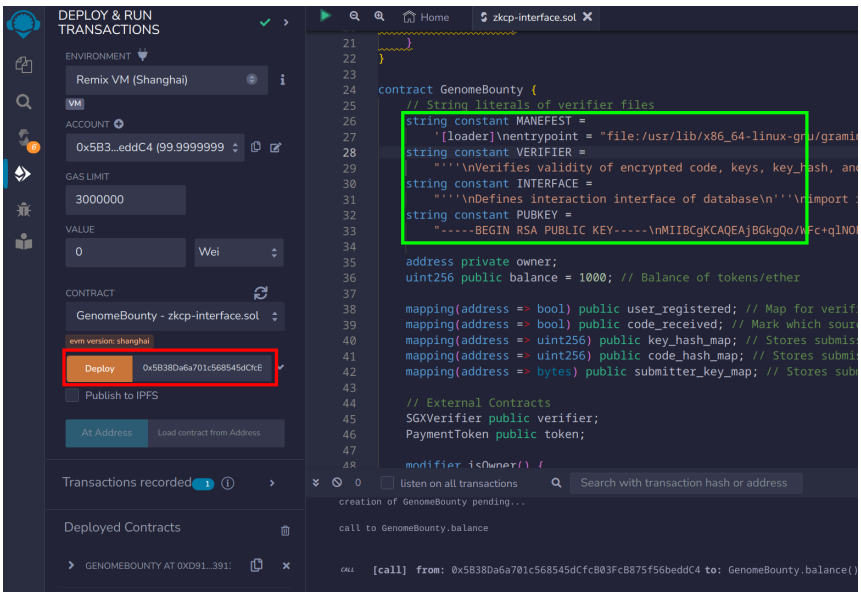

Figure 5.2: Deploying in Remix. Paste string literals to variables in green, then copy owner address to red region and deploy.

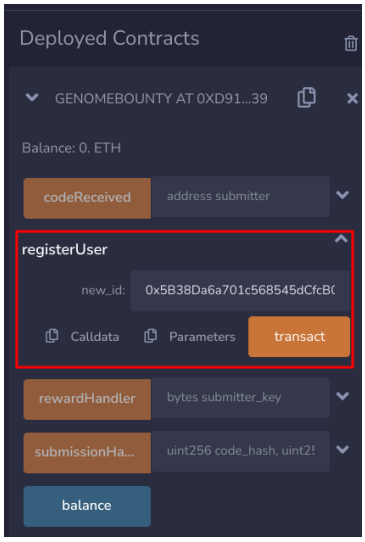

Figure 5.3: Registering a user.

1. **Retrieve Verifier**: Retrieve the strings from smart contract and write them to the relavent files in python/verification_code, also include the submitter's files. The Solidity source code can be publicly viewed if the contract is verified on Etherscan [19].

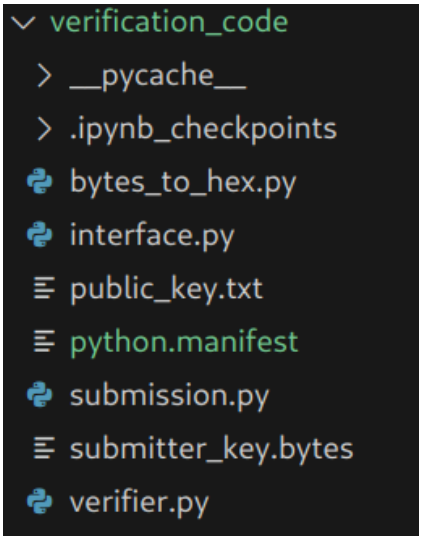

Figure 5.4: Files needed for verification.

2. **Exploit Verification**: Execute verifier.py in with gramine-direct (or just python for quick testing).

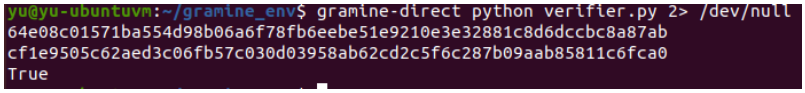

Figure 5.5: Output from verifier when executed in Gramine: key hash, code hash, verification result.

3. **Upload Commitment**: Copy the key and code hashes from verifier output, call $submissionHandler$ with them as arguments.

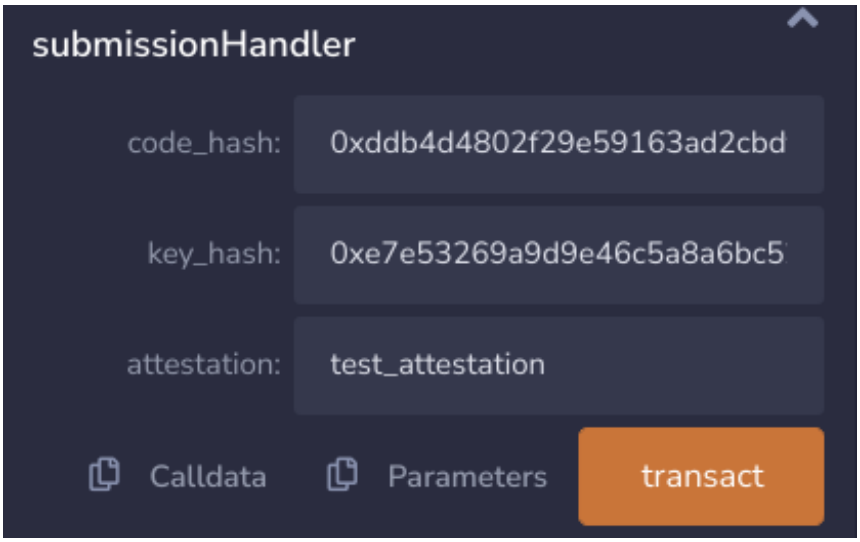

Figure 5.6: Calling submission handler with verifier outputs.

4. **Off-Chain Communication**: Submitter sends encrypted_code.bytes output by the verifier to the service provider. The service provider checks the hash of the received code against the smart contract commitment.

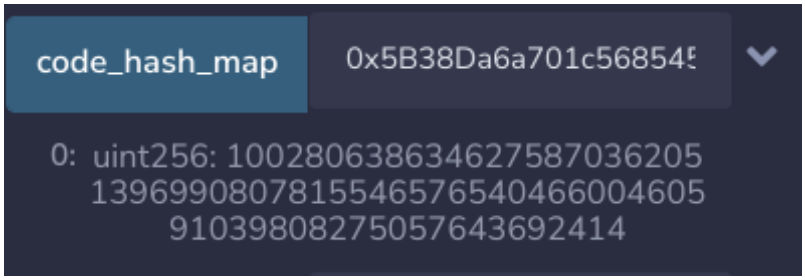

Figure 5.7: Retrieving code hash from smart contract. This can also be recovered from calldata without gas cost.

5. **Code Retrieved Verification**: Service provider calls $codeReceived$ with submitter address.

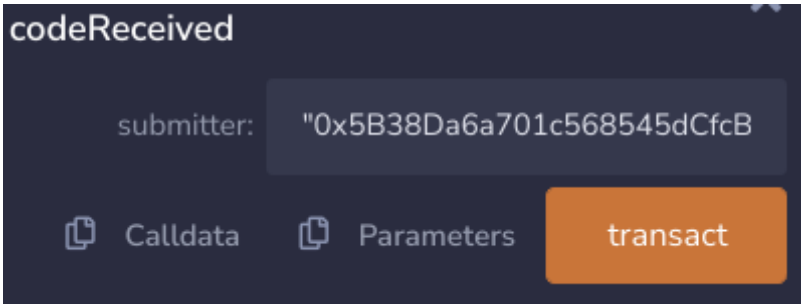

Figure 5.8: Calling $codeReceived$.

6. **Claim Reward**: Use bytes_to_hex.py on encrypted_key.bytes to get the encrypted key in hexadecimal, then call $rewardHandler$ with the encrypted key as input.

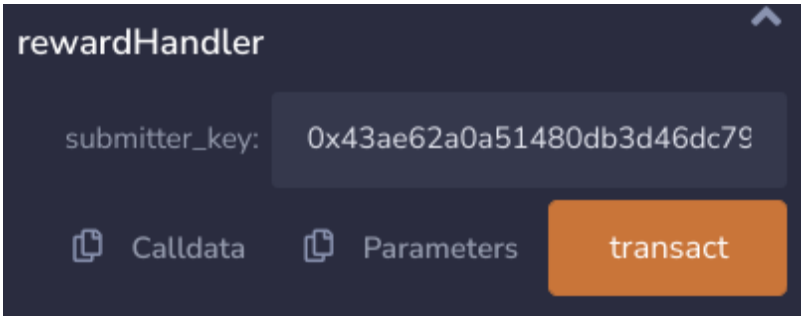

Figure 5.9: Calling $rewardHandler$.

7. **Recover Exploit**: Retrieve the encrypted key, and use the helper scripts in python/host_decryption to recover the exploit code.

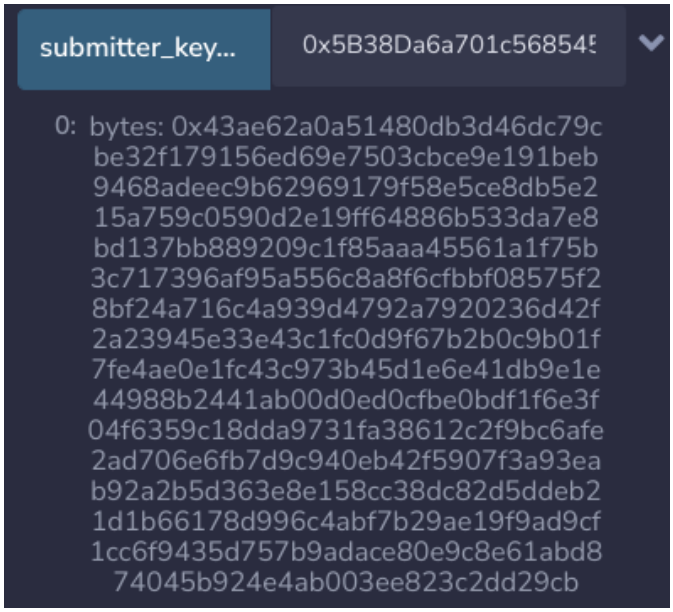

Figure 5.10: Recover encrypted key.

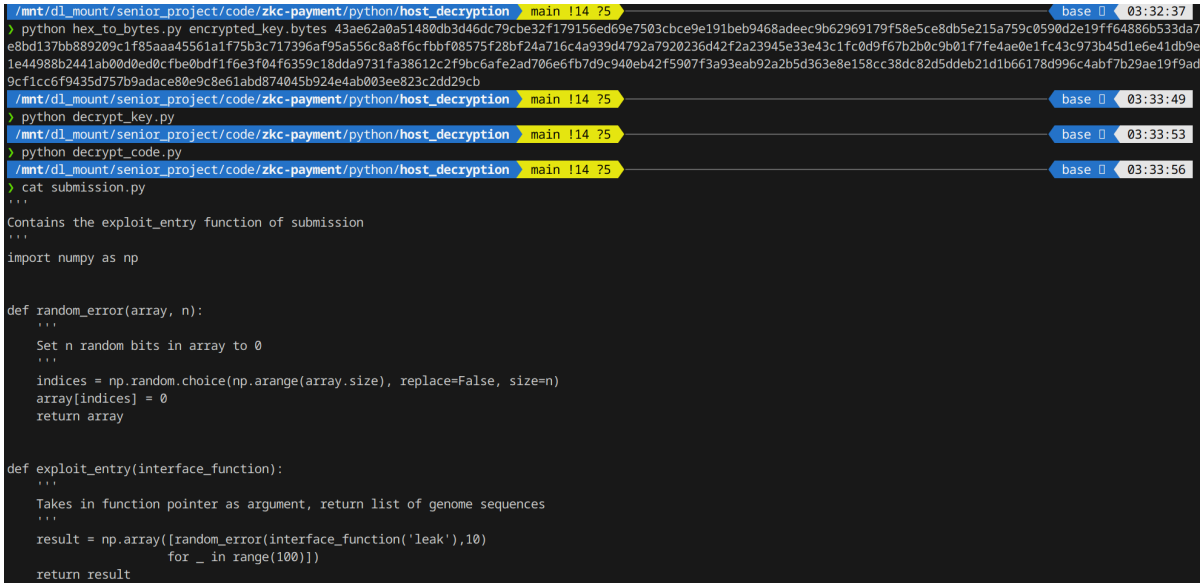

Figure 5.11: Recover exploit code.

5.3 Gas Cost

The sample implementation have the following total transaction gas costs in a Remix VM, note that token transfer and SGX verification are only using template implementations. Assuming gas price is 14 gwei and using price calculator from https://eth-converter.com/.

- Contract Creation: 1647200
- registerUser: 46368
- submissionHandler: 85221
- codeRetrieved: 46412
- rewardHandler: 244032
- Service Provider Total: 1739980 gas ≈ $ 75.10
- Submitter Total: 329253 gas ≈ $ 14.21

5.4 Small Data Performance Test

With a database of $n = 10000$ sequences of length $d = 10000$, the runtime of the sample exploit verifier in gramine-direct in a virtual machine is about 4.5 seconds. Executing directly in Python in the virtual machine takes about 2 seconds, so there is a noticeable slow-down with Gramine.

Chapter 6

Conclusions

This project aims to implement a bug bounty program for genome data, with the goal of ensuring fair exchange, allowing both parties to verify the integrity of the protocol, and preserve the privacy of reference data and exploit code.

We designed a protocol based on Sealed-Glass Proof bug bounties, added asymmetric encryption to preserve data privacy, and used locality sensitive hashing verifier under TEE to efficiently generate a verifiable attestation, thus creating a practical privacy-preserving bug bounty programs for genome data.

This protocol demonstrates the feasibility of using blockchain as a platform for secure bug bounty programs for genome data, or any large data service. It also demonstrates the necessity for secure TEEs for certain applications, as zk-SNARK can be impractical for large programs, and even leaky TEEs are very useful for certain applications.

Chapter 7

Future Work

Due to the limited time and resources for this project, there are many aspects that can be improved or further tested:

- **Speed Testing**: Due to restricted access to processors with SGX and limited computational resource, this project did not test the speed of the verification program using large data sets under SGX. Further testing could help determine the expected execution time of the program, and identify possible optimizations.
- **Local Execution Security**: The current protocol assumes that the verifier does not leak the randomly generated data to the exploit program, but there are no protection for the local database and program memory, so further analysis is needed to make sure the exploit does not take advantage of local execution bugs.
- **Testing on Reference Data**: The current protocol can only verify programs that work on random data. However, there may be exploits that are specific to certain properties of the reference data itself, so another protocol is needed if the service provider wishes to securely test on reference data.

Bibliography

[1] Matthew J. Mosca and Hyunghoon Cho. “Reconstruction of private genomes through reference-based genotype imputation”. In: *Genome Biology* 24.1 (Dec. 2023), p. 271. ISSN: 1474-760X. DOI: 10.1186/s13059-023-03105-6. URL: https://doi.org/10.1186/s13059-023-03105-6.

[2] Tsung-Ting Kuo et al. “Blockchain-enabled immutable, distributed, and highly available clinical research activity logging system for federated COVID-19 data analysis from multiple institutions”. en. In: *J Am Med Inform Assoc* 30.6 (May 2023), pp. 1167–1178.

[3] Nir Bitansky et al. “From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again”. In: *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*. ITCS ’12. Cambridge, Massachusetts: Association for Computing Machinery, 2012, pp. 326–349. ISBN: 9781450311151. DOI: 10.1145/2090236.2090263. URL: https://doi.org/10.1145/2090236.2090263.

[4] Florian Tramèr et al. “Sealed-Glass Proofs: Using Transparent Enclaves to Prove and Sell Knowledge”. In: *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. 2017, pp. 19–34. DOI: 10.1109/EuroSP.2017.28.

[5] “Gramine”. In: URL: https://gramineproject.io/.

[6] Matthew M. Hong et al. “Secure Discovery of Genetic Relatives across Large-Scale and Distributed Genomic Datasets”. In: *bioRxiv* (2024). DOI: 10.1101/2024.02.16.580613. eprint: https://www.biorxiv.org/content/early/2024/02/20/2024.02.16.580613.full.pdf. URL: https://www.biorxiv.org/content/early/2024/02/20/2024.02.16.580613.

[7] G. Maxwell. “Zero knowledge contingent payment”. In: URL: https://en.bitcoin.it/wiki/Zero%20Knowledge%20Contingent%20Payment.

[8] Hyunghoon Cho, David J. Wu, and Bonnie Berger. “Secure genome-wide association analysis using multiparty computation”. In: *Nature Biotechnology* 36.6 (July 2018), pp. 547–551. ISSN: 1546-1696. DOI: 10.1038/nbt.4108. URL: https://doi.org/10.1038/nbt.4108.

[9] David Froelicher et al. “Truly privacy-preserving federated analytics for precision medicine with multiparty homomorphic encryption”. In: *Nature Communications* 12.1 (Oct. 2021), p. 5910. ISSN: 2041-1723. DOI: 10.1038/s41467-021-25972-y. URL: https://doi.org/10.1038/s41467-021-25972-y.

[10] Jeff Johnson, Matthijs Douze, and Hervé Jégou. *Billion-scale similarity search with GPUs*. 2017. arXiv: 1702.08734 [cs.CV].

[11] Ardalan Naseri et al. "RAFFI: Accurate and fast familial relationship inference in large scale biobank studies using RaPID". In: *PLOS Genetics* 17.1 (Jan. 2021), pp. 1–19. DOI: 10.1371/journal.pgen.1009315. URL: https://doi.org/10.1371/journal.pgen.1009315.

[12] Raymond Cheng et al. "Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contracts". In: *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. 2019, pp. 185–200. DOI: 10.1109/EuroSP.2019.00023.

[13] Jia Pan and Dinesh Manocha. "Fast GPU-based locality sensitive hashing for k-nearest neighbor computation". In: *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. GIS '11. Chicago, Illinois: Association for Computing Machinery, 2011, pp. 211–220. ISBN: 9781450310314. DOI: 10.1145/2093973.2094002. URL: https://doi.org/10.1145/2093973.2094002.

[14] Mathis Steichen et al. "Blockchain-Based, Decentralized Access Control for IPFS". In: *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. 2018, pp. 1499–1506. DOI: 10.1109/Cybermatics_2018.2018.00253.

[15] "Automata DCAP V3". In: URL: https://github.com/automata-network/automata-dcap-v3-attestation.

[16] "EPID Verifier Contract". In: URL: https://github.com/Riderfighter/epid-verifier-contract/tree/main.

[17] "RAVe". In: URL: https://github.com/PufferFinance/rave.

[18] "Remix". In: URL: https://remix.ethereum.org.

[19] "Etherscan". In: URL: https://etherscan.io/.