Abstract

In this project, we built AnonRep++, the first anonymous reputation system resistant to intersection attacks. This is an improvement over AnonRep[1], built by Ennan Zhai and others and published in 2016. AnonRep is a practical anonymous reputation system that allows its users to have the benefits of reputation without letting anyone, even the maintainers of the servers, track the actions of its users; users can post messages anonymously and other users can give feedback on the messages (e.g. by voting the message up or down) anonymously. The system employs a variety of cryptographic primitives including verifiable shuffles and linkable ring signatures to achieve this anonymity. However, AnonRep is prone to long-term intersection attacks. A user with very high reputation may lose her anonymity because people will be able to link any message posted with high reputation to her. We fix this problem by using a blockchain-based solution named AnonRep++ in which users can post messages with any reputation that is less than or equal to their current reputation, while still preserving the anonymity guarantees of the original AnonRep system. This eliminates the possibility of an intersection attack and allows users to easily audit any changes of reputation in the system. In this project, my main contributions were implementing the parts of the project specific to AnonRep++, including CoinShuffle, the blockchain layer and integration with the blockchain layer, and benchmarking AnonRep++ against an analogous Python implementation of AnonRep that we also wrote. The implementation can be found on GitHub\footnote{https://github.com/ianzhou1/anonrep}.
1 Introduction

We define a “reputation system” as a system in which users can both post items (such as items for sale in eBay, links on Reddit, or pictures on Instagram) and give feedback on items (such as rating a seller on eBay, upvoting/downvoting on Reddit, or liking a photo on Instagram). Anonymous reputation systems today like Reddit and Instagram provide nearly zero anonymity to their users; these websites often keep complete histories of what a user has done on the website, so website maintainers or even other users can extract private information from long-term activity data. For example, [2] revealed the purchase histories of eBay users simply by analyzing the activity of pseudonyms on the website.

AnonRep is a system developed by Enman Zhai and others that is meant to make this type of long-term tracking impossible. In this system, users can still post content and give feedback on content, but they can post multiple pieces of content without the pieces easily being linked to each other, and vote on content without the server knowing their identity. The system manages to do this while staying secure against duplicate feedback or tampering with user data. AnonRep uses a series of rounds consisting of a message phase, during which users can post messages, and a feedback phase, during which users can provide feedback in the form of a vote up or a vote down on messages posted in the most recent message phase. Depending on the context of usage, these phases could last anywhere from a few minutes to an entire day. In [1], it is suggested to set the message posting phase to 16 hours and to have the feedback phase happen at the same time as the message posting phase for a scenario like the website Stack Overflow. Before each message phase, there is an announcement phase in which the servers collectively generate a one-time pseudonym for each client. The clients use these one-time pseudonyms to make posts. After each feedback phase, the long-term reputation scores are updated in a reverse-announcement phase. The announcement phase and reverse-announcement phase use a Mix-Net protocol that makes it so that no server can link a long-term pseudonym with a long-term reputation score. One of the problems with AnonRep is its vulnerability to intersection attacks; a user with very high reputation will be known for having such a uniquely-high reputation, and since subsequent posts that she makes will have that reputation, the posts can be linked to her.

Our system, AnonRep++, stores reputation in the Ethereum blockchain instead of linking reputation with long-term pseudonyms. As a result, the
annoucement and reverse announcement phases are not needed. Users have reputation in the form of “reputation coins” that they own in multiple “wallets,” all stored on the blockchain. When a user posts a message, she chooses to expose some number of wallets with reputation coin whose balances will change according to the feedback received on the message. This means that users can post messages with any amount of reputation less than or equal to their true amount of reputation. A vote up translates to adding a reputation coin to one of the wallets and a vote down translates to removing a reputation coin from one of the wallets. It follows that a user cannot lose more reputation due to feedback on a message than the amount of reputation she posted that message with. After the feedback stage, a CoinShuffle round is performed so that users will generate new wallets to post messages in future rounds. Because of the anonymity provided by CoinShuffle, no one else can link the new wallets’ public keys to that user, which mitigates the possibility of an intersection attack.

2 Architecture Overview

Similar to the AnonRep implementation on GitHub\(^2\), AnonRep++ features a coordinator to facilitate peer discovery in the servers and clients, a small number of servers that the clients interface with, and (potentially many) clients.

\(^2\)https://github.com/anonyreputation/anonCred
2.1 Coordinator

An AnonRep++ system has one coordinator. Some of its purposes are:

1. To dictate when the different phases begin and end.
2. To coordinate servers and clients that join.
3. To coordinate the CoinShuffle phase.
4. To keep servers in sync with each other.
5. To update reputations by making transactions on the blockchain.
6. To store a copy of the message board that clients can view.

2.2 Server

The main reason we have multiple servers instead of just having one coordinator server is for scalability; each server handles requests from a subset of clients. In the original AnonRep system, the servers also provide client anonymity during the announcement phase and reverse-announcement phase.

2.3 Client

The clients are the users of the system. They can post messages, provide feedback, view their reputation scores, and view the message board.

3 Details of Cryptographic Building Blocks

In the cryptographic primitives that follow, we use a generator \( g \) for prime multiplicative order \( q \) in the group \( \mathbb{Z}_p^\times \).

For benchmarking/testing purposes, we set \( g = 2203 \), \( p = 16000393 \), and \( q = 666683 \), although the values should be larger to ensure security. Such a larger setting can be found at the top of `util.py`.

3.1 ElGamal

ElGamal is an asymmetric cryptosystem based on the Diffie-Hellman key exchange. Its security is largely based on the discrete logarithm problem in cryptography.
3.1.1 Encryption

In standard ElGamal encryption, given a message \( m \), an ephemeral key \( e \), a private key \( z \), and a public key \( Z = g^z \), two ciphertexts are created. The encrypted output is \( (g^e, m \cdot g^{ez}) \).

In layer-by-layer ElGamal encryption, assume we have three servers with ephemeral keys \( e_j \), private keys \( z_j \), and public keys \( Z = g^{z_j} \) for \( j \in \{1, 2, 3\} \). We are given a message \( m \) to encrypt. Without loss of generality, assume the encryption order is \( \{1, 2, 3\} \). The encryption is as follows.

- Server 1 encrypts \((1, m)\) as \((g^{e_1}, m \cdot g^{e_1z_1})\).
- Server 2 encrypts \((g^{e_1}, m \cdot g^{ez})\) as \((g^{e_1+e_2}, m \cdot g^{(e_1+e_2)(z_1+z_2)})\).
- Server 3 encrypts \((g^{e_1+e_2}, m \cdot g^{(e_1+e_2)(z_1+z_2)})\) as \((g^{e_1+e_2+e_3}, m \cdot g^{(e_1+e_2+e_3)(z_1+z_2+z_3)})\).

3.1.2 Decryption

In standard ElGamal decryption, given ciphertext \((c_1, c_2)\) and a private key \( z \), the decrypted output is \( c_2 \cdot (c_1^e)^{-1} \). For ciphertext \((g^e, m \cdot g^{ez})\), this would give \( m \cdot g^{ez} \cdot ((g^e)^{-1}) = m \cdot g^{-ez} = m \), as desired.

In layer-by-layer ElGamal decryption, assume we have three servers with ephemeral keys \( e_j \) and private keys \( z_j \) for \( j \in \{1, 2, 3\} \). We are given ciphertext \((c_1, c_2)\) to encrypt. Without loss of generality, assume the decryption order is \( \{1, 2, 3\} \). The decryption is as follows.

- Server 1 decrypts \((c_1, c_2)\) as \((c_1, c_2 \cdot c_1^{-z_1})\).
- Server 2 decrypts \((c_1, c_2 \cdot c_1^{-z_1})\) as \((c_1, c_2 \cdot c_1^{-z_1} \cdot c_1^{-z_2})\).
- Server 3 decrypts \((c_1, c_2 \cdot c_1^{-z_1} \cdot c_1^{-z_2})\) as \((c_1, c_2 \cdot c_1^{-z_1} \cdot c_1^{-z_2} \cdot c_1^{-z_3})\).

For ciphertext \((g^{e_1+e_2+e_3}, m \cdot g^{(e_1+e_2+e_3)(z_1+z_2+z_3)})\), this would give

- Server 1 decrypts \((g^{e_1+e_2+e_3}, m \cdot g^{(e_1+e_2+e_3)(z_1+z_2+z_3)})\) as \((g^{e_1+e_2+e_3}, m \cdot g^{(e_1+e_2+e_3)(z_1+z_2+z_3)})\).
- Server 2 decrypts \((g^{e_1+e_2+e_3}, m \cdot g^{(e_1+e_2+e_3)(z_1+z_2+z_3)})\) as \((g^{e_1+e_2+e_3}, m \cdot g^{(e_1+e_2+e_3)(z_1+z_2+z_3)})\).
- Server 3 decrypts \((g^{e_1+e_2+e_3}, m \cdot g^{(e_1+e_2+e_3)(z_1+z_2+z_3)})\) as \((g^{e_1+e_2+e_3}, m)\).

As desired, the decryption ends with the original message \( m \).
3.1.3 Sign
Given a message $m$, a generator $g$, a hash function $H$, and a private key $y$, the signer chooses a random $k$ such that $1 < k < p - 1$ and $\gcd(k, p - 1) = 1$. The signer computes $r = g^k \mod (p - 1)$ and $s = (H(m) - y \cdot r)k^{-1} \mod (p - 1)$. The message is signed with $(r, s)$.

3.1.4 Verify
Given a signature $(r, s)$, a generator $g$, a hash function $H$, and a public key $Y = g^y$, the verifier computes $u = g^{H(m)}$ and $v = Y^r \cdot r^s$. The verifier checks that $u$ and $v$ are equal.

3.2 Mix-Net and Verifiable Shuffle
Since these primitives are used in the announcement/reverse announcement phase and these phases are not present in CoinShuffle++, details on Mix-Net and verifiable shuffle can be found in Ian’s report.

3.3 Linkable Ring Signature
The purpose of the linkable ring signature is to let users provide feedback without exposing their identities; it is only known that the feedback provider is one of the members of the ring. It follows that the bigger the ring, the more the anonymity. The linkable ring signature can also prevent users from providing feedback on a single message more than once. Details on this can be found in Ian’s report.

3.4 CoinShuffle
CoinShuffle is a decentralized coin mixing protocol originally designed for Bitcoin that prevents attackers from linking different transactions made by the same user with each other[3]. In AnonRep++, we use the CoinShuffle protocol to give clients a new set of addresses after each round. For example, suppose client uses a wallet $w$ to make a post, and after the feedback round, $w$ has four reputation coins in it. If the client used this wallet for subsequent posts, it would be known that any post that uses $w$ is the same user that originally used $w$. So, after a round of CoinShuffle, the client will have 4
wallets, each with one reputation coin in it, that the client can use to post future messages.

### 3.4.1 Hybrid Cryptosystem

Coinshuffle uses a hybrid cryptosystem to combine the advantages of public-key encryption and symmetric-key encryption. In particular, we use RSA with padding (RSA-OEAP) for public-key encryption/decryption and AES for symmetric-key encryption/decryption. The texts we are encrypting in CoinShuffle grow larger and larger in length, which is not suited for a public-key encryption scheme like RSA. However, AES can encrypt arbitrary amounts of data. The encryption for our cryptosystem works as such:

1. Securely-generate an AES key.
2. Encrypt the text with AES.
3. Encrypt the AES key with RSA.
4. The cipher text is the encrypted AES key, the nonce of the AES encryption, the tag of the AES encryption, and the AES-encrypted text, all appended with each other.

The decryption works in the reverse manner:

1. Extract the encrypted AES key, nonce, tag, and AES-encrypted ciphertext from the ciphertext.
2. Decrypt the encrypted AES key with RSA.
3. Use the AES key, nonce, and tag to decrypt the ciphertext.

### 4 System Design

#### 4.1 AnonRep

##### 4.1.1 Server Registration

A new server generates a public/private key pair and registers with the coordinator by sending its host/port and its public key. The server also generates an ephemeral key (and will generate a new ephemeral key each round). Details on how the keys are generated can be found in Ian’s report.
4.1.2 Client Registration

A new client generates a public/private key pair and sends its public key (long-term pseudonym) to one of the servers. The server then initializes the reputation score (0) for this client and all the servers sequentially encrypt the reputation score. The long-term pseudonym and its encrypted reputation score are then broadcast to every server. Details on the key generation and encryption can be found in Ian’s report.

4.1.3 Announcement Phase

Before the announcement phase, we know long-term pseudonyms and their encrypted reputations. The announcement phase uses the Mix-Net protocol to transform this into short-term pseudonyms (that change in every round) and their decrypted reputation. The details of how this works can be found in Ian’s report.

4.1.4 Message Posting Phase

A client can post messages using her one-time pseudonym. She also signs the messages with her private key to prove that she is the owner of the one-time pseudonym. Details about the determination of one-time pseudonyms and the signature scheme can be found in Ian’s report.

4.1.5 Feedback Phase

A client submits feedback using a linkable ring signature, whose details can be found in Ian’s report.

4.2 AnonRep++

4.2.1 Blockchain and Tokenization

A shortcoming of AnonRep is that it is prone to intersection attacks. A user with a unique reputation score, for example, one that is much higher than all the other users, has little anonymity. This is because other users will be able to link any message posted by a user with that high of a reputation to her.

Our proposed solution to this vulnerability is to tokenize reputation into a “reputation coin” on an Ethereum blockchain. One of the benefits of putting
reputation on the blockchain is that any record of changes of reputation is public and essentially permanent, allowing anyone to audit the legitimacy of the system. In addition, the smart contract is public, allowing anyone to audit its code.

Users will have their reputation in different “wallets” that they use to post messages and can use any amount of these wallets to post messages. Due to the mechanics of CoinShuffle, each wallet will have at most one reputation coin at the beginning of a round. Users can only have an integer amount of reputation coin. If another user votes up a message, one of the wallets used to post the message will gain a reputation coin. If the message is downvoted, one reputation coin will be removed from one of the wallets, unless there is no more reputation left in the wallets. A wallet cannot hold a negative amount of reputation coin, so a user’s total reputation is non-negative.

The implementation of tokenization is a smart contract written in the Solidity programming language which is deployed to an Ethereum blockchain. For benchmarking, the blockchain provider we use is Ganache\textsuperscript{3}, a local Ethereum blockchain that is suitable for testing purposes. The smart contract allows an admin (the coordinator) to add or subtract some amount of reputation coin from a wallet and also transfer one reputation coin from one wallet to another wallet for the CoinShuffle step. The smart contract also allows anyone to see how much reputation a given wallet has, and see the log of changes in reputation for all wallets. This allows anyone to audit the legitimacy of the system and make sure the coordinator is not behaving maliciously. The Python code interfaces with the blockchain using a library named web3\textsuperscript{4}.

4.2.2 Message Posting Phase Changes

In the original AnonRep system, clients proved that they had a certain amount of reputation by proving that they owned the long-term pseudonym that had that amount of reputation associated with it. In this new system, on the other hand, clients prove that they own enough wallets to have that amount of reputation. Concretely, this means that when users make a post, they also supply \(x\) wallets with a total of \(r\) reputation. To prove ownership of these wallets, they provide the public key of each wallet along with a message signed by the private key of the wallet that proves they own the wallet.

\textsuperscript{3}https://github.com/trufflesuite/ganache-cli
\textsuperscript{4}https://github.com/ethereum/web3.py
As mentioned before, due to the mechanics of CoinShuffle, every wallet will have either 0 or 1 reputation coin at the beginning of every round. If a client wants to post a message with 0 reputation, she can either use an existing wallet she owns that has 0 reputation (not recommended because of linkability concerns) or generate a new public/private key pair, and supply that wallet. If a client wants to post a message with \( x > 0 \) reputation, she must provide \( x \) wallets, each with one reputation. The client implementation in `client_blockchain.py` takes care of these details for the user.

The clients also need to start a CoinShuffle server and supply the host/port pair of the server when sending the message so that the coordinator can coordinate the CoinShuffle step. We assume that the clients will have some way to anonymize the host/port, perhaps by using an anonymous network like Tor.

### 4.2.3 Feedback Phase Changes

Since reputation now on the blockchain, the coordinator will keep track of the votes made as the feedback phase progresses. After the phase ends, the coordinator will apply the changes in reputation to the blockchain.

### 4.2.4 CoinShuffle

In this implementation, we use an abridged version of the CoinShuffle detailed in [3], leaving out the blame phase. The step of our version of CoinShuffle are as follows:

1. The coordinator tells all the clients that participated in the message phase whether they are participating in CoinShuffle or not. A client will be a participant in a CoinShuffle round if and only if at least one of the wallets she provided in the message phase ends up with \( > 0 \) reputation after the feedback phase.

2. The clients who are not participating shut down their servers. The clients that are participating generate a RSA public encryption key/private decryption key pair and send the public key to the coordinator. Each participating client with \( x_i \) reputation at the end of the feedback phase also generates \( x_i \) wallet public/private key pairs. These \( x_i \) keypairs will be the client’s new wallets that she can use for subsequent rounds.
3. The coordinator then gives each client the next address to send keys to, and an ordered list of encryption keys to use.

4. Suppose the participating clients are $a$, $b$, and $c$, in that order. The coordinator will tell $a$ to begin, then $a$ will encrypt all of her new wallet public keys with $c$’s encryption key, then $b$’s, puts these ciphertexts into a list, shuffles the list, then passes the list to $b$. Then $b$ takes this list, decrypts each item with her decryption key, adds all of her new wallet public keys encrypted with $c$’s encryption key to the list, shuffles the list, then passes the list to $c$. Finally, $c$ decrypts each item with her decryption key, adds all of her new wallet public keys, shuffles the list, then passes the list to the coordinator. The result is a shuffled list of all the new wallet public keys, encrypted and decrypted in a way such that none of the clients can discern which one of the other keypairs belongs to which client.

5. Finally, the coordinator gets the shuffled list of wallet public keys. The number of public keys should be the same as the reputation of all wallets used to post messages in this round. The coordinator performs transactions on the blockchain to transfer reputation from these wallets to the shuffled list, so that each public wallet key has 1 reputation coin.

5 Evaluation

5.1 Technical Specifications

To benchmark AnonRep++, we performed experiments on Yale’s Zoo machines. We ran the coordinator server was run on ladybug, four servers on tiger, turtle, aphid, and dolphin, and clients on swan. Since we didn’t test under conditions where clients perform actions simultaneously and interactions directly between clients are minimal by the nature of the system, we felt having all the clients be on the same machine is both fair and convenient. Each Zoo machine runs on Fedora 26 with 20 Intel Xeon(R) CPU E5-2650 v3 @ 2.30GHz cores and 62.8 GB of RAM. A network bandwidth of 76 Mbps was used.
5.2 Benchmarking Results

We benchmarked the following:

5.2.1 Announcement and Reverse Announcement Benchmarking (AnonRep-specific)

We measured how long announcement phase and reversed announcement phase takes for differing number of clients. The amount of time scales linearly with the number of clients, as expected.

5.2.2 Linkable Ring Signature Sign and Verify Benchmarking

We measured how long it takes to create a linkable ring signature and verify a linkable ring signature. The time scales quadratically with the number of clients due to lack of optimization. We are unconcerned about this because this is not the main part of the project. In addition, there exists a fix to make it scale linearly and it is shown in [1] that these operations can scale linearly.

5.2.3 Verifiable Shuffle Benchmarking

We measured how long verifiable shuffle takes to complete. The time scales linearly with the number of clients, as expected.

5.2.4 CoinShuffle Benchmarking

The time does not scale quite as well here, but the performance as number of clients increases seems to be on par with the measurements done in the CoinShuffle paper [3].

5.3 Comparison

The main differences between AnonRep and AnonRep++ are that AnonRep doesn’t include the announcement/reverse announcement phase or verifiable shuffle but does include CoinShuffle. CoinShuffle scales worse than these primitives, but if we keep a fixed maximum ring size for CoinShuffle and run multiple CoinShuffles in parallel, the cost of CoinShuffle is nearly constant. This is detailed more in the Discussion section.
Figure 2: Benchmarking results. We use a logarithmic scale for the axes, except for the CoinShuffle benchmarking figure.
6 Discussion

In this section, we discuss improvements of AnonRep++ over AnonRep, limitations of AnonRep++, possible solutions to these limitations, and future directions.

AnonRep++ protects against intersection attacks by allowing users to post with less reputation than they actually have. In AnonRep, a user with uniquely very high reputation would be easily identified when she posts by virtue of having such high reputation. In AnonRep++, this user can post with less reputation than her true reputation; the only thing that other users would know is that she has at least the amount of reputation that she posted her message with. Since the minimum amount of reputation a user can have is 0, users can post with 0 reputation for maximum anonymity. AnonRep++ also protects against linkability. In AnonRep, two clients with the same reputation will have the same encrypted reputation. At the very least, this results in a user being able to discern which users have the same amount of reputation as her. AnonRep++ doesn’t have this problem since we don’t encrypt reputations; rather, we store them on the blockchain. Finally, AnonRep++ eliminates the need to do an announcement phase. As a result, AnonRep++ does not need the cryptographic primitives that are unique to the announcement phase, such as verifiable shuffles.

One disadvantage of AnonRep++ is that users can effectively give other users reputation by giving them their public/private key wallet pairs. This is an inherent risk of blockchain-related solutions.

Another disadvantage of AnonRep++ is that our CoinShuffle phase does not scale very well (see Figure 2 and [3]). To combat this, we can replace CoinShuffle with faster alternatives, like [4]. We chose to implement the original CoinShuffle system because it was simpler to implement and still gave us protection from intersection attacks, as desired. We can also perform multiple CoinShuffles in parallel, each CoinShuffle ring consisting of up to a fixed number of participants, instead of one large CoinShuffle ring. This would prevent the time taken in the CoinShuffle step from getting out of hand. There are also minor optimizations to CoinShuffle that can be done. For example, since only the coordinator has write-access to the smart contract, the coordinator can cache the reputations to avoid making read requests to the smart contract.

Our LRS implementation scales quadratically, but an improvement to the hashing mechanism would allow LRS to scale linearly. In particular, since
we use the SHA-1 hashing function in LRS, which operates block-by-block, we can make SHA-1 a constant-time operation instead of one that scales linearly by only recomputing the blocks that we change, which is the last block. [1] already shows that LRS can scale linearly, so we chose to not make these improvements. In addition, we can limit the size of the LRS ring as suggested with CoinShuffle. For example, we can have each user belong to an LRS ring that is no bigger than $x$ participants, instead of having an LRS ring composed of every single registered client.

We can go even further with putting aspects of AnonRep on the blockchain. For example, the entire message board can also live on the blockchain. This would make it even easier for clients to audit the system for malicious behavior, and also provide the benefit of uncensorability; if the message board lives on the blockchain, the message cannot be erased or altered unless the attacker was able to take control of the blockchain. We did not put the message board on the blockchain because our main idea in this project was to eliminate the possibility of intersection attacks by putting reputation on the blockchain. But putting the whole message board on the blockchain would improve auditability and permanence of messages.

Our implementation can be made more secure by signing all messages sent over the network. We did not do this because it was not important for a proof-of-concept implementation.

7 Conclusion

We have built the first anonymous reputation system that is resistant to intersection attacks, AnonRep++. We use cryptographic techniques like linkable ring signatures to preserve anonymity and use smart contracts on the Ethereum blockchain to provide resistance to intersection attacks. We have also shown that AnonRep++ can, with some minor optimizations, can be theoretically deployed at scale.

8 Personal Contributions

In this group project, I mostly worked on the parts of the system that were unique to AnonRep++, including the blockchain layer, the integration of the blockchain layer with the Python code, CoinShuffle, and the cryptographic
primitives associated with CoinShuffle. I also spearheaded the benchmarking of AnonRep and AnonRep++.  

9 Acknowledgements

We would like to thank Ennan Zhai for giving us the inspiration to do this project and for providing answers to our questions, feedback, and guidance throughout the semester.

References


