University of Edinburgh
School of Informatics

Development of a Bitcoin mixing service using Secure Multi-party Computation techniques

4th Year Project Report
Artificial Intelligence and Philosophy

Nicholas La Rooy & Dr. Julian Bradfield

March 28, 2013

Abstract: Bitcoin uses a public proof-of-work scheme to prevent double-spend attacks in a decentralized manner. However, it is this use of public processing that makes Bitcoin transactions merely pseudonymous. This report documents the design and implementation of a decentralized mixing service: a service in which multiple participants input coins, which are then returned to them in a less traceable manner. Unlike existing implementations, the one presented here makes use of Secure Multi-party Computation, minimizing the trust a user must place in both the other participants and the service infrastructure.

Acknowledgements

I would like to thank Dr. Julian Bradfield, who took the gamble of supervising this project and has guided its progress throughout; Mike Hearn, for answering my questions regarding BitcoinJ; Piotr "ThePiachu" Piasecki for his Bitcoin faucet, which was invaluable during development; and finally, my parents for supporting me during my studies.

Contents

1. Introduction

Bitcoin is a digital currency with two particularly striking features. First, is its rapid adoption rate following its release by Nakamoto in 2008 [17], and second is its decentralized nature. In Bitcoin there is no central authority, such as a central bank, issuing the currency or clearing accounts. A public-private-key infrastructure provides proof of ownership, whilst a public peer-to-peer network prevents double-spend attacks. Since Bitcoin depends on having a wide public network to protect itself, Bitcoin's transaction history is public record. This makes Bitcoin merely pseudonymous; should a Bitcoin address be linked to its owner's real world identity, that owner's spending may be easily tracked.

1.1 Aims

The aim of this project is to develop a Bitcoin mixing service. A mixing service allows the transfer of Bitcoins between two addresses (essentially accounts), without creating a clear link between the two. The implementation for this project should make use of Secure Multi-party Computation to minimize the level of trust required for the service to operate successfully.

Since this is a new application of Secure Multi-party Computation, the system here aims to serve as a proof-of-concept only. A full service would exist as a peer-to-peer network over the internet with automated mechanisms for discovering other participants and black-listing malicious ones. This is discussed in Future Improvements.

2. Background

2.1 Bitcoin

2.1.1 Timestamp Servers and Double-spend Attacks

Bitcoin's peer-to-peer network – the "Bitcoin Network" – can be more readily explained when it is thought of as a specialized timestamp server [17]. A timestamp server cryptographically binds any data it receives with the current time using its private key and broadcasts it widely. Anyone can check this binding using the server's widely published public key. Thus, assuming faith in the server and the cryptographic scheme in play, one can be assured that any data from the server existed at or before the timestamp specified.

Such a server can protect against double-spend attacks. Suppose that Alice and Bob are both merchants. Mallory has 10 Digital Coins, and wishes to purchase some of Alice's products. She signs a transaction of 10 coins to Alice, in exchange for 10 coins worth of goods. However, Mallory performs the same transaction with Bob, signing over the same coins. At the end of the day when Alice and Bob send their signed transactions to the Digital Coin Bank, they find that there are only 10 Digital Coins to go between them, whilst Mallory has 20 coins worth of goods. This is a double-spend attack.

Now suppose that Alice and Bob have access to a timestamp server. They trust the server and keep a log of its output. Whenever they receive a transaction, they first check that the funds it sends are consistent with their logs. After accepting a transaction, they send a copy to the timestamp server. Now when Mallory attempts her double-spend attack Bob is not so easily fooled. He checks the logs from the timestamp server and finds that the coins being offered have already been sent to Alice.

Of course, such a solution depends on everyone submitting their transactions to the timestamp server; this toy example was chosen to illustrate the purpose of the Bitcoin Network.

2.1.2 The Bitcoin Network

The Bitcoin Network can be seen as a type of decentralized timestamp server, whose purpose is to order and verify transactions. Transactions are propagated through the network, with each node collating them into a "block". A block has a

header consisting of a hash of the previous block, thus forming a "blockchain". In this form, a block is not yet complete. Each node repeatedly tweaks the block's header, trying to make its hash less than or equal to a particular threshold. Assuming a good hash function is used, this process cannot be performed faster than a random search. Once an appropriate nonce has been found, the node sends its solved block to the network. The longest valid blockchain is considered the correct transaction history. This process is known as "mining".

How does such a scheme work? In the timestamp server example above, the prevention of double-spend attacks was a result of:

1. Alice and Bob having access to the timestamp server's records
2. Alice and Bob rejecting transactions that conflict with those records
3. All transactions being sent to the timestamp server

Bitcoin satisfies the first point by allowing Alice and Bob to be nodes on the network. Bitcoin's blockchain is easily accessible by any node, and the network is open for any client to join. Mirrors are also available on the web [10].

The second point is satisfied by a coalition of nodes. In Bitcoin, transactions that are not consistent with the current longest blockchain are discarded. A node could choose to include them, but then any blockchain containing them would not be accepted by the rest of the network. Bitcoin works on the assumption that no adversary controls 50 percent of the hashing power of the network. Under such circumstances no inconsistent blockchain will be accepted and extended.

The third point is impractical to achieve directly. Instead, the merchant waits for the transaction to be buried several blocks deep in the blockchain before delivering any goods. This is known as waiting for confirmations. If the merchant waits for one confirmation, that is for the transaction to be buried one block deep, then an adversary needs to be able to mine two blocks redacting the transaction before the network can mine just one. If the network's hashing power is strong, then the adversary's chances are slim. Waiting for more confirmations increases security against such attacks, albeit at a time cost.

Mining also acts as a distribution mechanism, granting some Bitcoins to whoever successfully mines a block. This reward is set to 25 BTC at the time of writing, but is programmed to halve every 210,000 blocks [7], giving Bitcoin distribution characteristics similar to that of gold.

2.1.3 Transactions and Anonymity

Bitcoin's decentralized approach comes at a cost. Since the security against double-spend attacks requires a large, open network, all transactions must be

widely publicized. This means that the flow of funds between addresses can be tracked by any node in the network.

Nakamoto provides a partial solution [17]. There is a distinction to be made between an address, that is a hash of a public key, and a user. Since new addresses can be generated at effectively zero cost, and transactions are between addresses, not users, Bitcoin remains pseudonymous. Indeed, it is considered good practice to generate a new key-pair for each transaction [7].

Transactions in Bitcoin are complex but flexible. A transaction consists of a number of inputs and outputs. An output is a declaration of the conditions under which funds may be claimed, whilst an input is a claim on a previous transaction's output. This notion of claim is handled by a stack-based, Turing-*incomplete* scripting language. Each input and output contains a short piece of script. An input may claim an output only when the output's script appended to the input's script evaluates true. This shall be discussed further subsequently; for now the important point is that each transaction has a number of inputs and outputs.

Reid and Harrigan draw a distinction between the transaction network and the user network [19]. The transaction network is constructed as follows; for each transaction create a new vertex; add directed edges from each vertex to the vertices it references as output. The transaction network is a directed acyclic graph. The user network begins with each address as a vertex with directed edges added as in the transaction network. Two vertices in the user network may in fact refer to the same person; we merge them when we have reason to think that this is indeed the case. The more this operation is performed, the better our knowledge of the flow of funds between users becomes.

Nakamoto gives a circumstance under which this operation may be performed [17]. In a multi-input transaction, the owner of the inputs is most likely the same user, since only one user should know an address's private key.

Suppose a merchant wishes to conceal who they do business with. As per best practice, they create a new address for each transaction with each customer. This leaves the merchant with funds spread across many addresses. Needing to pay a large bill, the merchant takes these distributed funds and uses them as inputs for a large transaction. But now, through proof of ownership of private keys, the merchant has conflated the user-identities of its addresses. Since each customer knows that the address they paid belongs to the merchant, and that each input address in the transaction probably also belongs to the merchant, they can infer that the addresses paying into them probably belong to customers.

A similar situation may occur when a user claims change from a transaction. A Bitcoin transaction's outputs are taken from the funds it claims as input; any remaining Bitcoins are given to miners as a transaction fee. Suppose that Alice

has an unspent output worth 50 BTC and wishes to send 30 BTC to Bob. She constructs a transaction claiming that 50 BTC and adds two outputs: one to Bob, worth 30 BTC, and another to a second address that she controls, worth 20 BTC. After the transaction has been processed, the original 50 BTC output has been spent, but now Bob has an extra 30 BTC and Alice has 20 BTC in the second address. To an outsider viewing this transaction there are two possible explanations. Either Alice needed to pay two parties exactly 30 BTC and 20 BTC, or Alice controls the second address and used it to receive change. The second explanation is more likely.

In such scenarios, the problem of retaining Bitcoin's pseudonymity is equivalent to the problem of transferring funds to a new account in a manner that conceals their origin.

2.1.3.1 Mixing Services

A mixing service takes a participant's funds and returns them in a manner that is not easily traced.

A centralized solution could work as follows. The mixing service widely disseminates a public key for encrypted communication. Users securely send their current Bitcoin addresses and a new address they have just generated. The mixer keeps record of this and returns a confirmation message along with its Bitcoin address. The customer now sends funds to that address. The mixer knows where to send these funds onto from its prior communication with the customer. After a number of confirmations, the mixer sends the funds to the new address, minus some fee. Funds can be sent in a number of transactions of small randomized amounts to mitigate traffic analysis.

Such services already exist [3, 5], but none have risen to prominence. The challenges such services face are twofold. First, the user must place a high level of trust in the mixing service itself; the service may 'cut-and-run' with the user's funds; it may also be leaking, or even selling, message transcripts. Second, such services require a high volume of users in order to be successful. If only one user is sending coins to a mixing service, then chances are the mixer will only be sending coins to that user.

There has been some discussion in the Bitcoin community about using Secure Multi-party Computation to tackle the mixing problem. An article describing one approach [25] was released independently of the work here.

2.1.4 Bitcoin Clients

Since Bitcoin clients only communicate via IP, a client need not be implemented with any particular underlying technology. The original client, first developed by Nakamoto, is written in C++ [6]. Later clients have been written in Java (BitcoinJ [4]) and Python (Electrum [13]).

Bitcoin clients may be categorized as either a "full client" or a "light client". A full client has the complete transaction history of the Bitcoin network, which it uses for verifying blocks and so on. This is typically downloaded from other nodes on start-up, which is already proving a scalability issue for Bitcoin. The original Bitcoin client is an example of a full client. A light client only requires the headers of the blockchain and the transactions relevant to the user of the program. This can be done since all transactions in a block are part of a Merkle Tree whose root hash is in the block header. Light clients allow for faster start-up times and a lower memory footprint, but they rely on the rest of the network to ensure that the blockchain is consistent.

2.2 Secure Multi-party Computation

Secure Multi-party Computation allows a set of parties each with a private input to compute some function of those inputs. The Secure Computation should provide each party with no further information about the inputs than what is given by a correct output. The applications of the field are numerous, ranging from electronic voting to secure auctions to checking no-fly lists.

Informally speaking, a secure computation protocol must guarantee:

- Correctness of output. The protocol must follow the given function precisely.
- Privacy of inputs. No information about each user's input may leak, beyond what is learned from a correct output.
- Independence of inputs. No party should be able to make their input a function of another's. Failing this would prove particularly catastrophic to a secure auction, for example.

Formally speaking, security is defined through simulation [11]. Suppose that there are two worlds: the "ideal world" and the "real world". In the ideal world, participants have access to a trusted third party. This third party is incorruptible, follows some publicly published algorithm and can be communicated with by each participant on their own secure channel. In the real world, no such third party is available, and so it is replaced by a Secure Multi-party Computation construction.

A construction is secure when it provides the same guarantees as the trusted party in the ideal world, against adversaries bounded by some adversary model.

2.2.1 Adversary Models

A real world construction's security will be bounded by some adversary model. The broad categories of these are given here:

1. Passive or Active. A passive, or curious, adversary is one that logs the messages sent from and received by corrupted players but otherwise follows the protocol. An active, or dishonest, adversary is one that may cause a corrupted player to deviate from the protocol.
2. Static or Adaptive. A static adversary may corrupt a set of players from the start of the protocol, but this set may not change for its duration. An adaptive adversary may corrupt any player at any moment.
3. Computationally Bounded or Unbounded. Encryption schemes based on the (supposed) hardness of certain problems can protect against a computationally bounded adversary; against a computationally unbounded adversary they will not. Solutions that are secure against unbounded adversaries are called information-theoretically secure. Such security is desirable but can be difficult to achieve.

An adversary may have a combination of these properties. For example, an unbounded passive adversary would follow the protocol exactly, but use its unlimited computational power to try and gather information from the messages it receives.

2.2.2 Secure Multi-party Computation Constructions

Secure Multi-party Computation can be achieved in many ways and under a number of different adversary models. Since an existing Secure Computation library will be used, only a brief summary of the major constructions is given here.

2.2.2.1 Yao's Garbled Circuits

The original motivation for Secure Multi-party Computation is given by Yao's Two Millionaires Problem [26] (two millionaires wish to find out who is the richer, but without revealing any further information about their wealth). Yao introduces both a protocol for solving the problem and a generalized construction for

two party Secure Computation. Yao's construction is secure against a static, computationally bounded, honest-but-curious adversary. The function computed must be represented as a boolean circuit.

2.2.2.2 GMW

GMW (Oded Goldreich, Silvio Micali and Avi Wigderson) allows two or more parties to securely evaluate an arithmetic and multiplication circuit [15]. The numbers in the circuit are restricted to a finite field. GMW is secure against a static, dishonest computationally bounded adversary controlling less the half the number of parties.

2.2.2.3 BGW

BGW (Ben-Or, Goldwasser and Wigderson) is an extension of GMW and also operates across an arithmetic and multiplication circuit [2]. BGW, however, is information theoretically secure against an adaptive and dishonest adversary, given a 2/3 majority of uncorrupted players. For honest-but-curious adversaries, the protocol is secure for an honest majority. A slight complication is that BGW requires secure channels between parties, so an actual implementation may only be conditionally secure.

2.2.3 Feasibility

When secure computation was first discussed in the 1980s it was practically infeasible. 2008 saw the first large scale application in which researchers facilitated an auction between sugar-beet farmers [12]. The actual secure computation was done between three laptops, each operated by competing entities, and communicating via a LAN. The procedure took only around 30 minutes to evaluate 1229 bids.

2.2.4 Determinism

The functions in question are deterministic. That is, assuming the computation runs successfully, the outputs are entirely determined by the inputs. Should any entropy be required in the output, each player must extend their input to include random bits as required. For efficiency, these bits may simply be used as a seed value for a pseudo-random number generator.

The obvious concern is that when the players are responsible for inputting the random bits, a malicious player may be able to manipulate the output to their own advantage. This can be prevented by using the result of successive XOR operations on the bits rather than just the bits themselves. To see how this works, suppose that an adversary controls all but one input bit. Call the result of successive XOR operations on the manipulated bits b. Now the remaining random bit r has value 1 or 0 with uniform probability, so the result of $rXORb$ also has value 1 or 0 with uniform probability.

3. Project Specification

3.1 Problem Analysis

There are multiple users each with funds in an address that may be linked to their real world identity. Each user wishes to transfer these funds to a fresh address that cannot be traced back to them, allowing the funds to be spent anonymously. The goal is to build a system that facilitates this whilst minimizing the level of trust that must be placed in both the other users and the system itself.

The problem can be distilled further to that of creating a particular kind of transaction. A transaction with each traceable address as an input and each fresh address as an output, all receiving some uniform amount, can provide the required level of anonymity. The amount must be uniform or adversaries will be able to link traceable and fresh addresses by it.

In such a transaction of n inputs, the probability of each input address belonging to the same user as any given output address is just $1/n$. Traceability can be reduced by taking the output of a mixing transaction and using it as the input for another. Such transactions break the assumption that all inputs in a transaction are controlled by one user, an important step in Reid and Harrigan's analysis [19]. The challenge is that each participant must add their fresh address as an output without revealing which is theirs.

3.1.1 Requirements

There is a hierarchy of requirements that the system must meet. Security is defined as adherence to these requirements. These are given in descending order of importance:

1. Lost Coins. Bitcoins should never be lost to a participant, even if all other parties are corrupted.

2. Anonymity. No information about the traceable-fresh address pair should be leaked, beyond that which can be deduced from the output transaction.

3. Denial-of-Service. The potential for a participant to perform a denial-of-service attack is minimized.

3.1.2 Restrictions

So far it has been assumed that each participant is a different person. Under this assumption, it is reasonable to expect that one participant is not being colluded against by the rest; an adversary would have to compromise $n - 1$ systems to achieve this. If the mix occurs across a WAN however, then this claim is more difficult to accept; an adversary may run multiple instances of the client, thereby increasing the chance of being an overwhelming majority in a mix.

Since this implementation is a proof-of-concept, the system will be designed for the less hazardous LAN scenario. To simplify the problem further, the system will be controlled via a command-line interface and participants will be able to agree on any necessary shared parameters outside of the system. It is also worth noting how targeting only a LAN lessens the importance of meeting requirement 3, since participants can resolve denial-of-service issues in person.

3.2 Design

3.2.1 Ideal World Solution

Under ideal world circumstances, the transaction can be generated by the following procedure:

1. The participants each ensure that there is the required number of Bitcoins in their traceable address.
2. They each send the public-private key-pair of their traceable address to the trusted party. Note that each address can be constructed from its respective key-pair.
3. The trusted party confirms the validity of the key-pairs and that they control an appropriate address.
4. Having received all of the key-pairs, the trusted party generates a fresh addresses for each participant. This is done using a good source of entropy.
5. The trusted party constructs a transaction with each traceable address as input and giving a uniform amount of Bitcoin to each generated address.
6. The transaction is signed using the submitted private keys.
7. Each participant receives as output the signed transaction and one of the fresh addresses' private keys. The private key received is randomly chosen, and no two participants receive the same key.

8. Any participant may now submit the transaction to the network.

3.2.1.1 Transaction Structure

As mentioned in Section 2.1.3, Bitcoin script is a simple stack-based programming language. Elements in the script are processed from left to right, with constants pushed to the stack and operations manipulating that stack. There operations are numerous, allowing for highly flexible transaction types, but most are currently disabled for security reasons. Transactions consists of a number of inputs and outputs. Each input contains a script for claiming a previous output, and each output contains a script declaring the conditions under which it may be claimed. The scripts used for the mixing transaction are crucial to the security of the system. They must ensure the following:

1. Each target address must receive the same amount of BTC.
2. Each source address must input at least the same amount of BTC.
3. The transaction is only valid when all input signatures are present. A transaction must be valid for it to claim outputs and provide inputs.
4. Modifying the inputs and outputs of the transaction invalidates existing transaction signatures.

```
Transaction:

    // For each source address i:
    Input:

        Previous tx:    <a standard tx with an output >= b which
                        may be claimed by i>
        Index:          <an appropriate index from the previous tx>
        scriptSig:      <signature from i> <public key of i> SIGHASH_ALL


    // For each target address j:
    Output:

        Value:          X BTC
        scriptPubKey:   OP_DUP OP_HASH160 <hash of j's public key>
                        OP_EQUALVERIFY OP_CHECKSIG
```

Figure 3.1: Transaction Structure

Figure 3.1 describes a standard pay-to-address transaction, which satisifes these properties. In order to claim an output, the public key of the payee address and a signature from it must be provided. The scriptPubKey is then appended to this, creating a single script for processing. The processing is quite straightforward; the Bitcoin client checks that the hash of the given public key matches that of the payee address and that the signature is correct.

The SIGHASH_ALL constant is used by OP_CHECKSIG. It indicates that the hash of all of the transaction's outputs are signed. This means that any modification of the outputs invalidates existing signatures, ensuring that a player always knows a transaction's outputs before signing.

3.2.1.2 Security

Security was defined as a hierarchy of objectives; here this design is informally assessed against them.

Lost Coins Unauthorized transactions cannot be made under this procedure since the private keys are shared only between their proper owner and the trusted party across a secure channel. The trusted party ensures that each participant receives the correct number of coins from the final transaction, since it necessarily follows the protocol.

Anonymity Fresh addresses are generated by the trusted party and the private key distributed to each participant is randomly selected. Hence no participant may (individually) deduce who controls which output in the final transaction. Of course, should a group of participants be corrupted, then some fresh addresses may be eliminated from consideration. If all but one participant is corrupted by an adversary, then anonymity is lost. This is the most that can be offered by this type of transaction.

Denial-of-Service A participant may only delay the mixing by not submitting a valid key-pair. To mitigate this, the solution can be modified to add a deadline within which a valid key-pair must be submitted. If a participant fails to meet the deadline, they are removed from the mix.

Another minor issue is due to a race hazard in the system. The Bitcoin network will only accept the output transaction if its inputs have not already been claimed. A malicious participant might submit an address with enough funds at the start of the procedure, but then spend them before the output transaction can be submitted to the network. One way to prevent this is for the trusted party to

transfer funds into a holding address (and wait for a number of confirmations) as part of the validation step.

These concerns are severe only for an implementation over a WAN, and so should be seen as scope for future improvements.

3.2.2 Revised Ideal World Solution

Since Bitcoin's correct operation is assumed, some aspects of the system can be moved outside of the trusted third party. This will yield a number of benefits when the trusted party is replaced by a Secure Multi-party Computation construction. The revised system is as follows:

1. Each participant generates a new address to be their fresh address. (This must be done for each mixing session)
2. The fresh addresses are sent to the trusted third party.
3. The trusted party shuffles the list of fresh addresses.
4. Each participant is sent a copy of the shuffled list.
5. All participants send their traceable addresses to the first participant.
6. The first participant generates a transaction from the traceable addresses to the each of the fresh addresses in the received list.
7. The transaction is sent to the other participants.
8. The other participants verify that the transaction has an output for each address in the received list, that their fresh address is one of them, that the amount is correct and that their traceable address is listed as an input. If these checks pass, a signature is sent back to the first participant.
9. The first participant adds the signatures to the transaction and submits it to the network.

3.2.2.1 Security

Lost Coins In this design, this is ensured by the verification step. Since a signature is only returned if the proposed transaction is agreeable to user, meaning that they receive the appropriate number of coins, no funds can be lost from a malicious party removing someone's fresh address from the pool. The transaction cannot be tampered with, since this would invalidate the signatures.

Anonymity No fresh addresses are revealed in transit to the trusted party since they are sent over a secure channel. The output list is guaranteed to be properly shuffled by the incorruptible trusted party.

The verification and signing steps are not dependent on the participant's particular fresh address, and so leak no more information than the signing procedure of before, even if the first party were to send an incorrect transaction.

Denial-of-Service Should a malicious party delay the sending of a fresh address, or the signing of the transaction at the end, then the entire procedure will halt. The first participant is given substantial control over submitting the final transaction, since they are the only party with all of the signatures. This could be resolved by having each participant try to create the transaction, but this would be of little benefit since anyone may still refuse to sign. This is definitely a weakness of the design.

3.2.2.2 Shuffling by Sorting

Rather unintuitively, the trusted party can use either a shuffling algorithm or a sorting algorithm on the fresh address list; the result will still be a random permutation from each participant's perspective.

This is a consequence of how addresses are generated in Bitcoin. A Bitcoin address is the public portion of an ECDSA key-pair with various hashing operations applied to it [7]. Assuming the key-pair is generated from a good source of entropy, and the hashing algorithms have good distribution, then the address can be considered a random bit-string.

To each participant, a securely sorted list of random bit-strings has a random order.

3.2.3 Adversary Model

Of course, the ideal world is named as such for a reason. An actual Secure Multiparty Computation construction is only secure against an adversary bounded by some adversary model. For this application, the adversary must sit somewhere between that which Bitcoin is secure against, and that which is acceptable to the user. The model need not exceed Bitcoin's, since if such an adversary existed it is unlikely Bitcoin could continue to function at all. Each adversary attribute is considered in turn:

1. Passive or Active. It is reasonable to expect an adversary that may deviate from the protocol at any time. Even across a LAN, auditing each participant's code would be a difficult procedure. Hence an active adversary will be assumed.
2. Static or Adaptive. In the real world the adversary may exploit vulnerabilities on a user's machine at any time; an adaptive adversary should be expected.
3. Computationally Bounded or Unbounded. Bitcoin is designed to be secure against a computationally bounded adversary controlling under 50 percent of the network's hashing power. A computationally bounded adversary will be assumed for the mixing service.

The system will be designed to be resilient to an active, adaptive, computationally bounded adversary. Note that this does not necessarily mean that the Secure Computation construction chosen must be secure against such an adversary. Instead the goal is to maximize the security of the system as a whole, given such an adversary.

3.2.4 Real World Solution

In the real world, the design can be implemented in two parts. The first part is a secure construction for shuffling addresses, as in steps 2 and 4. The program enabling this will be implemented using a Secure Multi-party Computation library. Second is the transaction generation part for handling steps 1 and 5 to 9. This will be implemented using network and Bitcoin libraries.

If an adversary exceeds the model of the shuffling step, then it may manipulate the fresh address list. In the worst case, this results in either a loss of anonymity for each user or the system coming to a halt. Provided that the checks in the second step are implemented correctly, a user should never lose any coins.

4. Implementation

4.1 Technology Choices

4.1.1 Language

Java is highly portable, compatible with many existing libraries and a familiar language. For these reasons it will serve as the primary language for the implementation. Google's Guava library [16] will also be used to provide higher level abstractions, such as callbacks, in the Java code.

4.1.2 Network Library: Netty

The protocol requires a number of messages to be sent between the players. The Secure Multi-party Computation stages will be handled by a separate library, but the address, transaction and signature exchanges will have to be implemented another way. Netty [18] is a mature asynchronous network framework for Java. It has been chosen due to its high performance – it helps power searches on Twitter [14] – and familiarity.

4.1.3 Secure Multi-party Computation

In the interests of time and correctness, an existing Secure Multi-party Computation library will be used. There are two main contenders, VIFF and FairplayMP.

4.1.3.1 FairplayMP

FairplayMP is the successor to Fairplay, a two party Secure Computation platform released in 2004. FairplayMP has a number of desirable features. The function to be securely computed is defined using a high-level Secure Function Definition Language, which is then compiled into a type of garbled circuit. The underlying implementation is a fast variant of the BMR protocol [1], running in just 8 communication rounds. It is written in Java, allowing easy integration with many other libraries.

Unfortunately FairplayMP did not work for the test functions attempted.

4.1.3.2 VIFF, the Virtual Ideal Functionality Framework

VIFF [23], first developed for the Danish sugar-beet auction, is a mature implementation of a number of Secure Multi-party Computation constructions. Written in Python, it makes use of various language features to add a layer of abstraction between the underlying secure computation protocol and the function computed. In effect, VIFF overrides the addition, subtraction, multiplication, comparison and bitwise operations. This makes VIFF relatively easy to use, since the secure functions can be written mostly in pure Python.

The downside to using VIFF is interfacing between the Python and Java components of the system. A simple solution is just to using piping between them; a more complex one would be to send messages across ports.

4.1.4 Bitcoin Library

The protocol requires the handling of many Bitcoin constructs; addresses must be created and transactions must be generated and signed. To simplify these tasks, BitcoinJ will be used. BitcoinJ is an open-source Bitcoin framework in Java [4]. It is not a full client, but it does provide the required functionality. The prominent Python client, Electrum, would make a better choice in terms of integration with the Secure Computation components, but BitcoinJ is superior in terms of documentation, support and features.

During development it was discovered that BitcoinJ v0.7 only provides a way to sign every input of a transaction but not a specific one. This functionality is crucial to the system, since each player must sign only their contributing input. A patch to BitcoinJ was written to resolve this, which should be considered if the system is to be integrated with future versions of the library.

4.2 Address Representation

Since the input to the VIFF portion of the code must be numeric, and the address input is a 34 character string, a one-to-one mapping between address strings and numeric representations is required. This can be done using ASCII codes. Each character in the string is replaced with its ASCII number, padded with leading zeros to length three. The result is then taken as the numeric value. To transform back, convert the integer to a string representation, add leading zeros to make its length a multiple of three and replace each ASCII number with its appropriate character.

4.3 Shuffling

Shuffling can be done in $O(n)$ steps in the regular setting [8]. In the secure setting, inputs are secret shared values and so only a restricted set of methods can be used. A pair of elements can be randomly swapped obliviously using a "random-swap gate":

```
1: function RandomSwapGate(a, b, r)
2:     s ← ¬r
3:     x ← a · r + b · s
4:     y ← a · s + b · r
5:     return x, y
6: end function
```

The function returns the arguments a and b in an order determined by the bit r. If $r = 1$, then the result will be a, b. If $r = 0$, then the result will be b, a. Since the gate makes use of only addition and multiplication, if secret shared arguments are used, then VIFF will perform the operations securely. If r is unknown to the participants, then the resultant order will also be unknown. The function is data-oblivious.

Using only random-swap gates, it is possible to construct a random shuffle for lists whose length is a power of two. This is done by taking every possible pair of different list indexes and performing a random swap between those elements. Note that this pseudo-code below assumes a random-swap gate with its own source of random bits.

```
1: function Shuffle(list, randomSwapGate)
2:     l ← length(list)
3:     for i ∈ range(l) do
4:         for j ∈ range(i) + range(i + 1, l) do
5:             list[i], list[j] ← randomSwapGate(list[i], list[j])
6:         end for
7:     end for
8: end function
```

The procedure is slow, taking $O(n^2)$ steps to complete, but mixing sessions will likely take place between just a few participants. Correctness degrades when the list is not a power of two. In such cases, the probability cannot be uniform since it is impossible to involve each index in a uniform number of swaps.

4.3.1 Shuffling by Sorting

Earlier it was mentioned that for this application sorting performs the same work as shuffling. Secure sorting is a better studied problem than shuffling, since the restricted set of operations available makes it equivalent to implementing a hardware sorting network. These are built from compare-exchange gates, which are similar to the random-swap gates of before:

```
1: function CompareExchangeGate(a, b)
2:     c ← a ≥ b
3:     d ← ¬c
4:     x ← a · c + b · d
5:     y ← a · d + b · c
6:     return x, y
7: end function
```

The compare-exchange gate takes two numbers and returns them in descending order, making use of only the operations allowed in the secure setting. There are sorting networks using these gates that are $O((\log n)^2)$ in depth, such as bitonic sort [9], but they require an input that is a power of two. For the small input sizes expected, padding the list with dummy addresses would likely lead to a slower implementation.

Compare-exchange gates add a restriction on the VIFF runtime used. VIFF runtimes provide a layer of abstraction between operations and their Secure Computation constructions. In order to implement a compare-exchange gate, a comparison function is required. This is provided by the Toft05 and Toft07 VIFF runtimes [24]. Each is based on the BGW construction, offering security against a passive adversary under an honest majority. Toft05 provides comparison results in the finite field with modulus 256, whilst Toft07 provides results in the field of the numbers compared. This makes Toft07 more convenient when comparison results are used in subsequent computations and so it was selected over Toft05. Unfortunately, a direct sorting of the addresses is not possible since neither runtimes can handle the large numbers used for address representation.

4.3.2 Shuffling by Sorting a Representative

The problem with using a sorting algorithm is that VIFF's comparison module cannot handle such large numbers. However, since the sort is only used to obtain a random permutation, it is not necessary to compare the addresses. Instead, each participant can input a number between 0 and some large threshold. The addresses are then sorted by their corresponding number. The threshold should

be small enough for the comparison module, but large enough to avoid collisions, in which case the order will be deterministic. For n participants and a threshold of k, the probability of a collision is n/k^2, which is negligible for large k.

This method provides a true random shuffle, with high probability. It also requires that participants only secret share one number, as opposed to $O(n^2)$ random bits as in the shuffling. For these reasons this method was adopted.

The random number pool of the operating system was used as a source of random bits. In the implementation, each player takes 32 bits to form their shuffling representative, giving k a value of $2^{32} - 1$.

4.4 Protocol Coordination

Once the shuffling process has taken place, players must generate the mixing transaction and sign it. It would be possible to do this in a peer-to-peer manner, but to simplify the implementation a server-client model was chosen. The server is a Java application built using the Netty framework. It receives the shuffled list of fresh addresses from each player along with an appropriate transaction output to use as input. From these, a transaction is generated that is sent to each player for signing. The players check that the transaction conforms to their expectations, and if so send a signature in return. The host adds the signatures to the transaction and sends a copy each player once it has been completed.

If the host maliciously tampers with the transaction then it will be detected by the client code. The host can stall the mixing process, but the same can also be achieved by a player who refuses to sign. The use of a client-server architecture provides no security downsides but makes coordinating the system considerably easier.

Network messages are created using Java's built in object serialization. This has more overhead than a custom built message encoding scheme, but it makes for quicker development since each message type can be defined once, in a self-contained Java class.

4.5 Wallet Preparation Tool

As input to the mixing transaction, each player must find an unspent output to contribute. Each player will be receiving the same number of Bitcoins in return, so any Bitcoins contributed above the output value will go to miners as a transaction fee. To avoid this, a wallet preparation tool was developed. This

takes a BitcoinJ wallet file and a target amount as input. It searches through the existing unspent outputs until it finds one above the specified amount. It then takes this output and generates a new transaction from it with the specified value going to a new address and the remainder going back to the wallet as change. Running this tool before a mixing session ensures that Bitcoins are not lost as transaction fees.

5. Evaluation

5.1 Testing

Bitcoin provides two separate networks: the production network (ProdNet) and the test network (TestNet). Both operate in a similar way but on different block chains and with a different set of addresses, distinguished by a prefix used in their creation. ProdNet Bitcoins have significant monetary value, whilst TestNet Bitcoins are often given away for free. There is also less hashing power behind TestNet, giving mining a faster return. These factors make TestNet suitable for the development and testing of the system.

Testing was done using a machine running Ubuntu 12.04.1 LTS 64-bit with 7.8 GiB of memory and a 2.80GHz Intel Core i7 processor. TestNet coins were sent from a Bitcoin faucet [20] to three separate BitcoinJ wallets, which were each used by an instance of the client. To simulate real world network conditions a packet loss of 1% was introduced along with varying amounts of network delay. In each test the system terminated successfully with the mixing transaction stored in the wallets. The transactions were submitted to the network using a broadcaster tool to ensure their validity. The first mixing transaction was included in block number 54262 [21].

The times recorded were for the mixing transaction to be generated only. In a real use case, additional time must be added to allow the network to confirm the transaction. Since the first two participants to launch must wait for the third, only times from the third participant were taken. The result for zero simulated latency had an average ping of 0.019ms in practice.

5.1.1 Predictions

The system can be seen as two connected parts running in serial: the shuffling step and the transaction construction step. The shuffling step is where the Secure Computation takes place and requires $n^2 - n$ compare-exchange gates. In terms of network communication, the most costly of part of the gate is the comparison operation [22]. Only one is required per iteration, giving the shuffling step a communication complexity of $O(n^2)$. The transaction construction step is more efficient, requiring just 4 network messages for any number of participants, due to its client-server architecture.

Since the shuffling step has such high communication complexity, it was expected

that this would be the dominant factor in the run-time of the system. If a network communication takes twice as long to perform, and n^2 communications are required, then the system as a whole should be delayed by a factor of four. This should be reflected in the shape of a graph of latency against run-time.

5.1.2 Results

Latency (ms)	Run-time (s)
0	2.19
100	10.3
200	19.7
300	28.3
400	36.7
500	43.0
600	51.4
700	60.0
800	67.6
900	74.1
1000	83.8

Table 5.1: Results of Testing System Run-time at Various Network Latencies

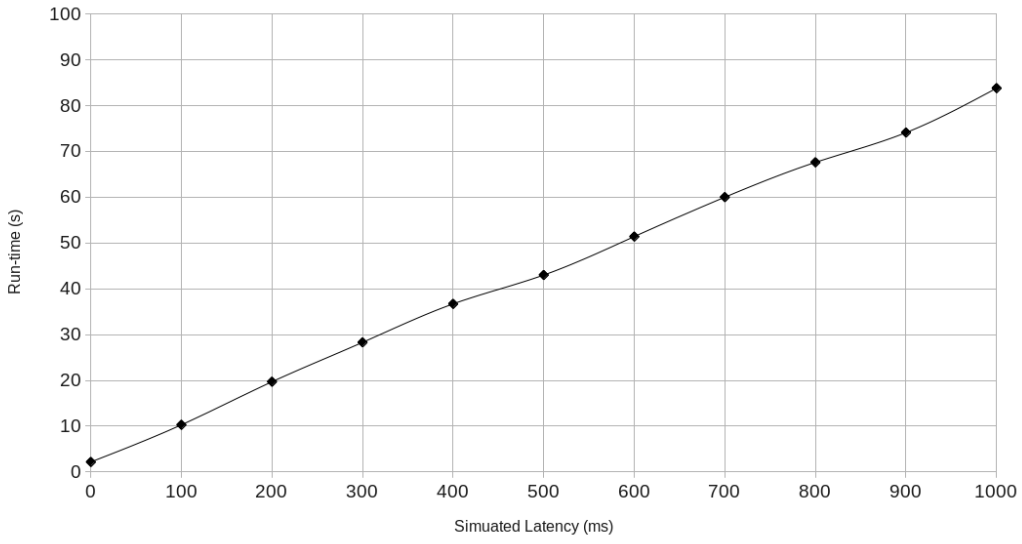

Figure 5.1: Graph of System Run-time at Various Network Latencies

5.1.3 Analysis

The results did not bore out the prediction, with the overall system run-time increasing linearly with communication latency. For typical internet connections, the run-time can be predicted as roughly 90 times the communication latency. This multiplier decreased slightly as latency increased, indicating that local computation was still a minor performance factor at low latencies. The reason for the linear relationship is likely the optimizations made in the VIFF framework. VIFF performs secure operations as soon as their required operands are available, meaning that many steps are automatically performed in parallel [22]. The same number of network messages are sent, but transmission bandwidth is used to its fullest.

5.2 Future Improvements

5.2.1 Single Language Implementation

The current implementation is written in a mixture of Java and Python, joined using shell scripts. This is sufficient for a proof-of-concept, but it leads to a poor production application. Frequently the output of a program is used as the input of another, and so any exceptions printed to standard output only lead to further errors later on. This masks the root cause of issues, making troubleshooting difficult. Usability is also poor. Each player must manually enter a number of parameters at the command line, and there is little hope of wrapping a GUI around the system in its present form. Finally, the shell scripts only allow the system to run on certain Unix variants, despite the core technologies it builds on being largely platform independent.

A single language implementation solves these issues, but requires either implementing parts of VIFF in Java or parts of BitcoinJ in Python. Both VIFF and BitcoinJ are large software projects, and so their translation was outside the scope of this project. Going forward, translating BitcoinJ is arguably a better choice, since VIFF makes use of many Python language features simply not present in Java.

5.2.2 Security Analysis

This report has presented informal - but hopefully persuasive - arguments as to the design's security. However, confidence in the system could be greatly improved by performing a formal analysis of it. Assuming that such an analysis

is successful, steps would have to be made to ensure that the implementation is indeed an implementation of the design. An incorrect implementation of a secure design can be as vulnerable as a perfect implementation of an insecure design.

5.2.3 Discovery Channels

There is currently no automated way to find other participants for a mixing session. This is a difficult task to solve; each participant must have a reasonable network connection, wish to mix the same amount of Bitcoin and be on-line long enough to allow the protocol to complete. Additionally, the channel must be resistant to a malicious party flooding it with messages, or even initiating mixing sessions and then stalling them.

Discovery can be achieved using a peer-to-peer network (as in the Bitcoin production network), or using some level of centralization, such as an agreed upon IRC channel.

Identifying malicious parties before they deviate from the protocol is likely impossible. Instead, steps should be taken to bar misbehaving players from future sessions. There are two main approaches: white-listing known honest players and black-listing known malicious players. A white-list alone is inappropriate here, since a larger number of potential participants provides better levels of anonymity between sessions. This leaves the option of black-listing misbehaving participants. In order to black-list a participant, however, it is necessary to identify them.

An identity system should, to the extent possible, have the following properties:

1. Authentication. The owner of the identity should be able to produce messages that (reasonably) could only have originated from them.

2. Decentralization. A central authority should not be required to manage the identities, since this runs counter to the security goals of the system. This includes preventing two players being given one identity.

A simple scheme that satisfies these constraints is a public-private key-pair. Authentication can be provided by a signed hash of a message in the usual way and the scheme is naturally decentralized since key-pairs can be generated locally. A large key size makes the chance of a collision negligible.

The problem with this method is that a new key-pair can be generated cheaply. This means that if a malicious player is black-listed, they can simply make a new identity and try again. This can be solved using a proof-of-work scheme similar to Bitcoin mining:

> Each participant generates a public-private key-pair. The participant then appends a nonce to the public key and adjusts it continually till its hash has n leading zeros. The public key and nonce are the identity and message origin can be proven using the private key.

The strength of an identity is then determined by the number of leading zeros in its hash. On average, a larger number of zeros will have required more candidate nonces to find, and so more computational work. Given that the alternative is to mine blocks, a strong identity has an opportunity cost in Bitcoin. If a player expects that there are a large number of malicious parties, then they can reject any weak identity, raising the cost of evading a black-list.

The downside to this approach is that an honest player must invest some work into establishing an identity. However, this is a one time cost, provided that they always behave honestly. Additionally, players may trust a weak identity if its owner has proven honest in the past using a supplemental white-list.

Another possibility is that an adversary launches multiple instances of the program and hence controls all but one participant in the mixing session. Under these circumstances the adversary can link the other participant's addresses by process of elimination. This can be mitigated somewhat by participating in multiple mixing sessions, since the adversary can only break a participant's privacy if it controls each mixing session. This can be furthered by requiring that each session contains participants with different IP addresses than the previous (although IP address is quite malleable) or with different identities (as in the scheme described above).

5.2.4 Automation

If there is an automated discovery system in place, then it is possible to automate the process as a whole. For example, a customized wallet program could perform mixing at regular intervals. The main issue here is that a user may wish to spend coins that are tied up in a mixing transaction. To avoid this, the user could declare an address for mixing. Coins claimable by this address would then be automatically mixed by the program, and would not be used as inputs in a regular transaction. To mitigate the risk of a mixing session controlled by a malicious adversary, there should be some chance of the mixed coins being sent back to the marked address.

6. Conclusion

The mixing protocol presented in this report provides a number of advantages over existing solutions. First, it makes anonymity more easily quantifiable. In existing systems, anonymity is a function of the trust placed in one entity, whereas here it is a function of the number of expected corrupted parties. Second, the system can be run in an ad-hoc manner. Participants only need network connections between each other, and some level of outside coordination, to successfully run a mixing session. There is no server whose failure can prevent future sessions from taking place. Finally, the protocol makes strong provisions against coins being lost, a serious issue for the centralized alternatives.

The proof-of-concept implementation was successful. It is capable of performing a mixing session for $n > 2$ parties mixing any number of Bitcoins. Crucially, it also runs in reasonable time, with a three-party mixing transaction taking around 10 seconds to generate, over a typical internet connection.

There are still many improvements to be made. Implementation of the features discussed in Future Improvements would allow for automated mixing in a secure and fairly efficient fashion. Distributed as part of a Bitcoin client, the system would provide Bitcoin with anonymity characteristics similar to that of physical currencies, removing another barrier to its widespread adoption.

Bibliography

[1] Noam Nisan Assaf Ben-David and Benny Pinkas. Fairplaymp - a secure multi-party computation system. *ACM CCS 2008*, 2008.

[2] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, STOC '88, pages 1–10, New York, NY, USA, 1988. ACM.

[3] Bitcoin fog. `http://www.bitcoinfog.com/`, 2013.

[4] Bitcoinj. `http://code.google.com/p/bitcoinj/`, 2013.

[5] Bitcoin laundry. http://www.bitcoinlaundry.com/, 2013.

[6] Bitcoin p2p digital currency. `http://bitcoin.org/`, 2013. The official landing page for the original client.

[7] Bitcoin wiki. `https://en.bitcoin.it/`, 2013. Bitcoin Wiki is a resource maintained by the Bitcoin community. It is the most complete document of how Bitcoin operates, compiled from the original white-paper, discussion with Nakaomoto and studying the source code of the original client.

[8] Paul E. Black. Fisher-yates shuffle. *Dictionary of Algorithms and Data Structures [online]*, 2011.

[9] Paul E. Black. bitonic shuffle. *Dictionary of Algorithms and Data Structures [online]*, 2012.

[10] Bitcoin block explorer - blockchain.info. `http://www.blockchain.info`, 2013.

[11] R. Cramer and I. Damgard. Multiparty computation, an introduction. *Contemporary Cryptology*, 2005.

[12] Ivan Damgrd and Tomas Toft. Trading sugar beet quotas - secure multiparty computation in practice. *ERCIM News*, 2008(73), 2008.

[13] Electrum. `http://electrum.org/`, 2013.

[14] Krishna Gade. Twitter search is now 3x faster. `http://engineering.twitter.com/2011/04/twitter-search-is-now-3x-faster_%1656.html`, 2011.

[15] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, STOC '87, pages 218–229, New York, NY, USA, 1987. ACM.

[16] Google guava. http://code.google.com/p/guava-libraries/, 2013.

[17] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[18] Netty. https://netty.io/, 2013.

[19] Fergal Reid and Martin Harrigan. An analysis of anonymity in the bitcoin system. *ARXIV*, 2011.

[20] Tp's testnet faucet. http://tpfaucet.appspot.com/, 2013.

[21] Online mirror of the first test result. http://blockexplorer.com/testnet/tx/26f5e04c38a4501b9229fde309a0fdde89b%f0fff33358a23fa108687cd677805, 2013.

[22] Tomas Toft. Implementing asynchronous multi-party computation. phd progress report. www.daimi.au.dk/~ttoft/publications/progress.pdf, 2005.

[23] Viff, the virtual ideal functionality framework. http://viff.dk/, 2013.

[24] Viff comparison protocols. http://viff.dk/doc/comparison.html, 2013.

[25] Edward Z. Yang. Secure multiparty bitcoin anonymization. http://blog.ezyang.com/2012/07/secure-multiparty-bitcoin-anonymization/%, 2012.

[26] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *Foundations of Computer Science, 1986., 27th Annual Symposium on*, pages 162 –167, oct. 1986.