Sigmajoin: Outsourceable Zerojoin

No Author Given

No Institute Given

Abstract. We present Sigmajoin, a privacy protocol for UTXO-based blockchains. Sigmajoin is an enhancement of Zerojoin, which is also a privacy protocol. Like Zerojoin, our protocol is non-interactive in the sense that Alice does not need to be online when Bob mixes their coins. However, Zerojoin requires a party to be online for remixing. Sigmajoin is even more non-interactive than Zerojoin because once parties add their coins to the pool, they don't need to be online for remixing, which can be carried out by a trustless service. Thus, while our protocol retains all the features of Zerojoin, it additionally provides *outsourceability*. Similar to Zerojoin, our protocol also uses proofs of knowledge of Diffie-Hellman tuples, where we prove equivalence of exponents in two pairs of group elements. The privacy of our protocol also depends on the hardness of the Decisional-Diffie-Hellman (DDH) problem. Like Zerojoin, our protocol can also be implemented using ErgoScript.

1 Introduction

Privacy enhancing techniques in UTXO-based blockchains generally fall into two categories. The first is obscuring the input-output relationships such as in Zerocoin [1], Composite Signatures (CS) [2], CoinJoin [3] and Zerojoin [4]. The second is hiding the amounts being transferred, such as in Confidential Transactions [5]. Some solutions such as Zcash [6,7], Quisquis [8] and MimbleWimble (MW) [9] combine both approaches.

In this work, we describe Sigmajoin, yet another privacy enhancing protocol based on the former approach of obscuring input-output relationships. Our protocol is motivated from Zerojoin [4] to overcome some of its limitations:

1. *Mix more than 2 boxes*: In Zerojoin, we can mix only two boxes at a time and it will be good to have a solution that works for more than 2 boxes
2. *Make it outsourceable*: In Zerojoin, Bob can mix a box of Alice without interaction. However, to remix a full-mix box, one must be online. Requiring participants to be continuously online makes the protocol *non-outsourceable*, and thereby less usable. This is also a problem with CoinJoin [3]. It is preferable to remix a box without the owner's intervention.
3. *Do away with half-mix boxes*: In Zerojoin, we need to create a half-mix box when playing Alice's role for remixing, while Bob's role can be played directly using the full-mix box. Thus, half-mix boxes are 'bloat' [4] and it will be better to directly work with full-mix boxes.

2 Background

2.1 Notation

Let G be a cyclic group of prime order q where the Decisional-Diffie-Hellman (DDH) problem is hard and let g, h, u, v be generators of G. In the following, $\textit{proveDlog}(g, u)$ refers to a zero-knowledge proof of knowledge of discrete logarithm of u to base g and $\textit{proveDHTuple}(g, h, u, v)$ refers to a zero-knowledge proof of equality of the following two discrete logarithms: u to base g and v to base h. Observe that $\textit{proveDHTuple}(a, a, b, b) = \textit{proveDlog}(a, b)$ for any a, b. We will use this relationship to emulate *proveDlog* in ErgoScript, our underlying smart contract language.

The proofs are described in Appendices A and B respectively, and are implemented using *Sigma protocols* [4,10]. For any two sigma protocols σ_1 and σ_2, the symbol σ_2 OR σ_2 refers to the zero-knowledge OR described in Appendix C, that is, a proof of knowledge of one of them without revealing which [10].

2.2 Basic idea

The following describes the high level idea for mixing two boxes at a time. The same idea can be trivially extended to mix three, four, etc boxes.

Zerojoin [4] consists of two types of boxes: a *half-mix* box protected by a script of type $\textit{proveDlog}(*, *)$ and a *full-mix* box having a script of type $\textit{proveDlog}(*, *)$ OR $\textit{proveDHTuple}(*, *, *, *)$. This makes the two boxes incompatible and we cannot remix as Alice without creating a half-mix box first. Note that we can still remix as Bob directly, but that is only possible if there are half-mix boxes available. Additionally, the half-mix box is a kind of "bloat" and it is preferable to operate with full-mix boxes only (called just "mix" boxes).

The idea of Sigmajoin is to start with two boxes having a script of type $\textit{proveDlog}(*, *)$ and have the mix also generate two boxes of type $\textit{proveDlog}(*, *)$. In particular, a mix box has two registers a, b containing elements of G and the owner has to prove the statement: $\textit{proveDlog}(a, b)$. Since we have only one type of box in Sigmajoin, we will have only one type of participant, called Alice.

3 Sigmajoin Protocol

Privacy in Sigmajoin is provided via a pool of coins (boxes). People add coins to the pool and later withdraw them in a manner that hides the links between the entry and exit boxes. A box in the pool, called a *mix-box*, is defined as follows:

- It has two registers labeled a, b containing elements of G.
- It is protected by the *mix-script* given in Section 3.4. One part of the script allows spending the box freely by proving the statement: $\textit{proveDlog}(a, b)$. The other part allows anyone to spend the box only in a restricted manner, that is, if it is used in mixing as per Section 3.2.

The protocol, illustrated in Figure 1, consists of three operations:

1. *Deposit*: Anyone can deposit coins in *fixed* denominations to the pool.
2. *Withdraw*: This allows anyone to withdraw their coins from the pool.
3. *Mix*: Anyone (the "mixer") can spend any two coins from the pool, thereby "mixing" them, and then add them back as two indistinguishable coins that preserve the original owners. The mixer need not be the owner of one of the inputs. A box is mixed several times before it is finally withdrawn.

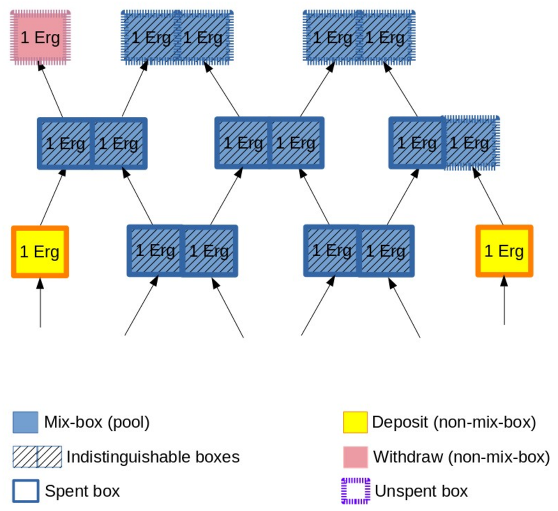

Fig. 1: Sigmajoin protocol

3.1 Deposit and Withdraw

Let g be a fixed generator of G. To deposit a box to the pool, Alice selects secret $x \overset{R}{\leftarrow} \mathbb{Z}_q$ and creates a box protected by the mix-script with registers (a, b) containing (g, g^x). Alice can withdraw her box at any time using the secret x.

3.2 Mix

A mix is done by selecting two mix-boxes from the pool and spending them in a transaction that generates two identical-looking mix-boxes. An output is generated by applying the following transformation each input.

1. Generate secret $y \overset{R}{\leftarrow} \mathbb{Z}_q$ and apply the transformation $(a, b) \leftarrow (a^y, b^y)$. In other words, register a (resp. b) of the new box is computed by raising register a (resp. b) of the old box to power y. The transformation preserves the secret exponent relationship between a and b, ensuring the owner's ability to spend the box anytime. This is called *re-randomising* the public key.
2. To ensure that the mix is done correctly, that is, both registers are raised to the same power, the mixer must additionally prove the following statement. Let (a_0, b_0) and (a_1, b_1) be the registers of the first and second output respectively. Then the mixer must prove:

$$\mathit{proveDHTuple}(a, b, a_0, b_0)\ \mathsf{OR}\ \mathit{proveDHTuple}(a, b, a_1, b_1)$$

3. We additionally need to ensure that $y \neq 0$, which can be done by requiring that $a_0 \neq b_0$ and $a_1 \neq b_1$.

3.3 Ergo and ErgoScript

We implement our protocol in ErgoScript, the scripting language of the Ergo blockchain [11,12]. We use Ergo because it has the *proveDHTuple* protocol built-in, is UTXO based and supports advanced scripts at context level C3 [13]. ErgoScript is a strict subset of the Scala language. It allows accessing the inputs and outputs of the transaction and specifying arbitrary constraints on them. Data in Ergo boxes is stored in up to 10 registers labeled R0 to R9, out of which R0 to R3 are reserved by the protocol and the remaining are for user-defined values [12]. We use the following mapping: $a \rightarrow$ R4 and $b \rightarrow$ R5.

The object to be proved, the content of a script is called a *proposition*, which consists of either one or two *statements*. A statement evaluates to `true` or `false` during runtime. When two statements are present, they must be joined using logical *And or* Or. A box can be spent when its script evaluates to `true`.

Statements are of two types: *primitive* and *compound*. A compound statement is made by joining two (primitive or compound) statements using *And* or *Or*. A primitive statement can be either a crypto-statement (such as `proveDlog` or `proveDHTuple`) or a predicate on the context (such as `a <= b`).

A crypto-statement requires a proof using which it evaluates to either `true` or `false` depending on whether the proof is valid or not. Thus, the output of every statement is always a boolean value, and the logical operations are meaningful. Whenever two crypto-statements are to be joined using *Or*, they are done using zero-knowledge (see Appendix C). Whenever a crypto-statement is to be joined with a predicate, the predicate is first evaluated and the crypto-statement is evaluated lazily. In other words, a statement like `proveDlog(...) && false` is considered equivalent to the predicate `false` and statements like `proveDlog(...) && true` or `proveDlog(...) || false` are considered equivalent to `proveDlog(...)`. The final statement is then just combination of pure crypto-statements joined using *Or* and *And*.

3.4 Smart Contract

The contract of a mix box is given below in ErgoScript

```
val a = SELF.R4[GroupElement].get      // current base for dLog
val b = SELF.R5[GroupElement].get
val owner = proveDHTuple(a, a, b, b)  // = proveDlog(a, b)
val mix = {
  val out0 = OUTPUTS(0)    // first output
  val out1 = OUTPUTS(1)    // second output
  val a0 = out0.R4[GroupElement].get  // register a of first output
  val a1 = out1.R4[GroupElement].get  // register a of second output
  val b0 = out0.R5[GroupElement].get  // register b of first output
  val b1 = out1.R5[GroupElement].get  // register b of second output
  val validOuts = out0.propositionBytes == SELF.propositionBytes &&
  out1.propositionBytes == SELF.propositionBytes &&
  out0.value == SELF.value &&
  out1.value == SELF.value &&
  a0 != b0 && // rule out point at infinity ToDo: explain more
  a1 != b1    // rule out point at infinity

  // at least one of the outputs has the right relationship between R4, R5
  val validAB = proveDHTuple(a, b, a0, b0) || proveDHTuple(a, b, a1, b1)

  validAB && validOuts
}

mix || owner
```

Contract 1: Mix-script

3.5 Analysis

With regards to security, no one should be able to spend Alice's box other than for mixing (*theft-prevention*) and Alice should always be able to spend her box (*spendability*). Both are guaranteed by ensuring the exponent relationship between a, b of at least one of the box in the mix, and additionally requiring a, b to be not O (the point at infinity). With regards to privacy, no outsider should be able to guess with an advantage, which output corresponds to which input. For $i \in \{0, 1\}$, the transformation of (a_i, b_i) from inputs to outputs using secret y_i can be written as: $(a_i, b_i) \leftarrow ({a_i}^{y_i}, {b_i}^{y_i})$. If the DDH problem is hard then the output distributions $(a_0, b_0), (a_1, b_1)$ are indistinguishable.

4 Extensions

Here we discuss various enhancements to the basic protocol presented above.

4.1 Outsourceability

Unlike Zerojoin, an interesting possibility with Sigmajoin is that of *outsourcing* the mix process, where the mixing is done in a trustless manner by third parties called mixers. In this approach, shown in Figure 2, multiple mixers operate on the same pool. The mixers are *non-custodial* in the sense that while they have the ability to deanonymize users, they don't have the ability to steal funds.

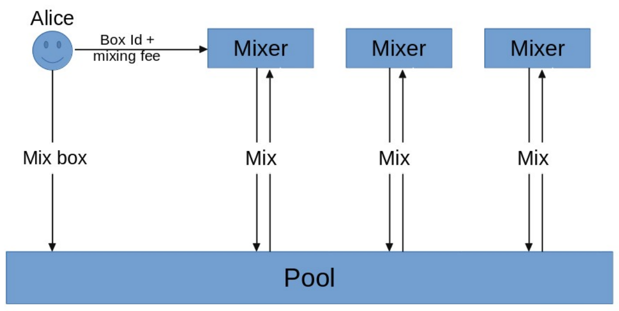

Fig. 2: Outsourced fee

Alice adds her mix-box to the pool and sends the box id (a globally unique identifier) to a mixer along with some *mixing fee*. The mixer then proceeds to mix Alice's box several times as determined by the mixing fee. One problem with this approach is that the mixer may lose track of Alice's box if someone else mixes her box in between. In order to prevent this, we can put a time-lock that prevents anyone else but the mixer from spending the box before a certain time. The modified contract is given below:

```
val a = SELF.R4[GroupElement].get  // current base for dLog
val b = SELF.R5[GroupElement].get
val m = SELF.R6[GroupElement].get  // public key of the mixer or dummy value
val h = SELF.R7[Int].get // height at which box was created
val lockTime = 5 // number of blocks for which box is time-locked
val owner = proveDHTuple(a, a, b, b)  // = proveDlog(a, b)
val mixer = proveDlog(m)              // = proveDlog(g, m), g is fixed
val timeOut = HEIGHT > h + lockTime
val mix = {
  val out0 = OUTPUTS(0)   // first output
  val out1 = OUTPUTS(1)   // second output
  val a0 = out0.R4[GroupElement].get  // register a of first output
  val a1 = out1.R4[GroupElement].get  // register a of second output
  val b0 = out0.R5[GroupElement].get  // register b of first output
  val b1 = out1.R5[GroupElement].get  // register b of second output
  val m0 = out0.R6[GroupElement].get  // just access group element
  val m1 = out1.R6[GroupElement].get  // just access group element
  val h0 = out0.R7[Int].get  // height at which first output is created
  val h1 = out1.R7[Int].get  // height at which second output is created
  val validOuts = out0.propositionBytes == SELF.propositionBytes &&
  out1.propositionBytes == SELF.propositionBytes &&
  out0.value == SELF.value &&
  out1.value == SELF.value &&
  a0 != b0 && // rule out point at infinity
  a1 != b1 && // rule out point at infinity
  h0 <= HEIGHT && // ensure that h0 is not too high
  h1 <= HEIGHT    // ensure that h1 is not too high

  // at least one output has the right relationship between R4, R5
  val validAB = proveDHTuple(a, b, a0, b0) || proveDHTuple(a, b, a1, b1)

  validAB && validOuts && (mixer || timeOut)
}

owner || mix
```

Contract 2: Mix-script with Outsourceability

To determine the mixing fee, assume that Alice wants to mix her box n times (example $n = 10$). At each mix, there is onex extra box created. In order to ensure that Alice's box is not identified by outsiders, the mixer must process every such extra box as if it were Alice's box. Thus, about 2^{n+1} mix transactions must be generated by the mixer, and in the worst case, the mixing fee should cover as many transactions. In reality, the mixer would combine Alice's and other's mixes, and so not all mixes will be just for Alice.

4.2 Stealth Withdraw

A mix box can either be used for (re)mixing or be withdrawn. When a mix box is reused in mixing, there is no *transfer-of-value* as the box owner remains the same. On the other hand, a withdraw is necessary for transfer-of-value. However, we note that the original protocol also allows transfers-of-value transactions that look like mix transactions and it becomes infeasible for an outsider to distinguish the two types.

This is possible because the owner has the ability to spend the box anytime because of the final condition `owner || mix`. The owner can do so in a manner where the final transaction looks like a mix transaction but is actually not. This can be done by ensuring that `validOuts` and `timeOut` in Contract 2 is true. The final contract then reduces to `owner || validAB` joined using a zero-knowledge proof. This allows the owner to create a transaction that resembles a mix transaction but does not require ownership to be preserved.

A transfer of value using this technique is a *stealth withdraw* since it cannot be distinguished from a mix transaction. Using the idea of stealth withdraw, we don't ever have to withdraw from the pool; all transactions happen within the pool, thereby increasing the privacy further. Note that while a mix can be outsourced, a stealth withdraw cannot.

5 Outsourced Fee

One of the challenging problems in mix protocols is that of mining fee. Since the fee has to be paid in a manner that does not reveal the source of funds, the parties cannot pay it using arbitrary boxes owned by them. Additionally, since the mix has to preserve the value, the fee cannot be paid using the tokens being mixed. There are two approaches to handling mining fee in such scenarios.

1. *Embedded*: The fee must be embedded in the mix boxes themselves. This is the primary approach is followed in ErgoMix [4] via *mixing tokens*.
2. *External*: The fee is paid by a third party who has no relationship with the participants. An example is the *altruistic fee* approach described in [4].

While the embedded fee approaches of [4] will also work with Sigmajoin, outsourceability allows us to design a much simpler external approach where the fee is paid by the mixer. We call this *Outsourced Fee*. Referring to Figure 2, each mixer maintains its own set of unspent boxes for covering mining fee, which we call *funding boxes*. [1]

There are two roles in the system and anyone can take one or the other role:

1. Mixers (privacy providers): Who provide the service for a profit in Ergs by performing the mix on behalf of users.
2. Users (Privacy seekers): Who want privacy of their tokens without ever losing control of them

The main requirement of a mixer is to keep a lot of funding boxes in advance to pay mining fee. The funding boxes should ideally be unrelated to the boxes being mixed. However, a mixer can use boxes where it received payment for its service as long as it selects such boxes randomly. We also discuss ways to make on-chain payment to a mixer without leaking information about the mixer or the box.

Although a user *can* perform a mix, (because the smart contract does not differentiate the two users), the privacy of our solution relies on the fact that only mixers perform a mix. A user performing its own mix can leak information via the source of fee. A user is free to mix random boxes from the pool, thereby following the "altruistic fee" approach. However, it makes more sense for such users to take the role of a mixer.

5.1 Contract

We will use the following modified contract in mix boxes. The contract has some differences from the earlier version:

1. For efficiency, instead of storing creation height in R7, we will use the context variable `creationInfo._1`, which already has the same value [14]. We will store some other information in R7 instead.
2. Instead of identifying a mixer using a single group element, we will use two elements, denoted m, n in the contract. This is to allow the possibility of stealth payments [15] to the mixer.
3. In addition to just the primary token (Erg), we require *all* tokens of the output boxes to be identical and same as the input tokens.

```
val a = SELF.R4[GroupElement].get  // public key of owner (base for dLog)
val b = SELF.R5[GroupElement].get  // public key of owner (base ^ secret)
val m = SELF.R6[GroupElement].get  // public key of the mixer (base)
val n = SELF.R7[GroupElement].get  // public key of the mixer (base ^ secret)
// We are using two elements (m, n) to represent mixer, rather than one
// because this allows "stealth" locking a box to a mixer
// (i.e., without revealing the mixer identity)
// So mixer will publish public key (u, v), two group elements.
```

[1] For a stealth withdraw, which cannot be outsourced, the simplest way is for the owner to privately purchase the private key of a funding box from a mixer.

```
// In order to lock the box to the mixer, Alice will set (m, n) = (u^r, v^r)
// for secret and random r

val h = SELF.creationInfo._1 // height at which box was created

val lockTime = 50 // number of blocks for which box is time-locked

val owner = proveDHTuple(a, a, b, b)  // = proveDlog(a, b)
val mixer = proveDHTuple(m, m, n, n)  // = proveDlog(m, n)

val timeOut = HEIGHT > h + lockTime

val mix = {
  val out0 = OUTPUTS(0)   // first output
  val out1 = OUTPUTS(1)   // second output

  val a0 = out0.R4[GroupElement].get  // register a of first output
  val a1 = out1.R4[GroupElement].get  // register a of second output
  val b0 = out0.R5[GroupElement].get  // register b of first output
  val b1 = out1.R5[GroupElement].get  // register b of second output

  val m0 = out0.R6[GroupElement].get  // access group element (not used)
  val m1 = out1.R6[GroupElement].get  // access group element (not used)
  val n0 = out0.R7[GroupElement].get  // access group element (not used)
  val n1 = out1.R7[GroupElement].get  // access group element (not used)

  val h0 = out0.creationInfo._1  // height at which first output is created
  val h1 = out1.creationInfo._1  // height at which second output is created

  // ensure outputs have same script as this box and have the same value
  val validOuts = out0.propositionBytes == SELF.propositionBytes &&
  out1.propositionBytes == SELF.propositionBytes &&
  out0.value == SELF.value &&
  out1.value == SELF.value &&
  out0.tokens == SELF.tokens &&
  out1.tokens == SELF.tokens &&
  a0 != b0 && // rule out point at infinity
  a1 != b1 && // rule out point at infinity
  h0 <= HEIGHT && // ensure that h0 is not too high
  h1 <= HEIGHT // ensure that h1 is not too high

  // at least one of the outputs has the right relationship between R4, R5
  val validAB = proveDHTuple(a, b, a0, b0) || proveDHTuple(a, b, a1, b1)

  validAB && validOuts && (mixer || timeOut)
}

owner || mix
```

Contract 3: Mix-script with Stealth Outsourceability

The contract implements the following rules:

1. Allows a 3rd party to 'lock' the box so that only that party can mix it for a certain time (`lockTime`).
2. Allows a 3rd party to mix the box as long as it does not violate rule 1 (i.e., after `lockTime` is over).
3. Allows the owner to always spend the box, overriding any conditions imposed by above rules.

In our solution, spending via rule 1 will be done by a mixer, one who has been explicitly paid to mix the tokens contained in the box. Spending via rule 2 will be done by a mixer who has not been paid to mix this box, but is doing so to enhance privacy of their mix transaction. In order to maintain privacy, the order of boxes will not determine whether it is being spent via rule 1 or 2.

Since spending the box requires mining fee, usually, anyone who spends via 1 or 2 will only do so if they get some benefit. There are two possible benefits:

1. A mixer will earn profit as service fee paid via a different (possibly offchain) channel.
2. A mixer will spend in order to gain privacy for a box it wants to mix.

5.2 Stealth Payments

Another useful primitive is *stealth payments* [15] (not to be confused with stealth withdraw of Section 4.2). To receive a stealth payment, first generate a random group element h and a secret x. Then compute $k = h^x$ and publish (h, k). A stealth payment is made by generating a box protected by the script `proveDHTuple`(h^r, h^r, k^r, k^r). This is considered 'stealth' because given (h^r, k^r), it is not possible to deduce (h, k) assuming that the DDH problem is hard.

In reality, we will actually store (h^r, k^r) in (R4, R5) and make the script `proveDHTuple`(R4, R4, R5, R5). Anyone expecting stealth payments to secret x must scan all boxes protected by the above script and select those where R5=R4^x.

Mixing fee can be paid via stealth payments to the mixer's public key (m, n) (stored in (R6, R7) of the mix box). In other words, the payment will be made to a box locked with the script `proveDHTuple`(m^r, m^r, n^r, n^r) for a secret r. Note that the initial value of (m, n) stored in the user's box should itself be the randomized version of the mixer's public key (as we did above with r), to hide the mixer to which the box is locked to. The mixer will then do a similar randomization at each mix.

The payment box can additionally store the user's box id encrypted using, say, El Gamal with the same public key (m, n).

5.3 Mixer Strategy

We define the strategy from a mixer's point of view. Let P be the set of mix-boxes that are free (i.e., not locked to any mixer) and let L be the set of boxes that are locked to this mixer. Thus, P and L are disjoint. We will assume that P always has an unlimited supply of boxes. We can incentivize users to keep their boxes in P as discussed later. The mixer can be bootstrapped once L has at least one box. The suggested strategy for the mixer is to always operate on boxes in L only

Operation Once a mixer is bootstrapped (i.e., L is sufficiently large), it can perform the actual mixes as follows.

1. Pick two boxes from L uniformly.
2. Mix them and add them back to L (i.e., lock them again).

3. Repeat and keep track of boxes in order to decide when a box is sufficiently mixed.
4. For boxes where the lock is about to expire (i.e., those about to be removed from L), spend them again in a mix transaction with other active boxes to ensure L always has enough boxes. If L already has enough boxes, the mixer can let such boxes be unlocked using a random strategy.

Bootstrapping Whenever the number of boxes in L go below a certain value, for instance, in a newly launched mixer, the following bootstrapping process needs to be followed:

1. If there are not enough boxes in L, a mixer will borrow from P by performing a mix transaction where one input is taken uniformly from L and the other from P and both outputs are added to L. This process is repeated until L is sufficiently large. Such mixes are considered 'fake' and will not be counted as mixing rounds for which users pay.
2. Once L is sufficiently large, the mixer is considered bootstrapped and operations can start.

In summary, the mixer will keep all customer boxes in L that have not completed their mix cycle. If there is more space in L, it will also keep those customer boxes that have completed their cycle (selected uniformly), and any remaining space is to be filled by non-customer boxes (i.e., from P). A mix transaction that takes one input from P and adds both outputs to L is called a 'fake' and doesn't count towards mix cycles. Real mixes are always to be carried out using boxes from L.

Customers can always track which boxes belong to them (by following the chain and testing each output with their secret). Thus, they can withdraw boxes from L or P anytime.

Mixing fee needs to be decided based on the cost of (1) maintaining a large L and (2) incentives to users for keeping boxes in P. Note that cost of (2) should be borne by multiple mixers, so the main cost is going to be (1). Several mixers can jointly share a large P and provide services varying in turnaround time, fee per round, etc. For implementation parameters, we suggest as follows:

Parameter	low privacy	medium privacy	high privacy
size of L	1k	10k	100k
mix rounds	20	30	50

5.4 User Strategy

A user should not use a single mixer, since it may be compromised. Instead, several mixers in succession must be used. The exact number depends on the assumptions. For example, suppose each mixer has probability m of being compromised, then user must select at least t mixers such that the chance of all t being compromised is, say, 1 in a million.

An optimistic implementation parameter would be use at least 3 mixers, while a more conservation one would be to have 10.

A user can switch mixers simply by spending the box from one mixer's L and adding it to the other's.

5.5 User Incentives

In order to ensure unlimited supply of companion mix boxes, we must incentivize users to park their funds in P. This can be achieved by rewarding boxes in P via stealth payments to the user's public key (a, b) (stored in R4, R5 of the mix box). As before, this is done by creating a box protected by the script `proveDHTuple`(a^r, a^r, b^r, b^r) for secret r [15]. The boxes getting the reward can be selected randomly, possibly based on the tokens being mixed. For instance, one may want to reward only boxes holding SigmaUSD [16] tokens.

Note that such incentives must be spent carefully lest they leak information about the box owner. Ideally they should be used for stealth payments only.

6 Future Work

The above idea is for two boxes at a time, but this number can be increased to three (and any fixed value) simply by adding the same constraints for the third output along with an additional *proveDHTuple* clause.

References

1. Ian Miers, Christina Garman, Matthew Green, and A.D. Rubin. Zerocoin: Anonymous distributed e-cash from bitcoin. In *Proceedings - IEEE Symposium on Security and Privacy*, pages 397–411, 05 2013.
2. Amitabh Saxena, Janardan Misra, and Aritra Dhar. Increasing anonymity in bitcoin. In *Financial Cryptography and Data Security*, pages 122–139. Springer, 2014.
3. Coinjoin: Bitcoin privacy for the real world. `https://bitcointalk.org/?topic=279249`, 08 2013.
4. Alexander Chepurnoy and Amitabh Saxena. Zerojoin: Combining zerocoin and coinjoin. In Joaquín García-Alfaro, Guillermo Navarro-Arribas, and Jordi Herrera-Joancomartí, editors, *Data Privacy Management, Cryptocurrencies and Blockchain Technology - ESORICS 2020 International Workshops, DPM 2020 and CBT 2020, Guildford, UK, September 17-18, 2020, Revised Selected Papers*, volume 12484 of *Lecture Notes in Computer Science*, pages 421–436. Springer, 2020.
5. Gregory Maxwell. Confidential transactions. `https://people.xiph.org/~greg/confidential_values.txt`, 2015.
6. Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 459–474, Washington, DC, USA, 2014. IEEE Computer Society.
7. Zcash. `https://z.cash`, 2016.

8. Prastudy Fauzi, Sarah Meiklejohn, Rebekah Mercer, and Claudio Orlandi. Quisquis: A new design for anonymous cryptocurrencies. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 649–678. Springer, 2019.
9. T.E. Jedusor. Mimblewimble. `https://download.wpsoftware.net/bitcoin/wizardry/mimblewimble.txt`, 2016.
10. Ronald Cramer, Ivan Damgård, and Berry Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. In Yvo Desmedt, editor, *Advances in Cryptology - CRYPTO '94, 14th Annual International Cryptology Conference, Santa Barbara, California, USA, August 21-25, 1994, Proceedings*, volume 839 of *Lecture Notes in Computer Science*, pages 174–187. Springer, 1994. `http://www.win.tue.nl/~berry/papers/crypto94.pdf`.
11. Ergo Developers. Ergo: A resilient platform for contractual money. `https://ergoplatform.org/docs/whitepaper.pdf`, 2019.
12. Ergoscript, a cryptocurrency scripting language supporting noninteractive zero-knowledge proofs. `https://docs.ergoplatform.com/ErgoScript.pdf`, 03 2019.
13. Alexander Chepurnoy and Amitabh Saxena. Bypassing non-outsourceable proof-of-work schemes using collateralized smart contracts. In Matthew Bernhard, Andrea Bracciali, L. Jean Camp, Shin'ichiro Matsuo, Alana Maurushat, Peter B. Rønne, and Massimiliano Sala, editors, *Financial Cryptography and Data Security - FC 2020 International Workshops, AsiaUSEC, CoDeFi, VOTING, and WTSC, Kota Kinabalu, Malaysia, February 14, 2020, Revised Selected Papers*, volume 12063 of *Lecture Notes in Computer Science*, pages 423–435. Springer, 2020.
14. Ergo data model. `https://docs.ergoplatform.com/dev/data-model/box/`.
15. Stealth address contract. `https://www.ergoforum.org/t/stealth-address-contract/255`, 06 2020.
16. Sigusd and sigrsv: A how-to guide. `https://ergoplatform.org/en/blog/2021-09-22-sigusd-and-sigrsv-a-how-to-guide/`, 09 2021.
17. Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology — CRYPTO' 86*, pages 186–194, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.
18. Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, 1991.
19. Ivan Damgård. On Σ-Protocols, 2010. `http://www.cs.au.dk/~ivan/Sigma.pdf`.

Appendix

A Proof of Knowledge of Discrete Logarithm

Let G be a cyclic multiplicative group of prime order q where Decisional Diffie-Hellman (DDH) problem is hard and let u, g be any generators of G. The following is a zero-knowledge *proof of knowledge of discrete logarithm* that proves knowledge of x such that $u = g^x$.

1. The prover, $\mathcal{P}$, picks $r \overset{R}{\leftarrow} \mathbb{Z}_q$ and sends $t = g^r$ to the verifier, $\mathcal{V}$.
2. $\mathcal{V}$ picks $c \overset{R}{\leftarrow} \mathbb{Z}_q$ and sends c to $\mathcal{P}$.
3. $\mathcal{P}$ sends $z = r + cx$ to $\mathcal{V}$, who accepts iff $g^z = t \cdot u^c$.

We apply the Fiat-Shamir transform [17] where the role of the verifier is replaced by a hash function, i.e., $c = Hash(t)$. This is a variation of Schnorr signatures [18] and denoted by *proveDlog*(g, u).

B Proof of Equivalence of Discrete Logarithms

Let $g, h, u, v \in G$. The following is a zero-knowledge *proof of equivalence of discrete logarithms* that proves knowledge of x such that $u = g^x$ *and* $v = h^x$. This is implemented using two parallel runs of *proveDlog*(g, u):

1. $\mathcal{P}$ picks $r \overset{R}{\leftarrow} \mathbb{Z}_q$ and sends $(t_0, t_1) = (g^r, h^r)$ to $\mathcal{V}$.
2. $\mathcal{V}$ picks $c \overset{R}{\leftarrow} \mathbb{Z}_q$ and sends c to $\mathcal{P}$.
3. $\mathcal{P}$ sends $z = r + cx$ to $\mathcal{V}$ who accepts iff $g^z = t_0 \cdot u^c$ and $h^z = t_1 \cdot v^c$.

As before we set $c = Hash(t_0, t_1)$. The protocol is called *proveDHTuple*(g, h, u, v). Note that *proveDHTuple*(g, g, u, u) = *proveDlog*(g, u).

C Sigma-Or

A protocol with this structure $(\mathcal{P} \overset{t}{\rightarrow} \mathcal{V}, \mathcal{P} \overset{c}{\leftarrow} \mathcal{V}, \mathcal{P} \overset{z}{\rightarrow} \mathcal{V})$ is called a sigma protocol if it satisfies *special soundness* and *honest-verifier zero-knowledge* [19].

The statement to be proved (example "I know the discrete logarithm of u to base g") is denoted by τ. Any sigma protocol can be made non-interactive via the Fiat-Shamir transform [17] by setting $c = H(t)$ where H is a hash function.

As shown in [10], any two sigma protocols for arbitrary statements τ_0, τ_1 can be efficiently composed to a single sigma protocol that proves knowledge of one of the witnesses without revealing which. Let $b \in \{0, 1\}$ be such that $\mathcal{P}$ knows the witness of τ_b but not τ_{1-b}. $\mathcal{P}$ simulates the proof of τ_{1-b} to get an accepting transcript $(t_{1-b}, c_{1-b}, z_{1-b})$ and generates t_b properly. $\mathcal{P}$ sends (t_0, t_1) to $\mathcal{V}$. On receiving c, $\mathcal{P}$ computes $c_b = c \oplus c_{1-b}$ and then uses t_b, c_b to compute the response z_b properly. Here $\oplus$ is the bit-wise XOR operation. Finally $\mathcal{P}$ sends (z_0, z_1, c_0, c_1) to $\mathcal{V}$, who accepts iff both (t_0, c_0, z_0) and (t_1, c_1, z_1) are accepting transcripts and $c = c_0 \oplus c_1$. We call such a construction τ_0 OR τ_1. The expression τ_0 OR τ_1 OR τ_2 is to be interpreted as $(\tau_0$ OR $\tau_1)$ OR τ_2.