Securing Monero Transactions using Tandem

Linus Gasser[1], Christian Grigis[1], and Wouter Lueks[1]

[1]EPFL, Lausanne, CH

September 2022

Abstract

When using cryptocurrencies, one of the main problems consists of handling the private key. An attacker should not be able to create private transactions.

One could use threshold-cryptography to solve this problem, but then all servers know about all transactions from the user. This is specifically a problem in privacy-enhanced cryptocurrencies like Monero.

In this paper we apply Tandem to the creation of transactions in Monero. We show how to build a Threshold Cryptography Protocol for Monero, and prove that it is secure and privacy preserving.

Creating Monero transactions using Tandem allows securing the private key of a user against malicious use by an attacker.

1 Introduction

With the upcoming Web 3.0, more and more services rely on private keys. This includes, but is not limited to, blockchains and identity management systems that allow a user to selectively disclose one or more attributes. Another element of Web 3.0 is decentralization, which makes it harder to revoke a stolen or lost private key.

The most common protections for these private keys include *Wallets as a service* like hardware wallets and hardware security modules. But these solutions all contain a single point of failure that can leak the private key or lead to abuse. To avoid a single point of failure, one can use threshold cryptographic solutions like Calypso or Sepior [https://sepior.com/]. However, for a service that is privacy preserving, these solutions leak the identity of the key user.

For this reason Tandem has been developed. It is a threshold-cryptographic solution like the ones cited above. But it does not leak the identity of the client, which is important when it is used for a service that is privacy preserving. The Tandem paper shows how to use this solution for Attribute-based credentials. In this setting a simple threshold-cryptographic solution can protect the key, but it will deanonymize the user.

We looked at the Monero blockchain, which provides anonymous token transfers. These transfers are initiated with transactions that are signed by a private key. We show how to

convert a Monero transaction into a Tandem-compatible protocol without the need to modify Monero. Our contributions are the protocols to be used between the client and the Tandem server, tests on the Monero test network, and a reference library written in Rust.

2 Monero and Tandem

2.1 Monero

Monero is a cryptocurrency built with a blockchain that has interesting privacy preserving properties: all data from a transaction, like the senders, the receivers, and the amounts, are hidden in the public blocks. Only the participants in the transaction, that is the sender and the receiver, can decrypt the information. Even though this information is hidden, Monero is secure against double-spending or rogue minting of assets through the use of *Zero Knowledge Proofs* and *Group Signatures*.

Like most of the cryptocurrencies using blockchains, Monero also relies on a private key to sign transactions. This is why using Tandem to secure signing the transactions can improve the security of the system against key theft under certain circumstances.

Monero uses the Unspent Transaction Output (UTXO) model, which means that each address is only used twice: once when it receives tokens, and once when the tokens are spent. These addresses are derived from what Monero calls *view keys*. Contrary to *spending keys*, *view keys* only allow to discover the transaction addresses and the amount sent. In order to move tokens from one address to another, the client needs to sign using the *spending key*. If this key is stolen by an attacker, all funds from all address can be moved to an address chosen by the attacker. For this reason we want to protect the usage of the spending key with Tandem.

2.2 Tandem

Tandem is a protocol that uses a central server, but still allows to add the following two securities in a privacy-preserving manner:

- Rate limiting of key usage
- Blockage of key usage in case of theft

The first security reduces the damage an attacker can do between the time they get access to the key and the time the client blocks it. While both can be implemented using a threshold-key sharing mechanism, this would not be privacy-preserving, as all the servers involved would learn who is using their key. Tandem allows to implement both of these securities without the servers learning anything about who is using their private key.

If an attacker can get hold of the phone set up for use with Tandem, they will be able to create *some* transactions. They will be rate-limited by Tandem, and can only empty addresses until the client discovers the attack and blocks them, or until they run out of Tandem tokens. As we must suppose that the attacker knows the addresses of the victim, as well as the amount of tokens in each address, we can limit the damage by putting a similar amount of tokens in each

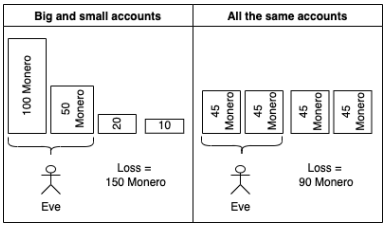

Figure 1: Comparison of emptying accounts with similar values and emptying accounts with high values

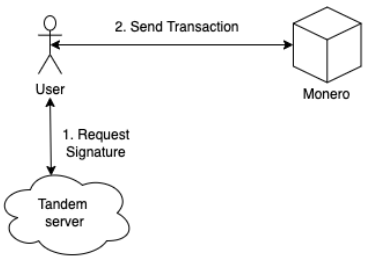

Figure 2: Actors in our system

address. By doing this, we can limit the amount of tokens stolen to the number of addresses an attacker can empty before they are detected and blocked. On the contrary, if most of the tokens are in only one address, the attacker will simply start emptying this address, and future blocking will not limit the damage to the user. The figure 1 shows two cases, where in each cases the attacker gets to empty the same number of accounts. In the first case, the attacker can withdraw much less funds than in the second case.

3 System

As can be seen in figure 2, Monero does not have to interact with the Tandem system. Only the Client interacts with Tandem. This means that there is no need to change the way Monero is built, but only the way the client interacts with Monero. In a regular interval, the client will request tokens from the Tandem server and store them locally. Now every time the client wants to sign a transaction for Monero, they send a token to the Tandem server and receive an answer that can be used to finalize the signature. The Tandem server never knows if a given token belongs to a certain client. Also if the client sends two tokens to the server, it cannot know that these two tokens belong to the same client.

We consider two threat models: in the first, the attacker has complete control over the phone of the client. This attack allows to withdraw funds from the clients addresses, so that the client has a financial loss. In the second, the Tandem server tries to infer who is asking for a signature by looking at the blocks produced by Monero. So if the Tandem server has knowledge of the produced signature, it can find the corresponding transaction and get a list of possible input and output addresses for each client. Over time the Tandem server could get enough information to censor certain clients.

The Tandem server can protect against the above threats by giving the following security guarantees:

1. Rate limiting of signatures per client
2. Allowing the client to block future signatures
3. Letting the client sign anonymously Monero transactions

To implement this, the Tandem server gives each client the possibility to get a limited amount of anonymous Tokens. Each token gives the client the possibility to request a signature on a Monero transaction from the Tandem server. During this signature request, the Tandem server doesn't learn anything about the client, except that they have the right to create a signature during this period and that they are not blocked. To limit the amount of signatures an attacker can do, the client can choose the validity of the tokens within epochs, which can span hours or days. In such a way only a certain amount of tokens can be used during an epoch, and if an attacker can steal the tokens, they cannot use them outside of this time window.

4 Background

When a client wants to transfer funds in Monero, they need to create a transaction and sign all the input funds using MLSAG signatures. The transaction contains different elements, as described in (02M, 6.3), but only a subset of those need to be protected by Tandem.

4.1 Notation

In order to simplify the notation used by Monero, the following changes are done to the Monero addresses compared with the document *Zero to Monero* (02M):

- don't include the $_t$ to indicate different address inputs
- don't include the $_B$ to indicate Bob's keys
- use a capital $_S$ and $_U$ to indicate the Tandem server and user share respectively

4.2 Tandem with Monero

In section 5 we describe the TCP protocols necessary to run Monero with Tandem. Here we describe the TCPs needed, as well as the optimizations necessary for this combination to work well.

4.2.1 Threshold Cryptography Protocols

A transaction of Monero is comprised of the following parts (02M 6.3). Parts with no involvement of the private key of the user have been omitted:

- MLSAG Signature, allows hiding the source of the payment by creating a multi-signature

- Key image, an input address corresponding to the sender, but obfuscated
- Pseudo output commitment, used to hide which of the inputs is the real input
- Output commitment, used to prove that the sum of the inputs equals the sum of the outputs
- One-time address, a destination address that hides the receiver, while allowing detection of the address by the receiver
- Encoded amount, in such a way that the signer can still prove that the sum of the inputs is the same as the sum of the outputs
- Range proof, a proof that none of the inputs are negative, which would allow the signer to create coins.

They are described in details in the next section.

4.3 Monero Addresses

As described in (02M, 4), Monero uses different types of addresses to hide payment patterns:

- *One-time* addresses are created by the sender from the public receiver's address
- *Subaddresses* can be created by a receiver to differentiate payments
- *Integrated addresses* are created by the sender by including a payment-ID that can only be read by the receiver
- *Multisignature addresses* that need more than one sender to sign before the transaction can be accepted - but they are out of scope of this document

The different addresses are used like this:

1. The receiver publishes one of the following:
   (a) their K^v, K^s keypair OR
   (b) their subaddress $K^{v,i}, K^{s,i}$
2. The sender generates a one-time address
3. Optional: the sender adds a payment-ID to create an integrated address

4.3.1 One-Time Address

To create a one-time address out of a receiver's keypair, or a receiver's subaddress keypair, the sender only needs to add a random number and can create the payment address:

$$K^0 = \mathcal{H}_n(rK^v)G + K^s \tag{1}$$

In the case of a subaddress, use $K^{v,i}, K^{s,i}$ instead of K^v, K^i. To spend the content of this address, the receiver has to calculate the corresponding private key like this::

$$k^0 = \mathcal{H}_n(rK^v) + k^s \tag{2}$$

According to (02M, 3.5), the sender has to hold the k_π for every input they want to spend. This corresponds to the k^0 described in Equation 2. The sender has to calculate the following:

$$\begin{aligned}\tilde{K} &= k_\pi \mathcal{H}_p(K_\pi) &(3)\\ &= (\mathcal{H}_n(rK^v) + k^s)\mathcal{H}_p(K_\pi) &(4)\\ &= \mathcal{H}_n(rK^v)\mathcal{H}_p(K_\pi) + k^s\mathcal{H}_p(K_\pi) &(5)\end{aligned}$$

K_π is the one-time address of a previous output. As k^s is protected by the Tandem protocol, we need to create a TCP to calculate $\tilde{K}$. Looking at Equation 5, the left part of the sum can be calculated by the client, and only the right part needs to be protected by the Tandem protocol.

4.3.2 Pseudo Output Commitment

According to (02M, 5.4), the pseudo output commitment is calculated without any private key intervening, only blinding factors. So we need to keep in mind that we should not threshold-share the blinding factors. If we must do so, this section needs to be revisited.

4.3.3 Output Commitment

According to (02M, 5.4), the user needs the *blinding factor* of the previous output commitment. This blinding factor is calculated using $\mathcal{H}_n(rK^v, t)$, which depends only on public parameters.

4.3.4 Encoded Amount

According to (02M, 5.3), the encoded amount depends on $\mathcal{H}_n(rK^v, t)$, which can be calculated as it only depends on public parameters.

4.3.5 Range Proof

According to the Bulletproofs paper, I cannot find any private key intervening in the creation of the Bulletproof. So IMO there is no need to use Tandem for the Range proof.

4.4 MLSAG Signature

The MLSAG signature of Monero is described in (02M, 3.5). A summary of the steps is:

1. *Key image*: $\tilde{K_j}$ - this can be calculated as described in subsubsection 5.2.1.
2. *Generate random numbers*: $\alpha_j \in_R \mathbb{Z}l$ and $r_{i,j} \in_R \mathbb{Z}_l$. α_j need to be used in the TCP, while the $r_{i,j}$ can be chosen by the user. This is because the α_j also appear in step 5, which involves the private keys of the inputs, and so must be done using a TCP.
3. *Compute initial commitment* $c_{\pi+1}$ also needs to be done in a TCP, as it involves the α_j.
4. *Compute other commitments* c_{i+1} can be done on the user's side, without the involvement of the Tandem server.
5. *Define all* $r_{\pi,j}$ needs to be calculated in a TCP, as it involves the private keys $k_{\pi,j}$

4.4.1 Threshold Cryptography Protocol (TCP)

To use Tandem, you need to be able to rewrite an existing cryptographic protocol as a Threshold Cryptography Protocol (TCP). Furthermore, this TCP must be *linearly randomizable*. So Tandem can be used to secure the keys of any cryptographic scheme (e.g., encryption, signature, or payments) for which a linearly randomizable threshold cryptographic version exists.

4.4.2 Blinding Public Inputs

Some of the calculations that need to be turned into a TCP contain public inputs, like when calculating $\alpha_i \mathcal{H}_p(K_{\pi,i})$ in the MLSAG signature. The simplest solution is that the user sends $\mathcal{H}_p(K_{\pi,i})$ to the server, so that the server can multiply it with its $\alpha_{i,S}$. However, this will give the server some information about the user, as the $\mathcal{H}_p(K_{\pi,i})$ references a unique input. So the user has to blind the input to the server like this:

$$\beta \mathcal{H}_p(K_{\pi,i}) \tag{6}$$

Now the server can multiply this with $\alpha_{i,S}$ and send it to the user. Then the user can multiply this with β^{-1} to get the desired result.

5 Threshold Cryptography Protocols

5.1 One Time Address

To create a TCP-version of this, the user will have to ask the server to calculate

$$k_S^0 = \mathcal{H}_n(rK^v) + \tilde{x}_S^s \tag{7}$$

with $\tilde{x}_S^s$ being the server share of the private key x^s. But this means that the user will be able to re-construct x when receiving the k_S^0, which is of course not wanted. To avoid the disclosure of the private share of the tandem server, this calculation is only executed when actually using it

Server		**User**
$\tilde{x}_S$		$\tilde{x}_U$
		$\beta \in_R \mathbb{Z}_p$
		$\tilde{\mathcal{H}} = \beta \mathcal{H}_p(K_\pi)$
	$\xleftarrow{\tilde{\mathcal{H}}}$	
$\tilde{\mathcal{H}}_S = \tilde{x}_S \tilde{\mathcal{H}}$		
	$\xrightarrow{\tilde{\mathcal{H}}_S}$	
		$\mathcal{H} = \beta^{-1}\tilde{\mathcal{H}}_S + \tilde{x}_U \mathcal{H}_p(K_\pi) = k^s \mathcal{H}_p(K_\pi)$
		$r \in_R \mathbb{Z}_p$
		$\tilde{K} = \mathcal{H}_n(rK^v)\mathcal{H}_p(K_\pi) + \mathcal{H}$

Table 1: A TCP implementation to calculate the *Key Image* needed in a Monero transaction.

in a MLSAG signature. As will be shown in subsection 4.4, applying k^0 in the MLSAG signature will avoid the problem of leaking the Tandem server share.

5.2 Simple Inputs

A number of inputs for the transaction are quite easy to transform into a TCP protocol. They are treated separate from the subsection 4.4 calculation of the MLSAG.

5.2.1 Key Image

The term $\mathcal{H}_p(K_\pi)$ needs to be blinded, in order to avoid that the Tandem server can recognize the client, as this expression is publicly available and linkable to a single user. So the TCP protocol becomes can be written as in Table 1.

5.3 MLSAG Signature

5.3.1 Compute Initial Commitment

In this step, the server and the user will chose the initial $\alpha_{j,S}$ and $\alpha_{j,C}$, which will also be used in subsubsection 5.3.2. All indexes $_j$ are computed for the whole range of $\{1, 2, ...m\}$.

5.3.2 Define all $r_{\pi,j}$

Before calculating the closing $r_{\pi,j}$ values, the user has to calculate the c_{i+1} values for $i = \pi + 1, ..., n-1, 0, 1, ..., \pi - 1$. This is possible without reverting to either one of the private keys or the the α_j values, so it is done on the user side without intervention of the server. Now the user has the value of c_π for the following TCP. Also, the server and the user use the same values for $\alpha_{j,S}$ and $\alpha_{j,U}$ as in subsubsection 5.3.1. For the private keys to the input $k_{\pi,j}$, they need to be calculated according to subsubsection 4.3.1, but this time inside of the equation for $r_{\pi,j}$:

Server		User
$\alpha_{j,S} \in_R \mathbb{Z}_p$		$\alpha_{j,U} \in_R \mathbb{Z}_p$
		$\beta_j \in_R \mathbb{Z}_p$
		$\tilde{\mathcal{H}}_{j,U} = \beta_j \mathcal{H}_p(K_{\pi,1})$
	$\xleftarrow{\tilde{\mathcal{H}}_{j,U}}$	
$\tilde{\mathcal{H}}_{j,S} = \alpha_{j,S}\tilde{\mathcal{H}}_{j,U}$		
	$\xrightarrow{\tilde{\mathcal{H}}_{j,S}}$	
		$\tilde{\mathcal{H}}_j = \beta_j^{-1}\tilde{\mathcal{H}}_{j,S} + \alpha_{j,U}\mathcal{H}_p(K_{\pi,1})$
$G_{j,S} = \alpha_{j,S}G$		
	$\xrightarrow{G_{j,S}}$	
		$G_j = G_{j,S} + \alpha_{j,U}G$
		$c_{\pi+1} = \mathcal{H}_n(m, [G_j], [\tilde{\mathcal{H}}_j], ...)$

Table 2: A TCP implementation to calculate the *initial commitment* needed in an MLSAG.

Server		User
$\tilde{x}_S$		$\tilde{x}_U$
$\alpha_{j,S}$		$\alpha_{j,U}$
	$\xleftarrow{c_\pi}$	
$r_{j,S} = \alpha_{j,S} - c_\pi \tilde{x}_S$		
	$\xrightarrow{r_{j,S}}$	
		$r_{j,U} = \alpha_{j,U} - c_\pi \tilde{x}_U$
		$r_{\pi,j} = r_{j,S} + r_{j,U} - c_\pi \mathcal{H}_n(\tilde{r}_j K^v)$

Table 3: A TCP implementation to calculate the $r_{\pi,j}$ needed in an MLSAG.

$$r_{\pi,j} = \alpha_j - c_\pi k_{\pi,j} \tag{8}$$
$$= \alpha_j - c_\pi(\mathcal{H}_n(\tilde{r}_j K^v) + k^s) \tag{9}$$
$$= \alpha_j - c_\pi \mathcal{H}_n(\tilde{r}_j K^v) - c_\pi k^s \tag{10}$$

To avoid giving away any additional information, we only calculate $\alpha_j - c_\pi k^s$ with the TCP. The TCP for the calculation of r_π can be found in Table 3.

Question: is it safe to send c_π to the server without blinding?

6 Conclusion

We present an adaption of Tandem to the Monero blockchain, allowing to protect the private key from theft while keeping signing of transactions private. The extension with Tandem of a Monero transaction does not need any change in the Monero protocol. We showed what the client and the server need to calculate, and show that the server does not gain any knowledge about the users' identity when they sign a new transaction.