Aggregate Addresses in CryptoNote: Towards Efficient Privacy

BETA COIN TEAM

September 8, 2015

1 Introduction

CryptoNote [1] is an application layer protocol for private egalitarian cryptocurrencies. As the first implementation of CryptoNote, Bytecoin inherits its properties while introducing a number of new features.

CryptoNote utilizes ring signatures for *untraceability* and one-time stealth keys for *unlinkability*. These properties should be satisfied by a fully anonymous electronic cash and are defined as follows [1]:

- **Untraceability**: for each incoming transaction all possible senders are equiprobable.
- **Unlinkability**: for any two outgoing transactions it is impossible to prove they were sent to the same person.

While CryptoNote protocol is designed to suit end users, it may set a challenging task for enterprise solutions. For example, a payment processor is required to differentiate incoming payments from many users, but creating a new address for each unlikable payment is not feasible. Another such case is a web wallet service which deals with thousands of users' keys and may encounter performance issues while searching the blockchain with thousands of transactions for those sent to each of the users' addresses.

Fortunately, CryptoNote protocol can be used efficiently in such settings. In this paper we describe an efficient solution for bulk handling of Bytecoin addresses without need to change the protocol. Our scheme is designed to process transactions efficiently regardless of the number of addresses. Nevertheless, it still ensures the same level of privacy as the regular CryptoNote protocol.

2 Privacy vs Efficiency

How CryptoNote Works

In CryptoNote, the secret data associated with an address consists of two elliptic private keys: a and b. The tuple (a, b) is called *spend key*. Two public keys are derived from them: A and B. A CryptoNote address is an encoded string `AB` plus additional information, like error-detection bits. The full specification can be found in CryptoNote Standard 007 [3].

To send money, a user takes the receiver's address `AB` and a random number r to generate (via Diffie-Hellman protocol) one-time public keys for each transaction output. He puts the value $R = rG$ in the transaction and this data is enough for the receiver to recover the corresponding one-time private keys with his spend key. One-time scheme guarantees that all output keys are unique and unlinkable to the receiver's address. The process of one-time keys generation is described in CryptoNote Standard 006 [2].

Introducing the second key gives rise to the unique feature of CryptoNote: *view key* [1, p. 8]. View key is a tuple (a, B), i.e. one private and one public key. It does not allow to redeem the money (unlike the spend key), but it can be used to determine if an output was sent to the address `AB`. A user can share his view key with a server to delegate transaction processing (blockchain parsing) and keep only a lightweight wallet client with his spend key on a mobile device.

CryptoNote also utilizes ring signatures to maintain the sender's privacy. A ring signature refers to a set of public keys of multiple users instead of a single one, which makes the actual signer indistinguishable from the other possible senders in the set. Ring size n is chosen by the signer: for example, with $n = 1$ he is unambiguously identified as the sender, while with $n = 20$ it is possible to identify 20 potential senders, but not to tell which one of them is the real one. Ring signatures are described in CryptoNote Standard 002 [4].

The Problem

The ability to send and receive transactions in private is undoubtedly useful for a single person, but sometimes may become an obstacle for a large scale operation.

The untraceability of a sender makes the process of receiving transactions less trivial for a large payment processor. An ambiguity occurs if there are two invoices with the same amount and only one (anonymous) transaction. It is not a significant issue for personal relations with ad-hoc examination, but for enterprises an automated solution should exist.

Another natural impediment appears when view key is used to delegate the ability to process transactions. A web wallet server with thousands of user keys

has to unambiguously determine all incoming payments for each address. A naive approach, as we will show below, would require asymptotically large amount of computational resources.

3 Existing Solutions

Naive Approach

The simplest way to deal with multiple addresses is to process each of them separately. This involves checking every combination of transaction output and user key. If there are N users and M outputs the total time is proportional to $M \cdot N$. Obviously, it does not suit enterprise purposes well, as they assume very large N.

Payment ID

Payment ID was introduced to address the problem of payment identification. When issuing an invoice, a server generates a random ID and sends it to the customer. The latter includes ID in his transaction as plain text, so it can be easily recognized by the server. As long as their channel is private (no one can see ID before the transaction is broadcasted) the scheme is secure.

This approach is quite simple and reasonable, but requires interactive communication between the customer and the server: ID must be known in advance. However, it is possible to construct a scheme for independent generation of the same ID on both sides (for example, by using a stream cipher).

4 Aggregate Addresses

The main idea behind *aggregate addresses* is simple: share one key between all addresses and save time on repeating calculations with the same data when receiving a payment.

The shared key is a, which is an essential (private) part of view key (a, B). The second key is, of course, different for each address, i.e. all spend keys — (a, b_1), (a, b_2), (a, b_3), etc. — are distinct. A server can now store one shared key a and generate random b_i instead of choosing both keys at random. Alternatively, it can store several shared keys a_k and divide the whole set of addresses into several subsets by aggregating by a_k. We will refer to such keys and addresses as "aggregate view/spend keys" or "aggregate addresses".

To understand the effect, we need to take a closer look at how the payments are received. Here is the picture from CryptoNote whitepaper:

has to unambiguously determine all incoming payments for each address. A naive approach, as we will show below, would require asymptotically large amount of computational resources.

3 Existing Solutions

Naive Approach

The simplest way to deal with multiple addresses is to process each of them separately. This involves checking every combination of transaction output and user key. If there are N users and M outputs the total time is proportional to $M \cdot N$. Obviously, it does not suit enterprise purposes well, as they assume very large N.

Payment ID

Payment ID was introduced to address the problem of payment identification. When issuing an invoice, a server generates a random ID and sends it to the customer. The latter includes ID in his transaction as plain text, so it can be easily recognized by the server. As long as their channel is private (no one can see ID before the transaction is broadcasted) the scheme is secure.

This approach is quite simple and reasonable, but requires interactive communication between the customer and the server: ID must be known in advance. However, it is possible to construct a scheme for independent generation of the same ID on both sides (for example, by using a stream cipher).

4 Aggregate Addresses

The main idea behind *aggregate addresses* is simple: share one key between all addresses and save time on repeating calculations with the same data when receiving a payment.

The shared key is a, which is an essential (private) part of view key (a, B). The second key is, of course, different for each address, i.e. all spend keys — (a, b_1), (a, b_2), (a, b_3), etc. — are distinct. A server can now store one shared key a and generate random b_i instead of choosing both keys at random. Alternatively, it can store several shared keys a_k and divide the whole set of addresses into several subsets by aggregating by a_k. We will refer to such keys and addresses as "aggregate view/spend keys" or "aggregate addresses".

To understand the effect, we need to take a closer look at how the payments are received. Here is the picture from CryptoNote whitepaper:

arately. With aggregate addresses, however, the first key (a) is the same in all addresses, so the derivation D only needs to be computed once per output. The subsequent addition involves B, which is unique for each address, but this can be circumvented: instead of computing $P' = D + B$ and comparing the result with P, it is possible to compute $B' = P - D$ and compare the result with B. Here, the subtraction needs to be evaluated only once for each output, not once for each address.

Finally, instead of naively comparing B' with each B_i, it is possible to create a hash table with the values B_i as keys. After that, a single lookup would be enough to check that B' is equal to some B_i and to get the value of i if this is the case. The improved scheme is shown in Figure 3.

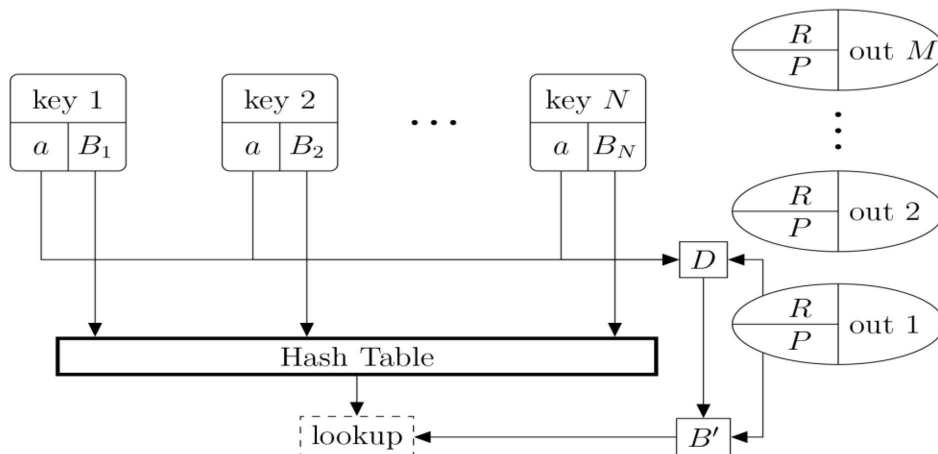

Fig. 3. CryptoNote transaction processing with aggregate addresses

The computations are performed as follows:

1. First, for each output, the derivation is computed: $D = H_s(aR||n)G$, where n is the index of the output in the transaction.
2. Then, for each output, the value B' is computed: $B' = P - D$.
3. The value B' is looked up in a hash table which contains the keys B_i. If there is a match, the output was to the corresponding address.

In contrast to the naive approach, the time it takes to process an output doesn't depend on the number of users. It is necessary to create a hash table in advance, but the time to do this is proportional to the number of users. As a result, the time it takes to process M outputs if there are N users is proportional to $M + N$, not $M \cdot N$ as with the naive approach.