Auditing wallets in CryptoNote

sowle <*val@zano.org*>

Zano project, https://zano.org

Cryptocurrency wallet auditing is the ability for a third party (the "auditor") to watch the transactions and to be able to calculate the correct balance without an ability to spend a coin.

This article explores several possible implementations of expansion cryptocurrency protocol CryptoNote 2.0 [1] with such ability. In the original CryptoNote protocol auditing *is only partially possible* with the help of the tracking key, namely, an auditor is able to distinguish incoming transactions in the blockchain, but the full set of secret keys is required to filter out outgoing transactions.

This article is intended for readers familiar with the general blockchain technology and “classic” cryptocurrencies, as well as with the basics of cryptography on elliptic curves.

1. Introduction

What is CryptoNote?

Surprisingly, most people interested in blockchain technology have never heard anything about CryptoNote, in spite of the fact that the technology has more than 300 forks, including Monero as the most famous.

Back in 2014 in the cryptocurrency community there were mentions [2] about a project, titled Bytecoin.
That project did not originate as a Bitcoin or other known project fork, having its own original codebase. It was very unusual at the time. Bytecoin general conception was to be an implementation of a privacy-technology named *CryptoNote*. There were two main privacy mechanisms: stealth-addresses and inputs` mixing-in with the help of ring signatures (at the time it was called "blockchain mixer"). Since Zcash existed only on paper / in theory at that time, CryptoNote became a competitive technology and has provoked much controversy in the cryptocurrency community.

Soon after a group of enthusiasts interested in one of the forks of CryptoNote, overtook that fork and with their great efforts brought a lot of attention from investors and the community. That fork was named BitMonero [3, 4], but very soon it was renamed to Monero.

Later on the development of Bytecoin and Monero were going their own different ways: Bytecoin remains to be a closed door project with an unknown background, while Monero became a big community-driven international project with lots of participants and developers.

Nevertheless both of them are an evolution of CryptoNote technology.

The audit and its applications

In a classic pseudo-anonymous blockchain like Bitcoin everyone is able to calculate the balance of an arbitrary address by scanning the blockchain and performing simple calculations. In contrast, in CryptoNote it is not

possible without additional data. Firstly, thanks to using stealth-addresses, there are no open unencrypted addresses in the blockchain. This property is usually referred to as **anonymity**. Secondly, it is hard to track coins movement because an input of a transaction refers to a set of possible outputs, but not the only one output. This property is usually referred to as **untraceability**.

Although for a traditional private cryptocurrency both of these properties are required, there are some cases in which the owner of a wallet may want to disclose the balance and transaction history for a third party with a guarantee they cannot spend the funds. As an example, it may be useful for exchanges in order to interact with the regulators. Or for foundations that would like to be transparent for certain parties or to be entirely public.

Formally speaking, cryptocurrency wallet auditing is the ability for a third party (the "auditor") to watch the transactions and to be able to calculate the correct balance without an ability to spend a coin. In the original CryptoNote protocol *auditing is only partially possible*. With the help of the tracking key an auditor can distinguish incoming transactions in the blockchain, but the full set of secret keys is required to filter out outgoing transactions.

A little bit about the author and the goal of this article

This article was written by sowle, one of Zano project's lead developers. Zano is an evolution of CryptoNote with a new strong consensus algo (PoW/PoS) hybrid and has been in development for several years by the same hands that wrote the original code of the technology.

Our team is looking forward to implementing wallet auditability for Zano and have begun to do research in that area. The author wants to introduce the results of some of these studies to readers in this article.

2. Basic intro to elliptic curve cryptography

CryptoNote uses the elliptic curve from a public-key signature system ed25519 [5].

Recall the main parameters of this curve and give additional definitions.

1. Chosen a big prime number q = 2^{255} - 19.
   All calculations are performed in finite field F_q of integers modulo q.

2. An elliptic curve is defined over the field F_q, denoted as E/F_q:
   $-x^2+y^2= 1 +dx^2 y^2$
   where d = −121665/121666 (this is an integer because, as everything else, is calculated in F_q).
   It's important that the curve is symmetric with respect to x and y.

3. There is an operation defined over the set of the curve points, that for any two points A, B gives another point C: F(A, B) = C, which is called "addition", and a special point at infinity as a neutral element (please refer to [6] for detailed explanation). This operation is closed to the set of the curve points, has associativity, commutativity and every element has an inverse, thus all the points with this operation is an *abelian group*, denoted as $E(F_q)$
   The order of this group (number of all the points) is: $\#E(F_q) = 2^c l$, where c = 3 (cofactor) and $l = 2^{252} + 27742317777372353535851937790883648493$.

4. Each curve point is defined by its coordinates (x, y). Because both coordinates are linked by the curve equation, using both x and y to represent a point is excessive, and therefore for the sake of data economy only y-coordinate encoded as a 256-bit integer is used in the implementation. Thanks to curve equation

symmetry (see above) the x-coordinate can have only one of two possible values for a given y-coordinate, and that choice is encoded as the most significant bit of a 256-bit integer representing y.

5. Set a special point G = (x, −4/5). It is set by its y-coordinate and from two possible x values the positive one is chosen.

6. Multiplication by integer n is defined as addition (see also i. 3) point G to itself n times. This operation forms closed multiplicative group **G** with order less, than $E(F_q)$:
   $\#G < \#E(F_q)$,
   while $\#G = l = 2^{252} + 27742317777372353535851937790883648493$.

7. Public key X is a curve point belonging to the group **G**:
   $X \in G$

8. Secret key x, or scalar, corresponding to public key X — is an integer such that:
   $X = x * G,\ x \in [1;\ l-1]$
   Secret key is encoded as a 256-bit integer as well as a public key.

9. The main hash-function H (in the code and other sources it is called `cn_fast_hash`). It maps arbitrary data to a 32-bytes long hash:
   $H : \{0,\ 1\}^{*} \rightarrow [0, 2^{256} - 1]$

10. Scalar hash-function H_s (subscript s stands for "scalar") maps arbitrary data to an integer such that it is a scalar, i.e. a valid secret key:
    $H_s : \{0,1\}^{*} \rightarrow [1;\ l-1]$
    It is possible to trivially define H_s in terms of H:
    $H_s(x) = H(x)\ \ mod\ (l-1)\ + 1$

11. Deterministic hash function H_p (p stands for "point") maps arbitrary data to a element of group **G**, i.e. a valid public key:
    $H_p : \{0,1\}^{*} \rightarrow G$
    Implementation of deterministic hash function is not a trivial task because, firstly, not any 246-bit number can be decoded to an elliptic curve point (see also i. 4), and, secondly, not any curve point belongs to **G** (see also i. 6).
    A possible simple implementation of H_p is consequent data hashing: H(H(...H(x)...)) until the result could be decoded into a point $X \in G$.
    CryptoNote is using a much more complex and more effective implementation (named `ge_fromfe_frombytes_vartime` in source code). It is covered in details in [7].
    Let's define a function that maps arbitrary 256-bit integer into an element of **G** in a deterministic way as:
    $\text{to\_point} : [0, 2^{256} - 1] \rightarrow G$
    In CryptoNote deterministic hash-function H_p implemented as follows:
    $H_p(x)$ = to_point(H(x))

3. Accounting and balance calculation in CryptoNote 2.0

Recall how the funds are sent and balance is calculated in the original version of the protocol.

When Alice sends coins to Bob she forms transaction outputs as the follows (Fig. 3.1).

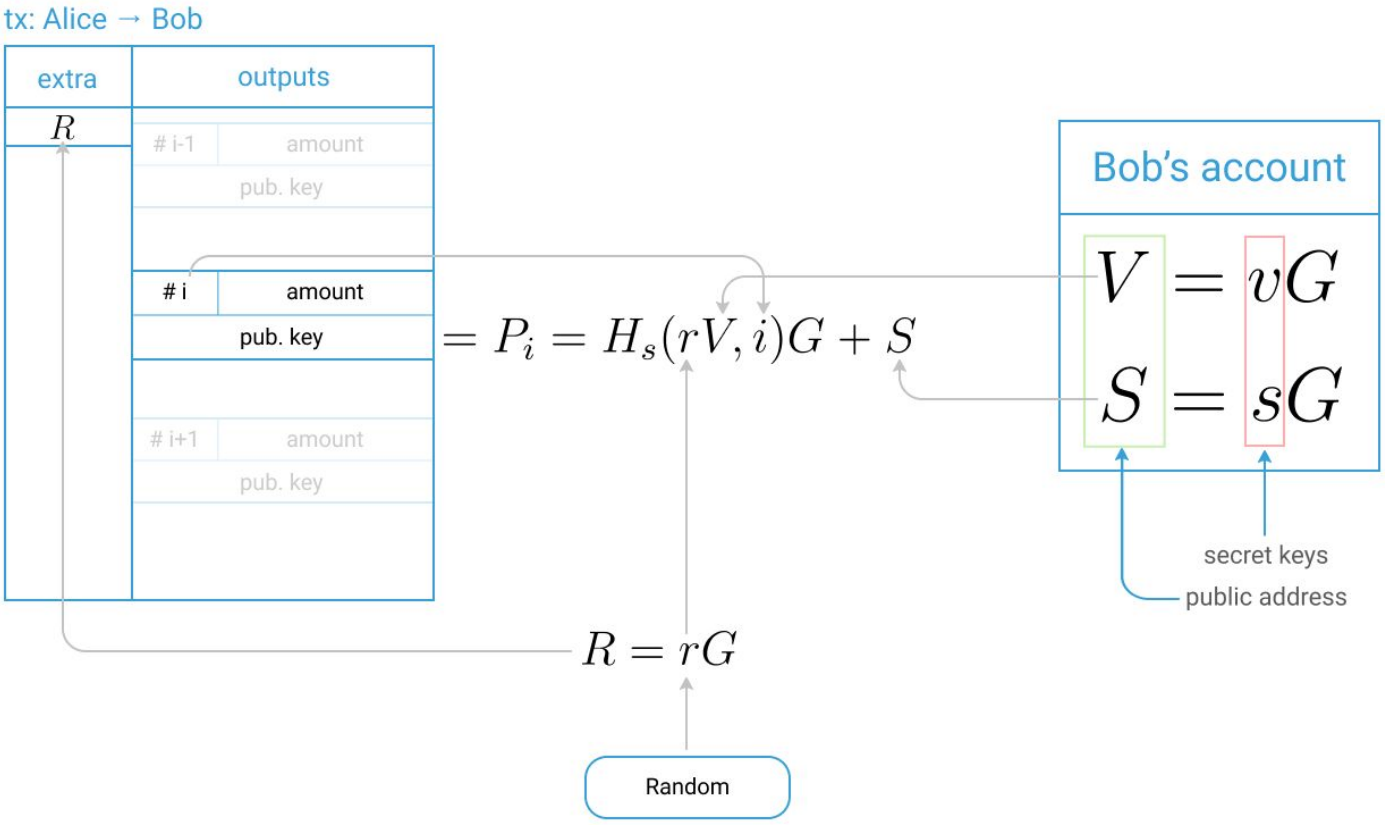

Fig. 3.1. Alice constructs transaction outputs while sending funds to Bob

1. Bob has a pair of secret keys (v, s). He calculates his public address as a pair of corresponding public keys (V, S) and then makes it publicly available or sends it to Alice.

2. Alice randomly chose transaction secret key r and calculates corresponding public key R = r * G, which she puts in the "extra" part of the transaction.

3. For each output Alice calculates corresponding stealth-address (also known as "one-time destination key"):
   $P_i = H_s(r * V, i) * G + S$, where i is the output index number within the transaction.

4. Alice signs and sends the transaction.

If an outside observer tried to analyze stealth-addresses P_i he could not link a specific output to Bob's address. Moreover, he could not even determine whether different outputs with the keys P_i и P_j are targeted to the same recipient or not.

In order to receive funds Bob scans over all blockchain transactions and checks them as the follows (Fig. 3.2).

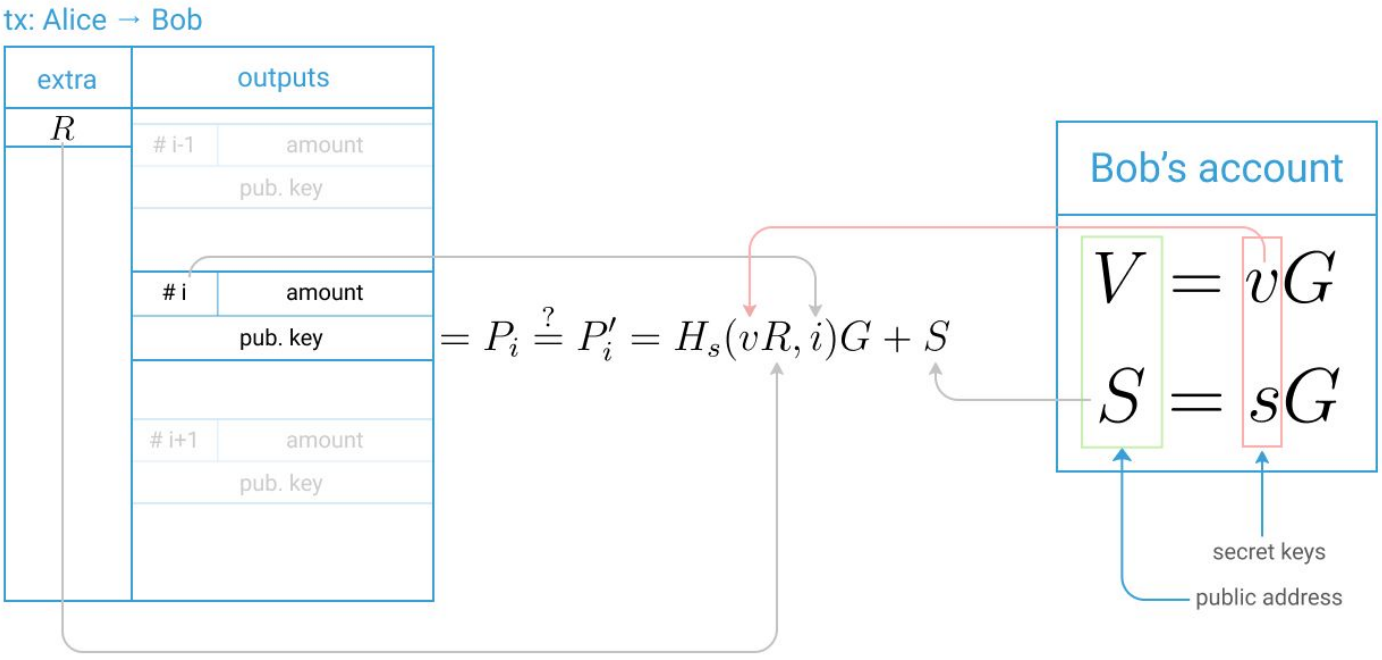

Fig. 3.2. Bob checks transaction for outputs that are targeted to him

5. Using his secret key v, for each output Bob calculates:
   P'_i = Hs(v * R, i) * G + S (where i is an output's index, S is Bob's public spend key).
   If P'_i == P_i, it means that Bob is the receiver of that output and he can later spend it by calculating the corresponding secret key.
   Therefore Bob increases his balance by the amount of the output.

To spend an output that targets Bob as the valid receiver, he does the following.

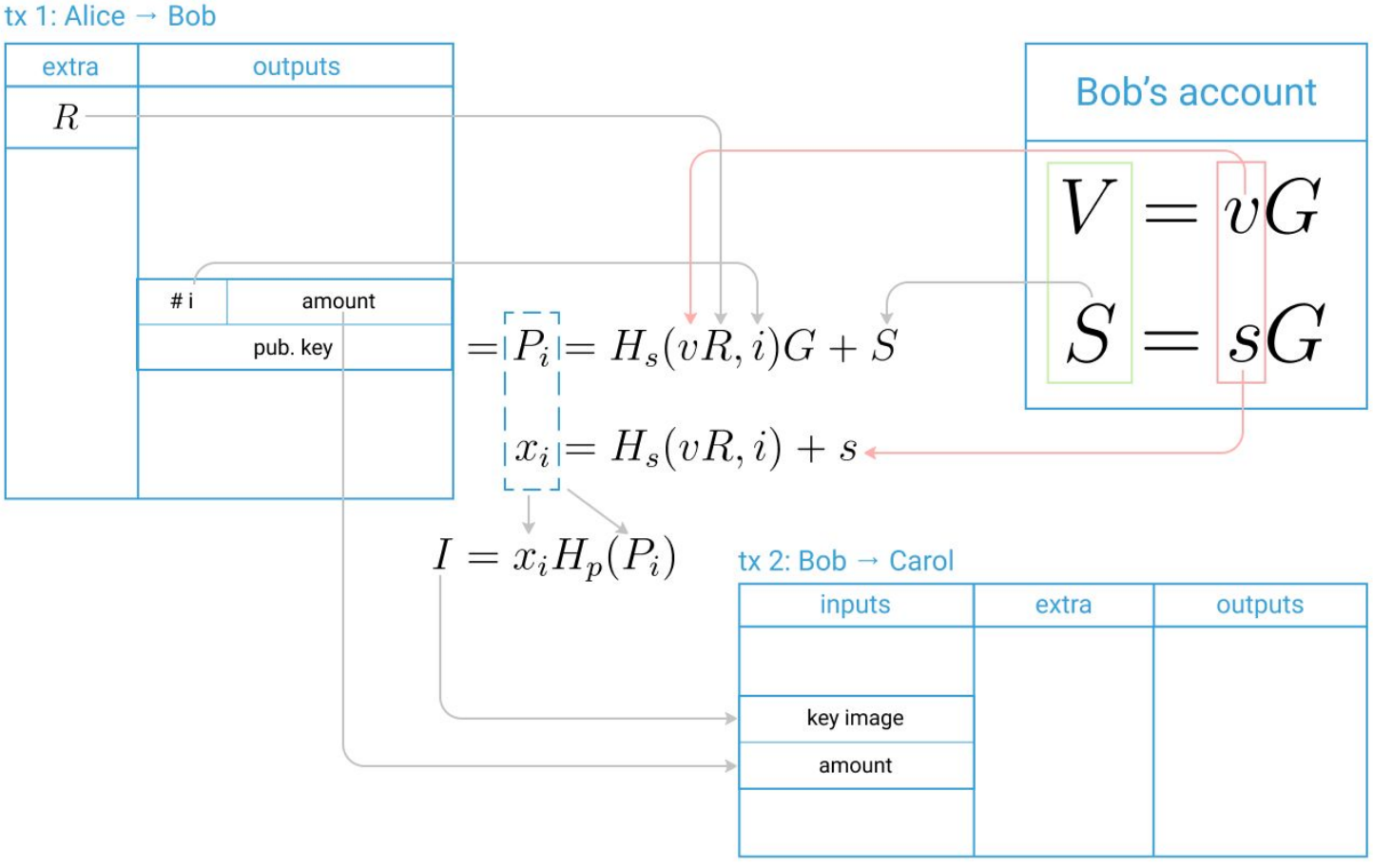

Fig. 3.3. Bob spends his output making an input for a new transaction to Carol

6. Using secret keys (v, s) Bob calculates one-time secret key $x_i = H_s(v * R, i) + s$ for a one-time public key (stealth-address) P_i, i.e. $P_i = x_i * G$.

7. Bob calculates the *key image* for the output: $I = x_i * H_p(P_i)$ and put it, the amount and a reference to corresponding output into the input part of his transaction to Carol.
   It is important to emphasize that, firstly, only the owner of secret spend key s is able to calculate the key image and no one else (correctness of the key image will be verified by a ring signature) and, secondly, an outside observer is generally unable to link the key image I with corresponding output's stealth-address P_i.

8. Bob decreases his balance by the amount of used output from i. 6.

9. Bob completes the transaction by filling its outputs as it was shown above in i.i. 2-3, then signs and broadcasts the transaction.

If we suppose that Bob completely lost his transaction history but still has the secret keys (v, s), he can recover transaction history and calculate his actual balance as the following (Fig. 3.4).

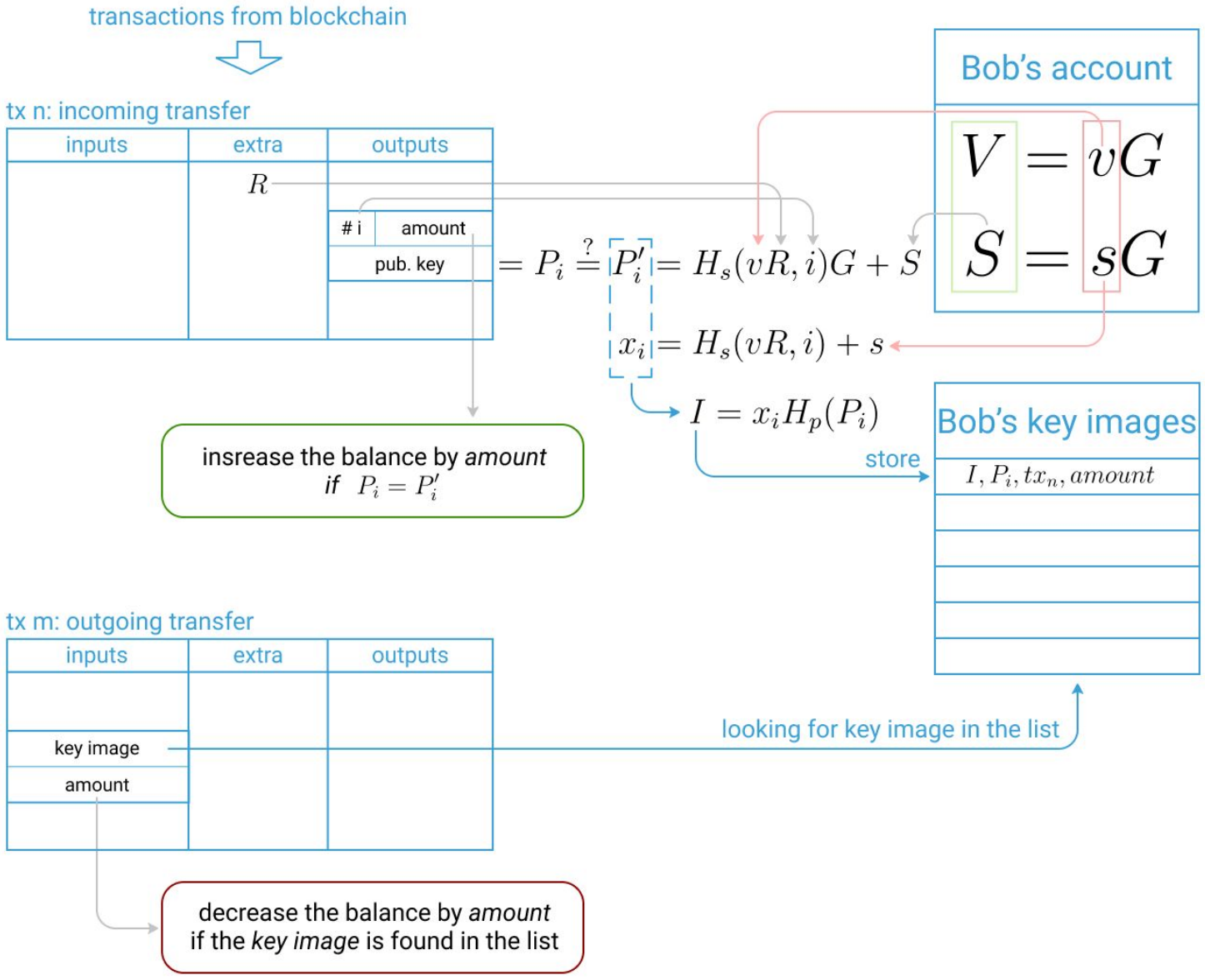

Fig. 3.4. Bob scans the blockchain for his spends and incomes, and calculates the balance

10. Bob scans over the entire blockchain and checks each transaction for outputs that are targeted to him (see also i. 5).

11. When such an output P_i is found Bob increases his balance by output's amount. Using secret spend key s he calculates corresponding output's secret key x_i (i. 6) and the key image I (i. 7). Then he puts the key image, P_i and other data of that transfer to the list.

12. If Bob finds a known key image in an input of a transaction while scanning the blockchain it will mean that transaction was made and signed by Bob. In that case he decreases his balance by output's amount.

Following the steps above Bob can fully restore the transaction history and his actual balance.

Note that if an auditor Dan gets Bob's secret view key v, he will be able to identify all Bob's *incoming* transfers. However, without the secret key s, he **will not be able** to recognize Bob's outgoing transactions and, therefore, calculate the correct balance. As will be shown later, in the case of direct output spending (without mixing-in), Dan will be able to identify such a transaction as Bob's outgoing transfer, but in the general case this cannot be done.

Thus, **to calculate a balance in the general case both secret keys (v, s) are needed.**

However, if Bob gives both of his secret keys (v, s) to the auditor Dan it will be tantamount to transferring the funds themselves, since Dan, having s, can spend them. Therefore, a full wallets audit in the original CryptoNote 2.0 is impossible in general case.

In the following sections we will look at a few protocol modifications that allow to implement a full wallets audit.

Note, that it's worthy for Alice to keep a transaction secret key r forever (and it actually happens in some implementation of the protocol). Anyone who knows r for a transaction can check whether an output i targets the given address (V, S) following the calculations from i.3:
$P'_i = H_s(r * V, i) * G + S$
and comparing the result with P_i.
Alice can use it, for example, to prove that she actually sent the funds to Bob.
However, it's impossible to recover the target address (V, S) from P_i, even knowing the secret key r.

4. Option 1 of 3: Bytecoin Auditable Coins

Bytecoin was the first and only implementation of CryptoNote when it appeared, thus it has all features and limitations, covered above.

On February 7th, 2019 Bytecoin version 3.4.0 "Amethyst" was released [20]. It had new interesting changes to CryptoNote that we will cover below. Some of the information in this section was published on their blog or site, but the most was obtained by deep code analysis by our own research.

In the scope of the article's topic the most interesting change is the ability to make a special copy of a wallet called *auditable wallet* (AW) having the following features:

1) AW cannot spend any coins;
2) balance of AW is always equal to the balance of its original main wallet (the wallet the AW was created from). And it's impossible to change the balance of the main wallet in such a way that the balance of corresponding AW would not change.

However, such a feature is applicable only for addresses of new version wallets, so called *amethyst-addresses*. Prior-to-amethyst addresses and accounts are backwards compatible, they are referred to as *legacy-addresses*. New features are possible in transactions of a new version only, therefore in the Bytecoin network currently both old and new transactions are possible at the moment. New version txs support both amethyst and legacy addresses, thus resulting in three cryptographic schemes:

1) tx.version < amethyst, legacy address;
2) tx.version >= amethyst, legacy address;
3) tx.version >= amethyst, amethyst address.

Let us take a closer look at them.

4.1. tx.version < amethyst, legacy address

This scheme is equivalent to the original CryptoNote with deterministic generation of r.

Wallet's account is a combination of a spend secret key s and hash v_s. Secret view key v is generated in a deterministic way as a function of v_s.

Now transaction secret key r is not a random but calculated using v_s:
$r = H_s(h_t, v_s)$, where $h_t = H(tx.inputs, tx.version)$

Thus, there is no need to store secret keys r locally for the further references, because for each own transaction put into the blockchain such secret key could be easily calculated using v_s.

Wallet balance calculation is the same as in CryptoNote (see also section 3), i.e. to identify outgoing transfers one need to know spent secret s.

If we assume that *all* recipient addresses that were ever used to send coins to are stored locally, then it is possible to calculate wallet balance *without* knowing spend secret s. It could be done as the following:

1. Alice scans the blockchain, identifies all incoming transfers with her view secret key v, and increases the balance as it was described in section 3.
2. For each transaction output Alice iterates through all her recipients' addresses (V, S) to whom she had ever sent coins to and calculates:
   $P'_i = H_s(r * V, i) * G + S$
3. If $P'_i = P_i$ then this output is Alice's outgoing transfer to address (V, S) and thus she decreases the balance accordingly.

Using this method Alice could calculate her balance using only v_s and v.

It's easy to see that if any address, to which coins were sent are absent in the local storage for some reason, the balance will then be incorrect. Thus, this method is unreliable and impractical, and

Therefore it is rather of theoretical interest.

4.2. tx.version >= amethyst

As was mentioned above, Bytecoin changed the transaction format from the version 3.4.0 Amethyst.
If tx.version >= amethyst, transaction outputs have another format.
Each output in addition to amount and public key P_i, also has additional public key Q_i (in the code it is referred to as `encrypted_secret`) and additional byte with the encrypted address type: either amethyst or legacy (referred to as `encrypted_address_type`). Outline of these data structures is shown on Fig. 4.2.

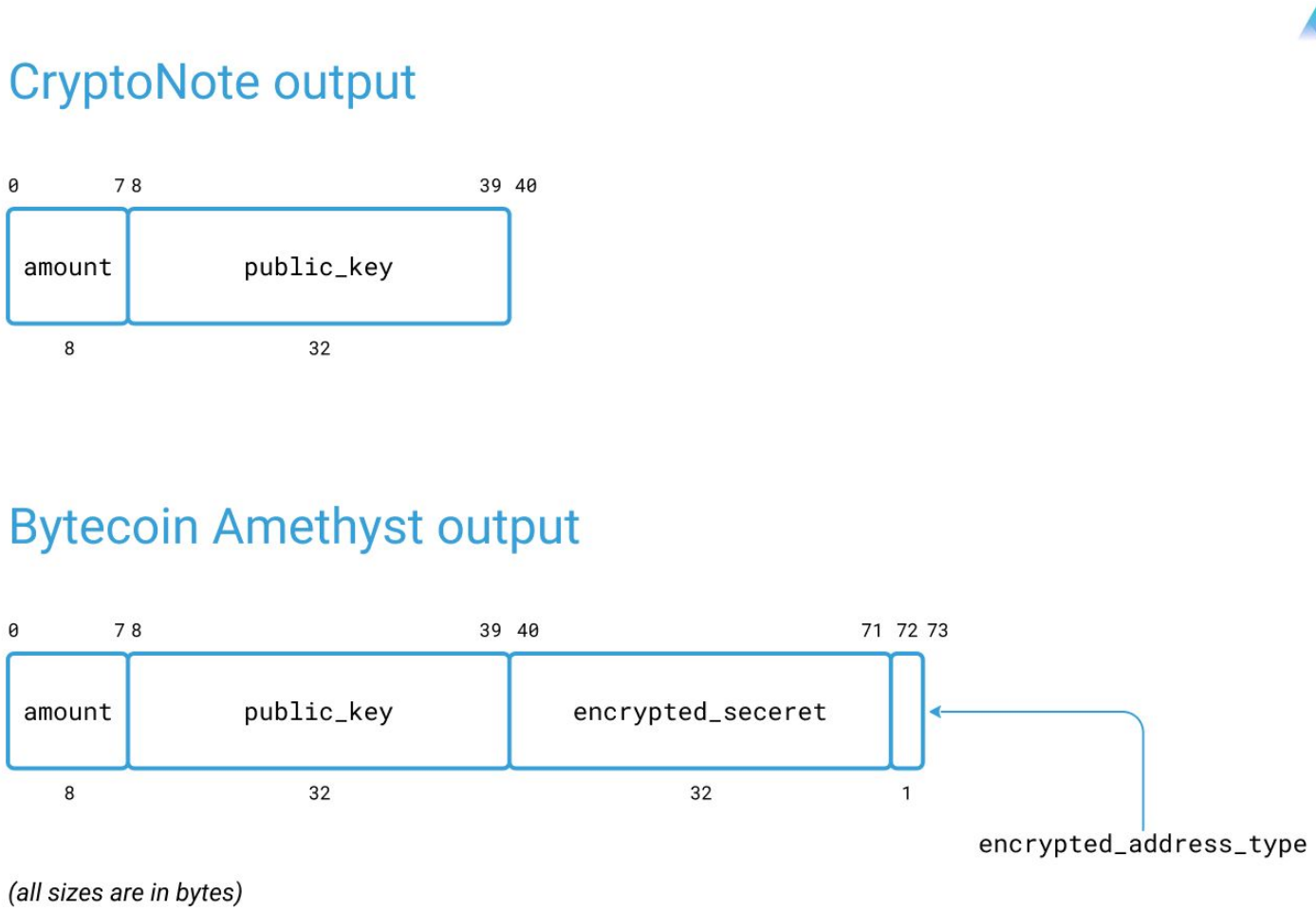

Fig. 4.2. Comparison of transaction output structure in CryptoNote and Bytecoin Amethyst

Therefore, the size of every output will increase by 33 bytes.

For each output i address type is encoded and decoded as the following:
`encrypted_address_type(i) = (H(o_i) & 255) xor address_tag`
where:
$o_i = H(h_t, v_s, i)$ (in the code referred to as `output_seed`),
address_tag is 0 for legacy addresses and 1 for ametyst addresses.

4.3. tx.version >= amethyst, legacy address

Wallet's account is a combination of spend secret key s and hash value v_s which is used as a source to deterministically generate secret view key v.

When Alice sends funds to Bob's address (V, S), she forms transaction outputs as the following:

1. Calculate $h_t = H(\text{tx.inputs, tx.version})$, and then for each output i:
2. $D_i = H_s(h_t, v_s, i) * G$ (in the code referred to as `output_shared_secret`)
3. $P_i = S + H_s(D_i, h_t, i) * G$
4. $Q_i = H_s(h_t, v_s, i) * V$ (here $Q_i = v * D_i$, however view key v of the recipient is unknown)
5. Tuple (P_i, Q_i) are the public keys of output i.

To receive the funds Bob scans all the transactions as the following:

6. For each transaction he calculates $h_t = H(\text{tx.inputs, tx.version})$, and then for each output i:
7. $D_i = v^{-1} * Q_i$
8. $S' = P_i - H_s(D_i, h_t, i) * G$

9. If S' equals to Bob's public spend key S it means Bob is the recipient of the output. Thus, Bob increases his balance by the corresponding amount.

To spend an output that was previously received by Bob and sent coins to Carol, he does the following:

10. Using his secret keys v and s, calculates the secret to P_i:
    $D_i = v^{-1} * Q_i$
    $x_i = s + H_s(D_i, h_t, i)$, and it is easy to see that $P_i = x_i * G$ (see also 3.)

11. Bob calculates key image: $I = x_i * H_p(P_i)$ and put it, the amount and a reference to the corresponding output into the input of his transaction to Carol.

12. Bob decreases his balance by the amount of the used output.

13. Bob forms outputs in his transaction to Carol according to (1-5). Then he signs the transaction and broadcasts it.

This scheme has a feature: anyone, who gets Alice's secret hash v_s is able to reveal all recipient addresses (V, S) for any Alice transaction:

14. $D_i = H_s(h_t, v_s, i) * G$
15. $S = P_i - H_s(D_i, h_t, i) * G$
16. $V = H_s(h_t, v_s, i)^{-1} * Q_i$

However, in that case the problem is to identify Alice's transactions among all others. Not knowing this one would get random useless addresses in (14-16) that are indistinguishable from real addresses of Alice's recipients. It could be partly resolved by decoding encrypted_address_type as for Alice's transactions that field after decoding should give a limited set of correct values {0, 1}. Unfortunately, it might be a false positive: `encrypted_address_type` for an arbitrary transaction could be decoded into a random value that *looks* correct.

In this scheme key image calculation also requires secret spend key s, like in CryptoNote. It means that wallet auditing problem i.e. the ability to calculate exact balance without right to spend coins, is also unsolved.

4.4. tx.version >= amethyst, amethyst address

Constant H

This cryptographic scheme requires a new constant — point H, the element of group **G**. The order of H is unknown. In other words, H is a fixed public key with corresponding secret key is guaranteed to be unknown, and the probability of finding a secret part of H is negligible:
H = y * G, where y is unknown.

H could be defined like suggested in [8]:
$H = H_p(G)$ = to_point(cn_fast_hash(G))

In Bytecoin H is hardcoded rather than calculated, and no comments can be found regarding its nature:

bytecoin/src/crypto/crypto_helpers.hpp, line 67

```
constexpr P3 H{ge_p3{{7329926, -15101362, 31411471, 7614783, 27996851, -3197071, -11157635, -6878293, 466949, -7986503},
        {5858699, 5096796, 21321203, -7536921, -5553480, -11439507, -5627669, 15045946, 19977121, 5275251},
        {1, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {23443568, -5110398, -8776029, -4345135, 6889568, -14710814, 7474843, 3279062, 14550766, -7453428}}};
```

However, searching for this sequence resulted in the fact ([9]) that Bytecoin uses the same constant as used in Monero for RingCT, which method of calculation and rationale is discussed in detail in [8].

Since H is an element of group **G**, it also means that H is also a generator of **G** like the base point G.
It means that $\forall\ x \in [1, p-1],\ x * H \in G$

Unlinkable addresses

In CryptoNote each wallet (i.e. a pair of secret keys) has only one public address that is used to receive funds.

This cryptographic scheme allows the creation of an unlimited number of public addresses for each account (i.e. set of secret keys). And also:

1) addresses are being generated in a deterministic way;
2) they cannot be linked, i.e. an eavesdropper cannot conclude they belong to the same account;
3) transaction scanning and accounting for N unlinkable addresses is computationally easier than scanning and accounting for N separate accounts.

Wallet's account is a combination of spend secret key s and hash value v_s like in the previous scheme. Here, however, secret hash v_s is used to deterministically generate additional audit key a, as well as the secret view key v.

Generation of i-th unlinkable address goes as the following (Fig. 4.4.2):

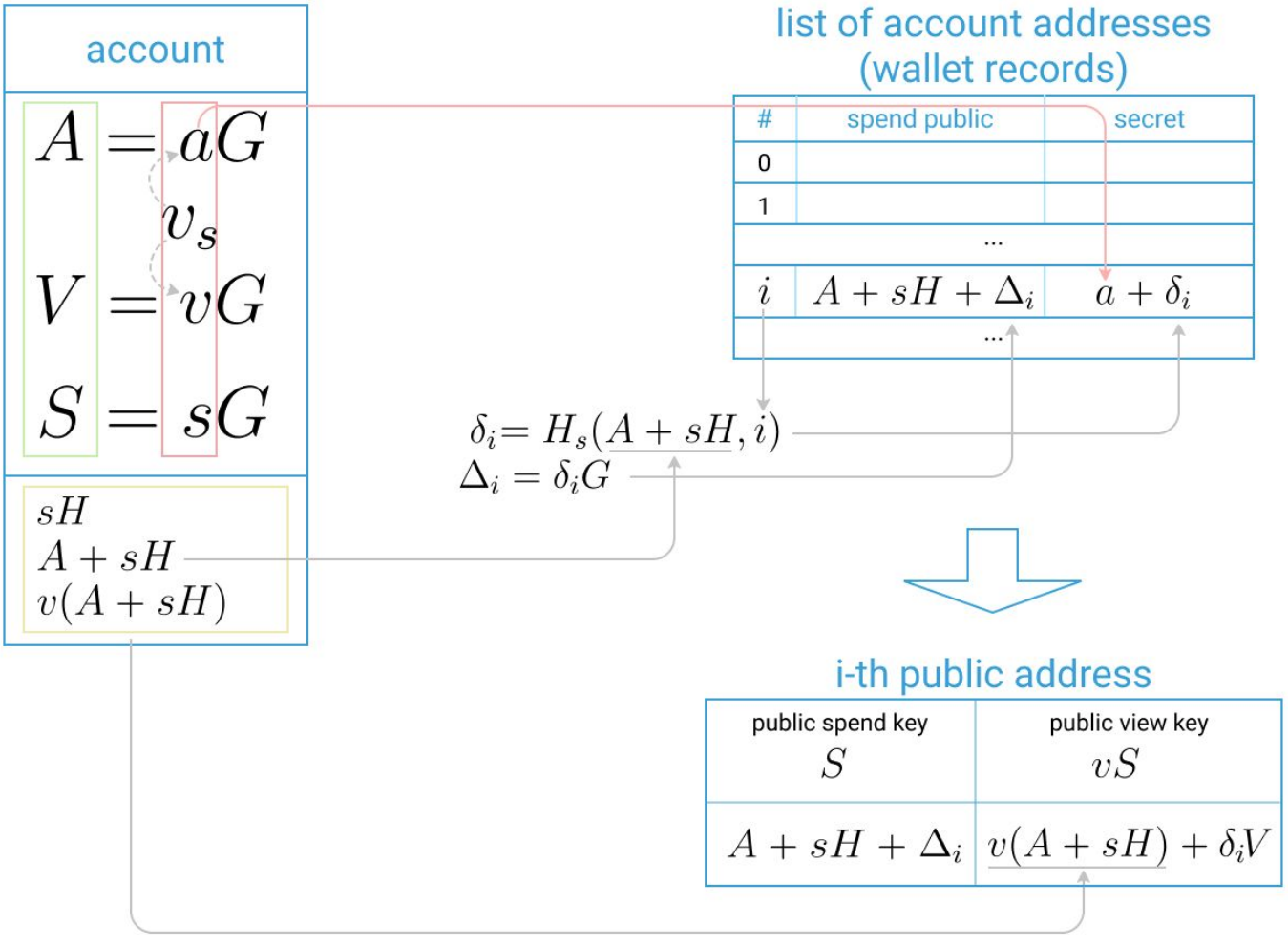

Fig. 4.4.2. Generation of amethyst addresses for Bytecoin account (yellow stroke highlights pre-calculated values which disclosure would not reveal secrets v, s and v_s)

1. Calculate $\delta = H_s(A + s * H, i)$, $\Delta = \delta * G$
2. $S = A + s * H + \Delta$
3. $V = v * S = v * (A + s * H + \Delta) = v * (A + s * H) + \delta * V$
4. pair (V, S) = (v * S, S) is the i-th public address of the account.

Note that to calculate public unlinked addresses one needs only to know these values:

- A
- V
- s * H
- v * (A + s * H)

Values s * H и v * (A + s * H) are pre-calculated during account initialization and being considered safe, meaning they cannot be used to calculate s or v.

Generated public addresses are stored locally in a container optimized for searching by S.

Constructing outputs for sending funds

When Alice sends coins to Bob's address (V, S) she constructs transaction outputs as the following (Fig. 4.4.3).

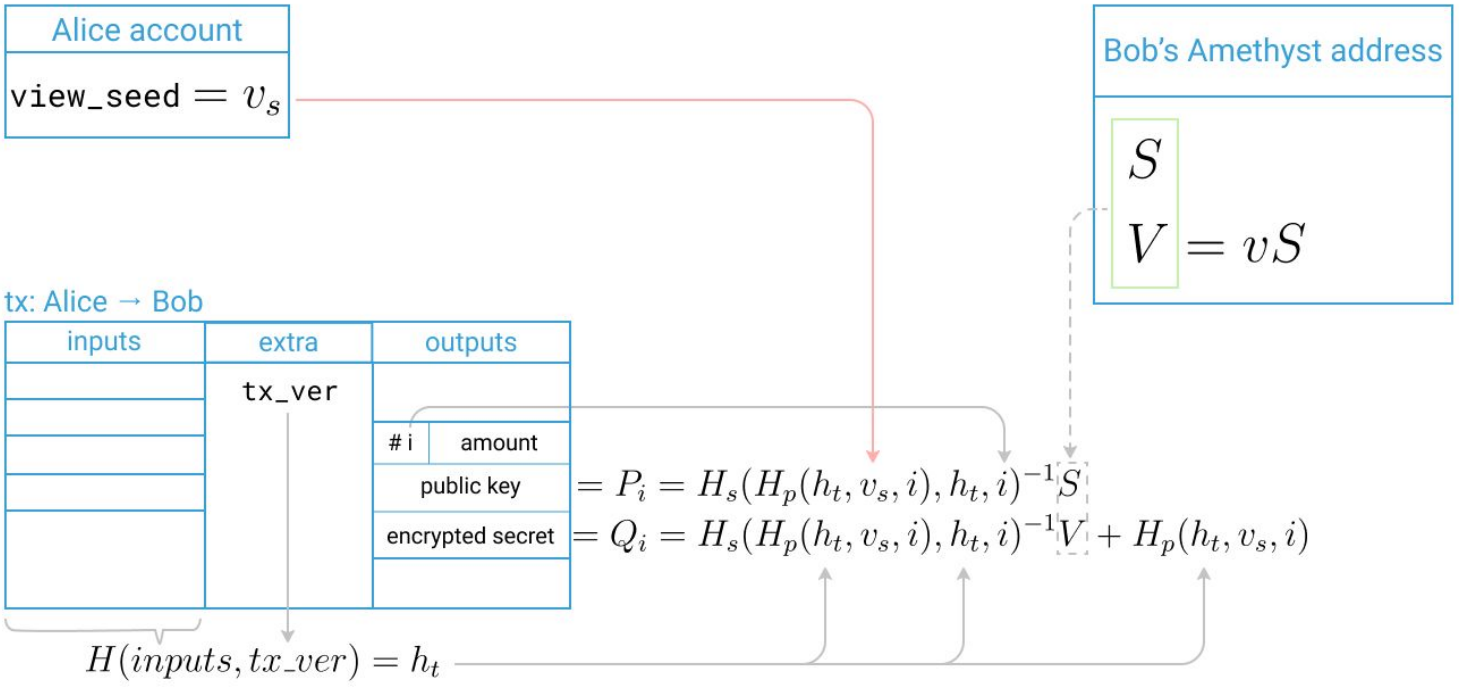

Fig. 4.4.3. Alice constructs transaction outputs when sending funds to an Amethyst address

1. Calculates h_t = H(tx.inputs, tx.version), then for each output i:
2. $P_i = H_s(H_p(h_t, v_s, i), h_t, i)^{-1} * S$
3. $Q_i = H_s(H_p(h_t, v_s, i), h_t, i)^{-1} * V + H_p(h_t, v_s, i)$
4. So we get (P_i, Q_i), the public keys for the output i.

Accounting of incoming transfers

In order to receive funds Bob scans over all blockchain transactions and checks them as follows (Fig. 4.4.4).

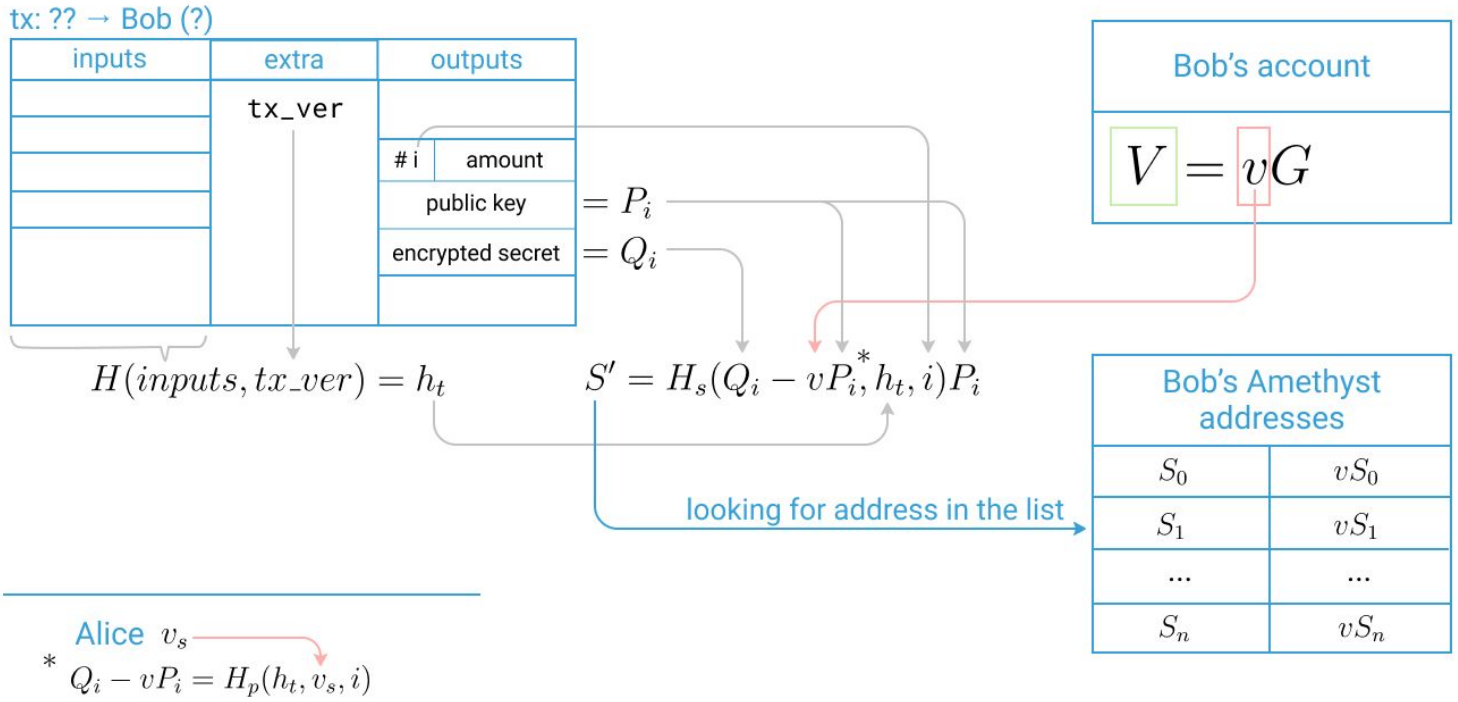

Fig. 4.4.4. Analysing transaction outputs for incoming transfers

1. For each output i using secret view key v Bob calculates:
   $K = H_p(h_t, v_s, i) = Q_i - v * P_i$ (`output_shared_secret` in the code)
2. $S' = H_s(K, h_t, i) * P_i$
3. Then he searches for S' in the list of his addresses. If found, the output is targeted to Bob and he increases the balance for the corresponding address.

It is important to mention that besides the list of unlinkable addresses (or keys for their generation) secret view key v is necessary to identify incoming transfers and take them into account in order to correctly calculate the balance.

Construction of outgoing transfers

If Bob wants to spend previously received output i and send coins to Carol, he does the following:

1. $K = H_p(h_t, v_s, i) = Q_i - v * P_i$ (`output_shared_secret` в коде)
2. $x_i = H_s(K, h_t, i)^{-1} * (a + \delta)$
3. $X_i = x_i * G + H_s(K, h_t, i)^{-1} * s * H$
4. Calculates key image $I = x_i * H_p(X_i)$
5. Bob decreases the balance corresponding to his address $S = H_s(K, h_t, i) * P_i$ by the amount of the output i.
6. Then Bob constructs transaction outputs, signs and broadcasts it.

It's easy to see that for key image calculation both secret keys — v and a — are required.

This cryptographic scheme has an important feature: **an ability to calculate key image** (and therefore an ability to identify own outgoing transfers, thus to calculate the balance) **without using spend secret key s**.

Recipients' addresses disclosure for the needs of audit

Like the previous one, this cryptographic scheme allows to obtain recipients' addresses from transaction outputs if the sender's secret hash v_s is known.

This can be done as the following (Fig. 4.4.5):

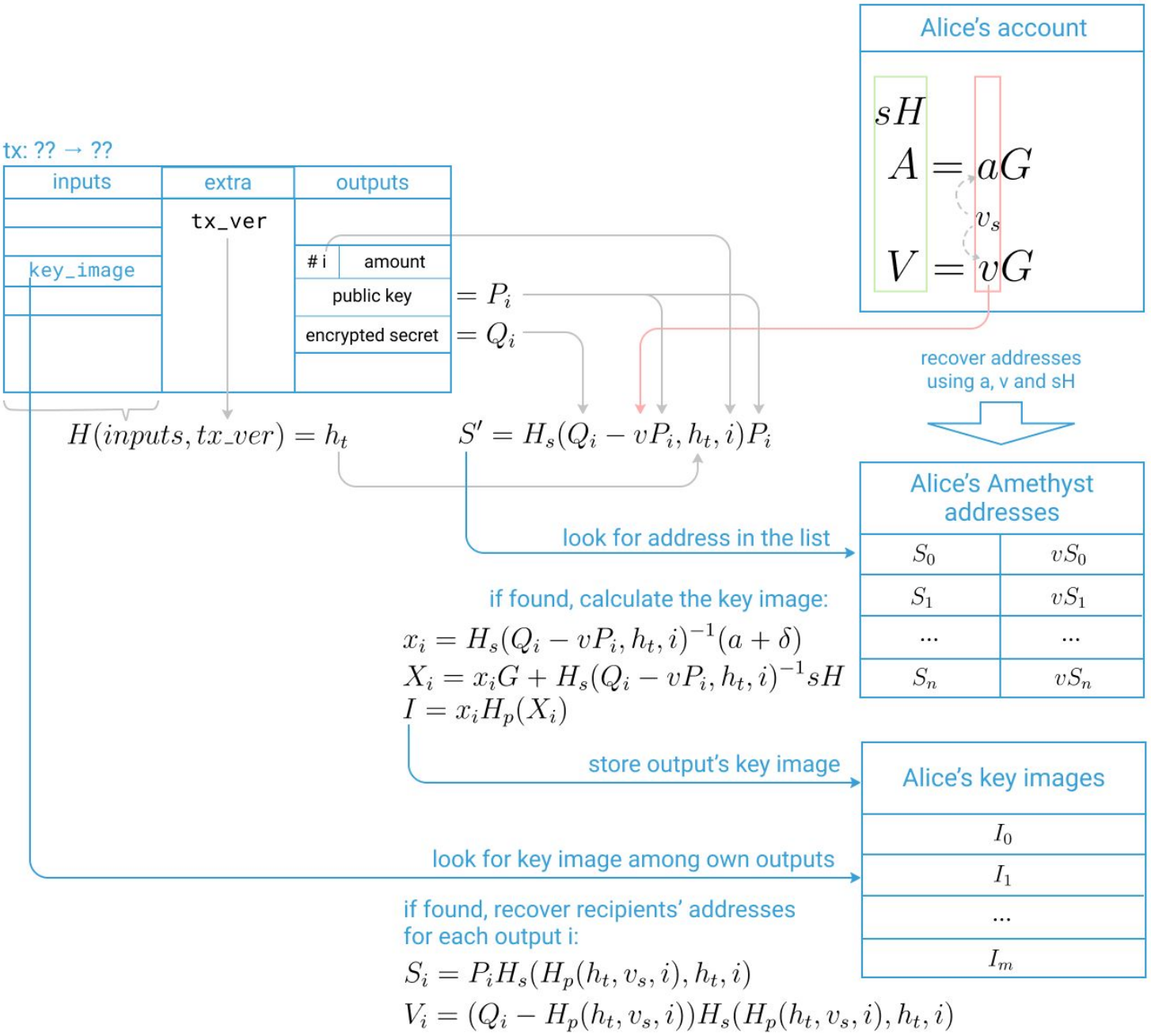

Fig. 4.4.5. Using v_s an auditor extracts all the information about incoming/outgoing transfers, balance and recipients' addresses

Assume, Alice gave v_s and s * H to Carol. Then Carol:

1. Restores secret keys v and a, restores a list of Alice's unlinkable addresses.
2. Scans blockchain and for each output i of every transaction checks whether it is targeted to Alice by searching
   $S' = H_s(Q_i - v * P_i, h_t, i) * P_i$
   among all Alice's addresses.
3. If found, Carol increases the balance of corresponding address, then, using a and v calculates key image (see above) and stores it locally.
4. If a transaction contains Alice's key image in its input, it means this transaction was sent by Alice. Carol recovers all recipients' addresses for all outputs:
   $S = P_i * H_s(H_p(h_t, v_s, i), h_t, i)$
   $V = (Q_i - H_p(h_t, v_s, i)) * H_s(H_p(h_t, v_s, i), h_t, i)$
   and decreases the balance of Alice's corresponding address by the amount of the corresponding input.

Thus, partially disclosing accounts data one can give to a third party different levels of access to the wallet.

To generate all unlinkable addresses one will need:

- A
- V
- s * H
- v * (A + s * H)

To identify only incoming transfers:

- A
- v
- s * H

For wallet's auditing (to calculate the balance for own addresses, without disclosing recipients' addresses):

- a
- v
- s * H

For wallet's auditing (to calculate the balance for own addresses, disclosing recipients' addresses):

- v_s
- s * H

4.5. Ring signatures comparison in CryptoNote and Bytecoin Amethyst

Data structures and signatures sizes

As was mentioned above, wallet auditing in Bytecoin Amethyst without disclosing a secret spend key was implemented at the price of additional data in each transaction output. Compared to the original protocol, the size of each output in amethyst transactions was increased by 33 bytes: one additional public key and one byte for address type identification.

In CryptoNote each transaction input is signed separately. For each output public key P_i which an input of a transaction refers to, a pair of scalars (r, c) with a total size of 64 bytes is added to the signature of that transaction. (An input may refer to several outputs but only one of them is the real one, others are used to increase anonymity.)

Therefore, if a transaction has N_{inputs} inputs, and each of them refers to N_{mixins} output then the total size of the ring signature in bytes can be expressed as:
$S = 32 * 2 * N_{mixins} * N_{inputs}$
Minimum signature size for a transaction is 64 bytes (one input that refers to an output directly).

In Bytecoin Amethyst the ring signature is calculated for the whole transaction and its structure is much more complicated (Fig. 4.5.1):

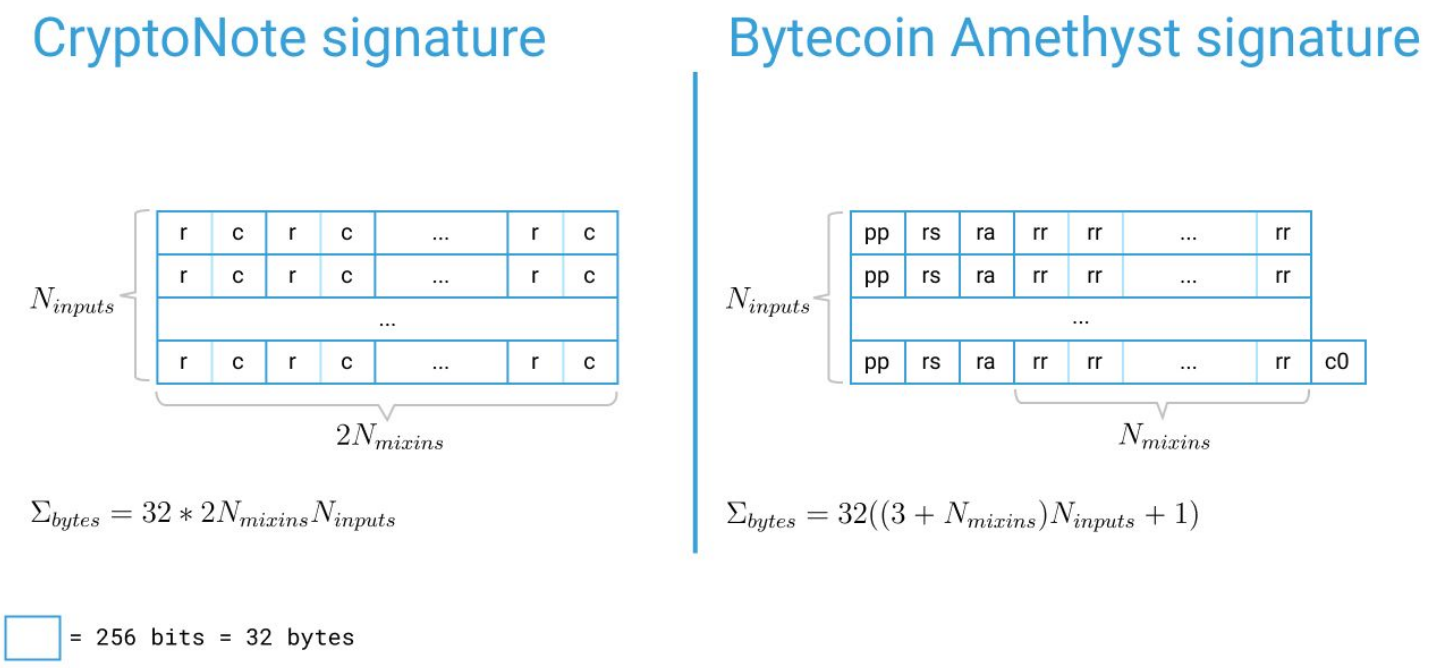

Fig. 4.5.1. Ring signatures structure comparison in CryptoNote and Bytecoin Amethyst

For each input a point and 2 + N_{mixins} scalars are added to the signature. Finally, one more scalar is added. Total signature size in bytes is:
$S = 32 * ((3 + N_{mixins}) * N_{inputs} + 1)$

Minimum signature size for a transaction is 160 bytes (one input that refers to an output directly).

It's easy to see that both functions grow proportional to multiplication $N_{mixins} * N_{inputs}$
To make the comparison more clear let's calculate the values for both functions for the most probable values of N_{mixins} and N_{inputs} and put them into a table (Fig. 4.5.2).

	N outputs						
N inputs	1	2	3	4	5	10	20
1	150%	50%	17%	0%	-10%	-30%	-40%
2	125%	38%	8%	-6%	-15%	-33%	-41%
3	117%	33%	6%	-8%	-17%	-33%	-42%
4	113%	31%	4%	-9%	-18%	-34%	-42%
5	110%	30%	3%	-10%	-18%	-34%	-42%
6	108%	29%	3%	-10%	-18%	-34%	-42%
7	107%	29%	2%	-11%	-19%	-34%	-42%
8	106%	28%	2%	-11%	-19%	-34%	-42%
9	106%	28%	2%	-11%	-19%	-34%	-42%
10	105%	28%	2%	-11%	-19%	-35%	-42%
11	105%	27%	2%	-11%	-19%	-35%	-42%
12	104%	27%	1%	-11%	-19%	-35%	-42%
13	104%	27%	1%	-12%	-19%	-35%	-42%
14	104%	27%	1%	-12%	-19%	-35%	-42%
15	103%	27%	1%	-12%	-19%	-35%	-42%
16	103%	27%	1%	-12%	-19%	-35%	-42%
17	103%	26%	1%	-12%	-19%	-35%	-42%
18	103%	26%	1%	-12%	-19%	-35%	-42%
19	103%	26%	1%	-12%	-19%	-35%	-42%
20	103%	26%	1%	-12%	-20%	-35%	-42%

Fig. 4.5.2. Difference in Bytecoin signature size comparing to CryptoNote (in percent relative to CryptoNote signature size; green highlighting shows where Bytecoin outperforms CryptoNote)

It's easy to see, in a case of direct spending ($N_{mixins} = 1$) or when only one foreign output is mixed in ($N_{mixins} = 2$), Bytecoin signature is greater than CryptoNote for up to 150%.
As N_{mixins} grows further, Bytecoin signature size becomes less than CryptoNote.

It is worth mentioning that in order to increase anonymity Bytecoin developers since the version 3.4.0 Amethyst set the minimum allowed number of mixed in outputs to three [10]. Taking this into account, Bytecoin signatures would be less in size.

The complexity of signature verification

In addition to the size of the ring signature, which directly affects the size of the blockchain, another important characteristic is the computational complexity of its verification. It determines such important parameters of the cryptocurrency system as, for example, the speed of synchronization with the network of new nodes and the computational load on the network with a large flow of transactions.

The complexity of signing verification for CryptoNote and Bytecoin could be easily compared practically by writing a test that generates and then verifies a large number of signatures with the given N_{mixins} and N_{inputs}. However, since further in the article we will consider schemes not yet implemented in practice, it will be logical to evaluate the complexity of these schemes empirically, according to the number of cryptographic operations used.

CryptoNote and Bytecoin use several basic primitives (see section 2). In the table, in Fig. 4.5.3. the typical execution time of the most commonly used primitives is shown on a modern middle-end computer with a Core i5-6500 processor (for the sake of comparison, the original source code compiled in Microsoft Visual Studio 2017 with all possible speed optimizations was used).

operation		typical execution time (mcs)	
		CryptoNote	Bytecoin
points addition	A + B	0.52	0.56
scalar multiplication	a * B	122.6	132.1
hash-function Hp	Hp(x)	12.46	15.6
hash-function Hs	Hs(x)	1.4	3
inverse	s^{-1}	(not used)	15.6

Fig. 4.5.3. Typical execution time for most commonly used cryptographic primitive

The results obtained from tests in Bytecoin and CryptoNote are in good agreement. It is easy to see that the largest contribution will be made by the scalar multiplication, and to a lesser extent, by the calculation of the hash function H_p, while the addition of two points and the calculation of the hash function H_s will not significantly affect the complexity.

Consider the CryptoNote ring signature verification procedure (Fig. 4.5.4, the signature generation procedure is discussed in detail in [1] and will not be considered here).

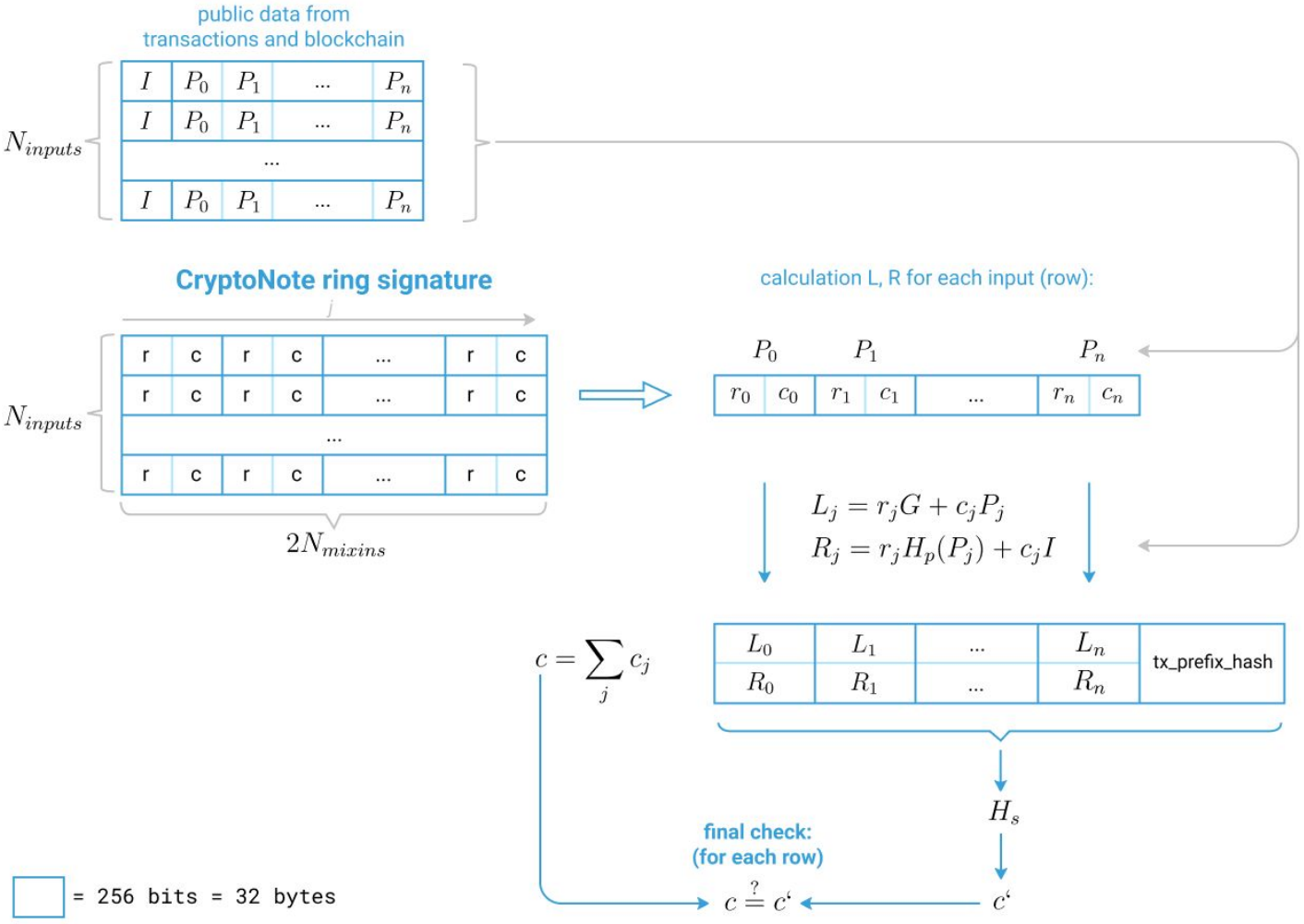

Fig. 4.5.4. CryptoNote ring signature verification

As already noted, in CryptoNote, each transaction input is signed separately, respectively, and each input is also checked separately. Therefore, the verifier for each transaction input verifies the corresponding line (in the figure) of the signature as follows.

1. For each pair of r_j and c_j values, using the key-image of input I and the public key P_j of the next output that this input refers to, the values are calculated:
2. $L_j = r_j * G + c_j * P_j$
   $R_j = r_j * H_p(P_j) + c_j * I$
   (index j ranges from 0 to N_{mixins})

3. Calculate the sum *c* of all c_j.

4. Calculate hash c`=Hs(tx_prefix_hash, L_0 .. L_n, R_0 .. R_n)
   where tx_prefix_hash is the hash of a transaction prefix (without signature).

5. Checking equality c` = *c*. If it is satisfied, then the ring signature is valid.

Let us estimate the number of multiplications and the calculations of the hash Hp.
Each calculation of L_j and R_j requires two scalar multiplications. The number of pairs L_j, R_j corresponds to the number of mixed outputs N_{mixins} for each input. Therefore, we have:
$O(*) = N_{inputs} * 4 * N_{mixins}$

Moreover, the hash function H_p is used once for each calculation of R_j, therefore:
$O(H_p) = N_{inputs} * N_{mixins}$

Now, consider the ring signature verification algorithm in Bytecoin Amethyst (Fig. 4.5.5).

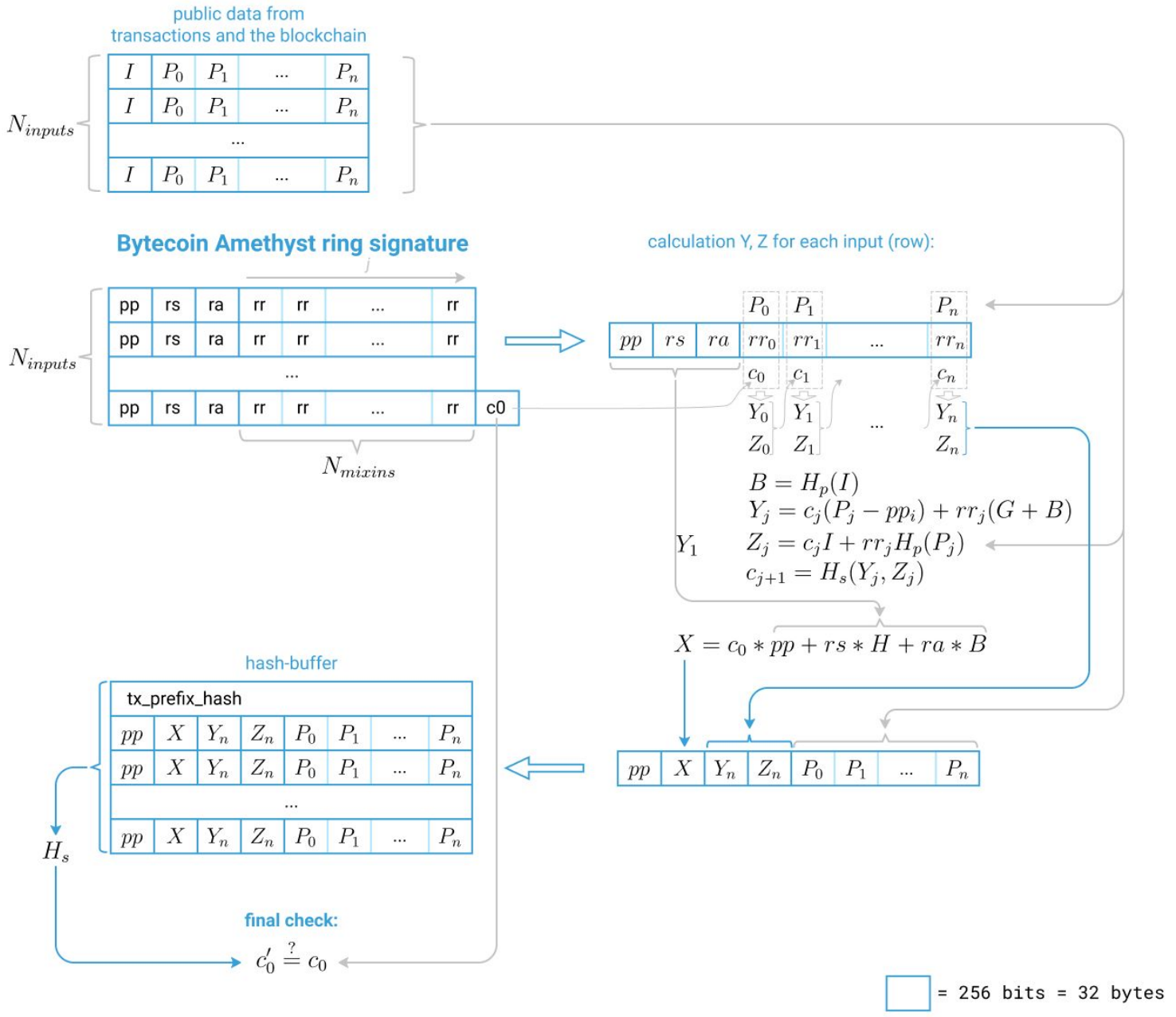

Fig. 4.5.5. Bytecoin Amethyst ring signature verification

The entire signature is checked thoroughly, for all inputs at once. It happens like this:

1. The prefix hash of the transaction (without signature) is written to the hash buffer.
2. For each input i (signature row in the figure):
    a. Calculate X.
    b. The values of pairs are sequentially calculated (Y_0, Z_0) ... (Y_n, Z_n), where n = N_{mixins}
    c. The set (pp, X, Y_n, Z_n, P_0 ... P_n), where P_j outputs' public keys this input refers to, is added to the hash buffer.
3. Calculate H_s using the whole hash buffer and compare the result with c_0. If the result and c_0 are equal, the signature is valid.

Let us estimate the number of scalar multiplications and the calculations of the hash H_p.
Each calculation of Y_j and Z_j requires two scalar multiplications, plus a calculation of X requires three scalar multiplications. The number of pairs Y_j, Z_j corresponds to the number of mixed outputs N_{mixins} for each input.
Therefore, we have:
O(*) = N_{inputs} * (3 + 4 * N_{mixins})

In this case, the hash function H_p is used once for each calculation of Z_j and once for the calculation of B for each of the inputs, therefore:
$O(H_p) = N_{inputs} * (N_{mixins} + 1)$

To clearly compare the computational complexity of both algorithms on standard data, we introduce the following metric. Add up the number of scalar multiplication operations and H_p calculation operations with weights proportional to the characteristic execution time of these operations:
$O(total) = 130 * O(*) + 15 * O(H_p)$
Then compare the relative results for CryptoNote and Bytecoin in percent (Fig. 4.5.6).

	N outputs						
N inputs	**1**	**2**	**3**	**4**	**5**	**10**	**20**
1	76%	38%	25%	19%	15%	8%	4%
2	76%	38%	25%	19%	15%	8%	4%
3	76%	38%	25%	19%	15%	8%	4%
10	76%	38%	25%	19%	15%	8%	4%
15	76%	38%	25%	19%	15%	8%	4%
20	76%	38%	25%	19%	15%	8%	4%

Fig. 4.5.6. Computational complexity of Bytecoin Amethyst ring signature verification compared to CryptoNote (no dependency on N_{inputs})

As you can see, Bytecoin signature verification is a much more time-consuming operation.

However, as noted above, in Bytecoin from version 3.4.0 Amethyst, in order to increase anonymity, the minimum number of mixed outputs is set to 3 [10], so the worst practical value will not exceed 25% (in theory).

To summarize:

1. The size of each output is increased by the public key Q that is 32 bytes.

2. The size of the ring signature varies in comparison with the size of the CryptoNote signature depending on the number of outputs mixed (with a large number it turns out to be less):
   $S = 32 * ((3 + N_{mixins}) * N_{inputs} + 1)$

3. Computational complexity is higher than in CryptoNote and significantly depends on the number of outputs to be mixed:
   $O(*) = N_{inputs} * (3 + 4 * N_{mixins})$
   $O(H_p) = N_{inputs} * (N_{mixins} + 1)$

5. Option 2 of 3: CryptoNote auditing researched by Anton Sokolov

In the fall of 2019, a series of essays [11, 12, 13, 14, 15, 18] on the audit problem in CryptoNote and possible solutions to it authored by Anton Sokolov were published on Medium.com. It theoretically considers several options for modifying the original protocol in such a way as to solve the audit task for a third party.

We consider the last of them [15] as the most optimized. We will abbreviate it as "AS".

Note: for consistency, spend keys will continue to be denoted by the letters s and S, view keys by the letters v and V, despite the fact that b, B and a, A are used in the original works, respectively.

Outputs construction

A wallet account is represented by a set of not two secret keys, like CryptoNote, but three secret keys {v, s, d}: view-, spend- and audit-keys, respectively.

The set of three public keys {V, S, D} represents the public address of this account.

Alice, sending money to Bob, acts as follows (Fig. 5.1).

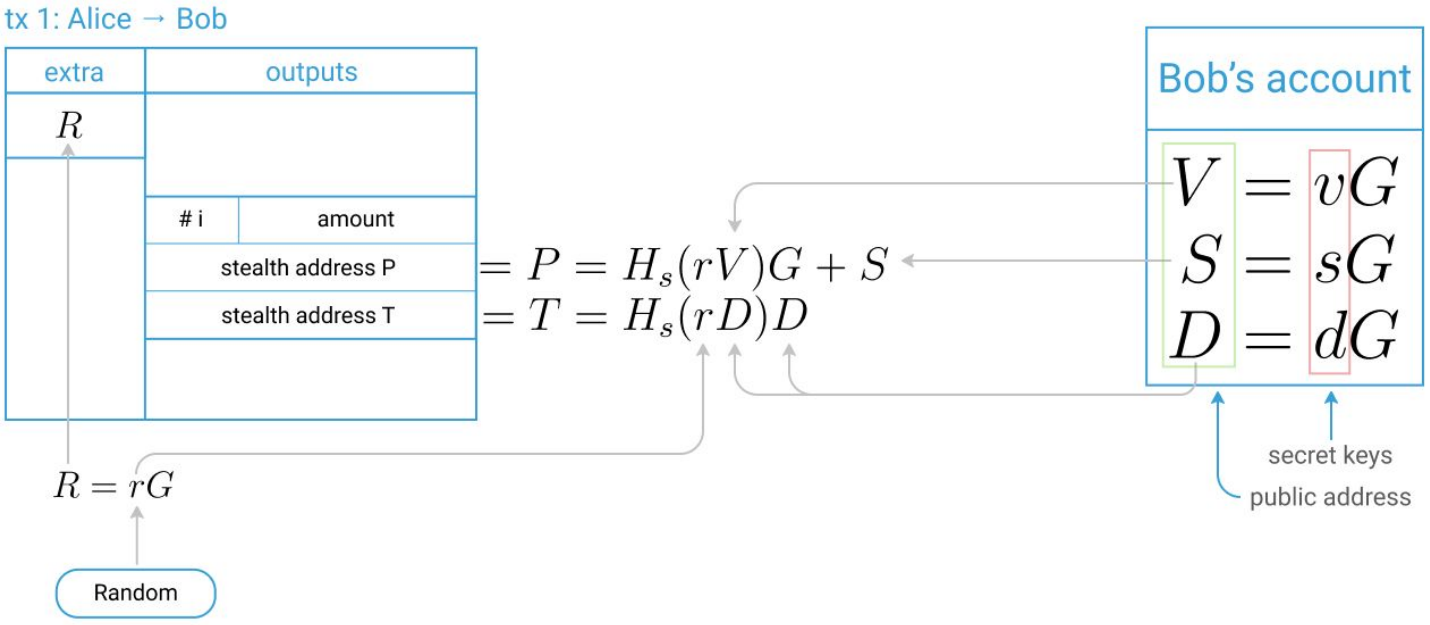

Fig. 5.1. Transaction outputs' construction in AS scheme

1. Bob publishes his address, so Alice knows his public keys V, S, and D.

2. Like in CryptoNote, Alice chooses a random secret transaction key r and calculates the public key R = r * G, which writes to the special field extra transactions.

3. For each output, Alice calculates not one, but two stealth addresses P and T. The first is calculated similarly to CryptoNote:
   P = H_s (r * V) * G + S
   The second is different:
   T = H_s (r * D) * D
   The serial number of the output is not used in the original work, however, it is reasonable to assume that this is done for simplification and in practice it will be one of the arguments of the H_s function by analogy with CryptoNote (see section 3).

4. Alice signs and sends the transaction.

An eavesdropper will not be able to associate P and T with Bob's address.

Recognition of incoming payments and construction of inputs

To accept and spend money, Bob acts as follows (Fig. 5.2).

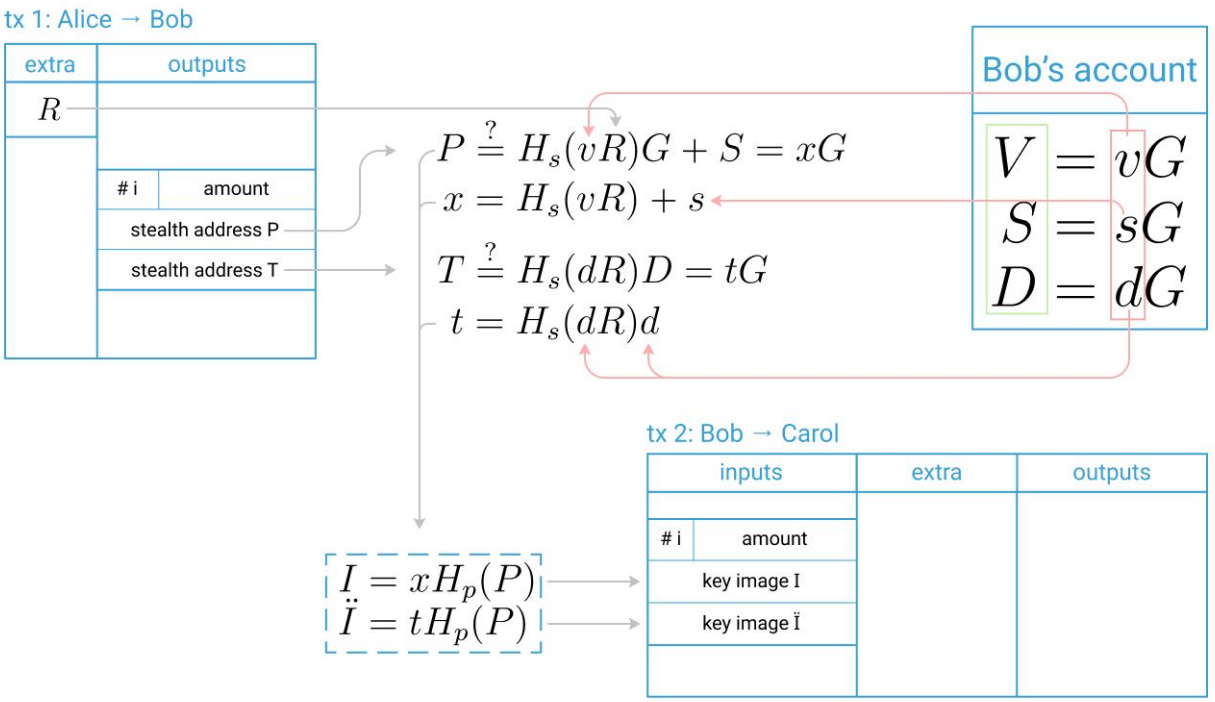

Fig. 5.2. Identification of incoming transfers and construction of an input for spending transaction

5. Using his private view key v, Bob compares the stealth address P of each output with a point:
   P '= H_s (v * R) * G + S
   (This step is similar to CryptoNote)

6. In case equality is fulfilled, this means that this output is addressed to Bob. He increases his balance by the value of the nominal output.

7. If Bob needs to transfer the received funds to Carol, using his secret spend- and audit-keys s and d, he calculates two key images: I and Ï. The first is similar to CryptoNote:
   I = x * H_p (P)
   and optional:
   t = H_s (d * R) * d
   Ï = t * H_p (P)
   Then he writes them, the amount and a link to the corresponding output into the input of his transaction for Carol.

8. Then Bob constructs the transaction outputs for Carol, reduces his balance in proportion to amount of spent outputs, signs the transaction and sends.

As you can see here, just like in CryptoNote, the third party — the Auditor — having only the secret view key v, will be able to recognize only incoming transactions.

Auditing

If the Auditor also has a secret audit key d, then he will be able to recognize Bob's outgoing payments and calculate his balance as follows:

9. For each incoming payment, the Auditor will calculate and store an additional key image Ï in the local storage:
   $t = H_s(d*R)*d$
   $Ï = t * H_p(P)$

10. If an additional key image Ï in the inputs of any of the blockchain transactions matches one of the saved ones, this will mean that this transaction is Bob's outgoing payment. The auditor will reduce the balance by the amount of the corresponding input.

Thus, the Auditor, having a set of keys {v, S, d}, will be able to recognize Bob's incoming and outgoing transfers on the blockchain and restore his correct balance. At the same time, the Auditor will not be able to spend money, because without the secret spend-key s, he will not be able to calculate the main key image I.

The problem is solved.

Ring signature

Applying the ideas from [16], the author was able to reduce the size of the signature compared to CryptoNote: now, for each input, only Nmixins + 1 scalar is stored in the signature (Fig. 5.3).

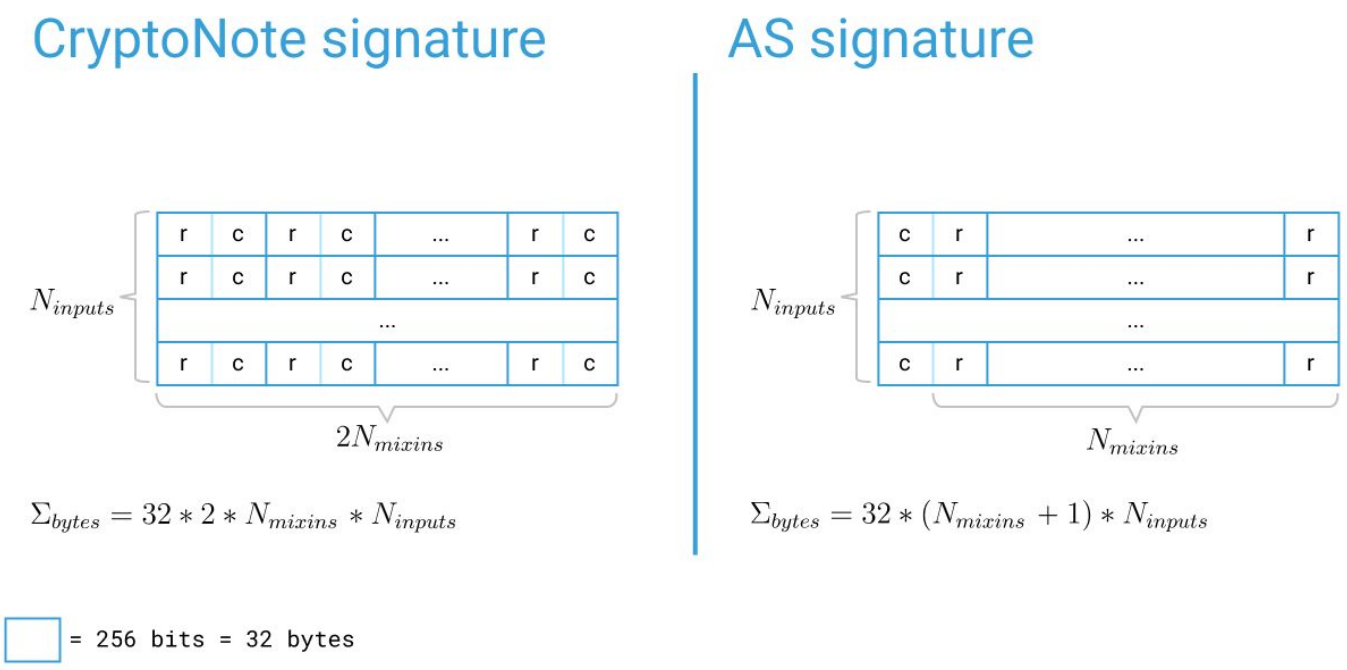

Fig. 5.3. Comparing AS ring signature size and structure with CryptoNote

Thus, the size of the signature is reduced by almost half.

Consider the computational complexity of its verification. The signature verification scheme is shown in Fig. 5.4.

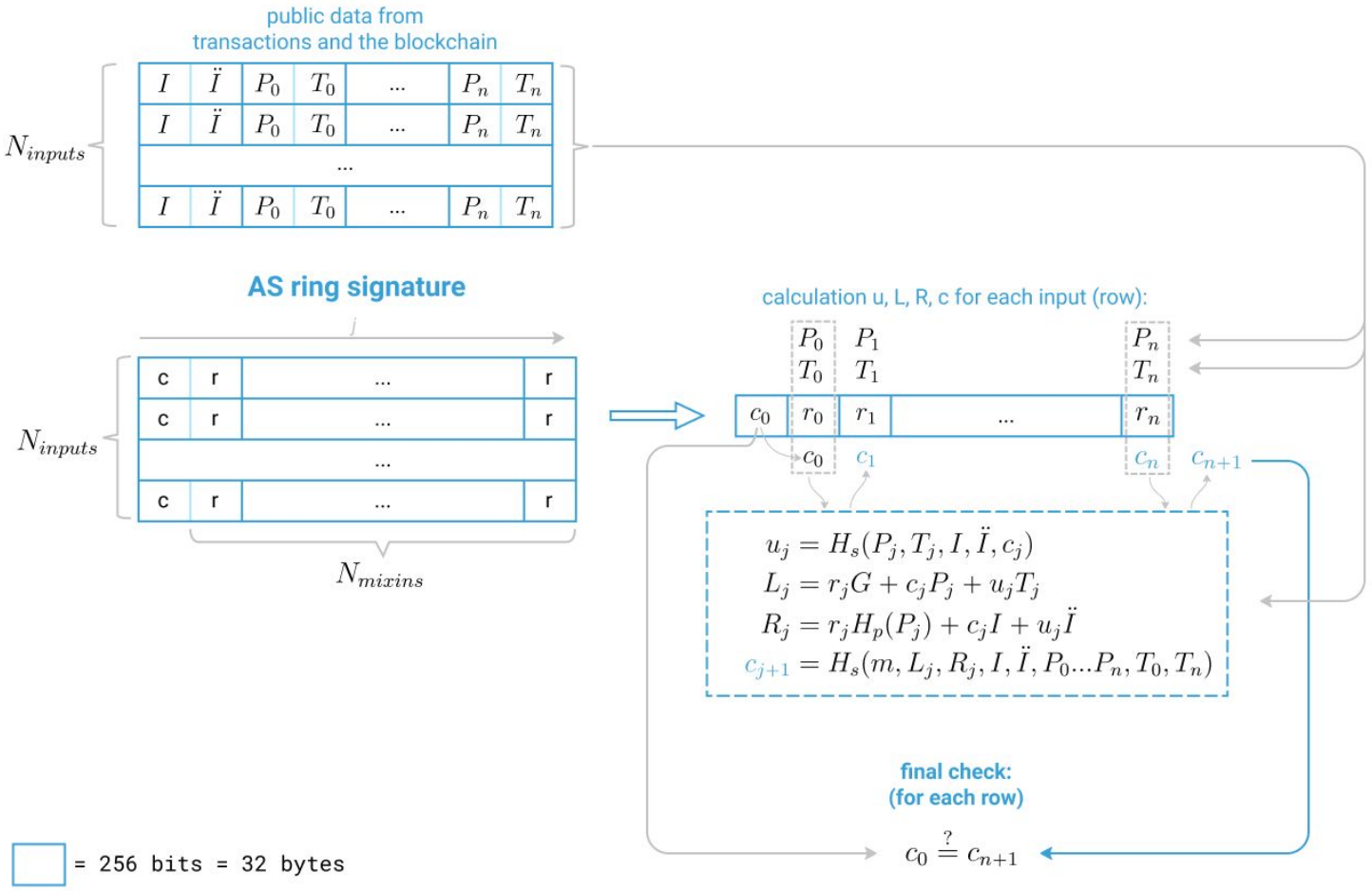

Fig. 5.4. Ring signature verification in AS scheme

This algorithm is very similar in structure to that used in CryptoNote (see Fig. 4.5.4). The check is carried out separately for each input and consists in sequentially calculating the values of u_j, L_j, R_j, c_j for all outputs used for mixing in this input. As a result, the cycle of c_j must be closed and the value c_{n+1} must coincide with c_0, in this case the signature is considered valid.

The computational complexity of the algorithm is determined by six scalar multiplications and one calculation of the hash function H_p per iteration, the number of which is the product of N_{inputs} * N_{mixins}.

Comparison with CryptoNote by data and computational load

1. The size of the address increases by 50%, because the address now represents a collection of three public keys: {V, S, D}, rather than two {V, S}.
   The encoded representation of the address will increase by about the same: for example, a standard Zano address containing two public keys takes 97 characters:
   `ZxD5UBX5PM3RTsEtTRd9ATUFxXyocoQzDRk3baVBahuWQJRK8QHTUT9GQM7jk7GoedK5B2nP4HxSPDpuLHvizpwj2q99bmz7t`
   A similar Zano address containing three public keys will have a length of about 141 characters:
   `ZxD5UBX5PM3RTsEtTRd9ATUFxXyocoQzDRk3baVBahuWQJRK8QHTUT9GQM7jk7GoedK5B2nP4HxSPDpuLHvizpwjcenhnGbhpJFLk8vkhJywHCcht4d9EKA7CHHav1H6QPpB1cLsTvPfj`

2. The size of each output is increased by an additional stealth address T that is 32 bytes.

3. The size of each input is increased by an additional key image Ï that is 32 bytes.

4. The size of the ring signature is smaller than in CryptoNote:
   $S = 32 * (N_{mixins} + 1) * N_{inputs}$

5. Computational complexity is higher than in CryptoNote:
   $O(*) = N_{inputs} * 6 * N_{mixins}$
   $O(H_p) = N_{inputs} * N_{mixins}$

6. Option 3 of 3: Audit through output mixing limit

Consider another way to implement auditing in CryptoNote.

Output attribute `mix_attr`

In the Zano project, which is the successor to CryptoNote, the transaction output has an additional 8-bit mix_attr attribute, and it is a rule restricting the use of outputs in mixing, depending on its value.
The structure of the outputs of CryptoNote and Bytecoin (Fig. 4.2) can now be extended with Zano (Fig. 6.1.).

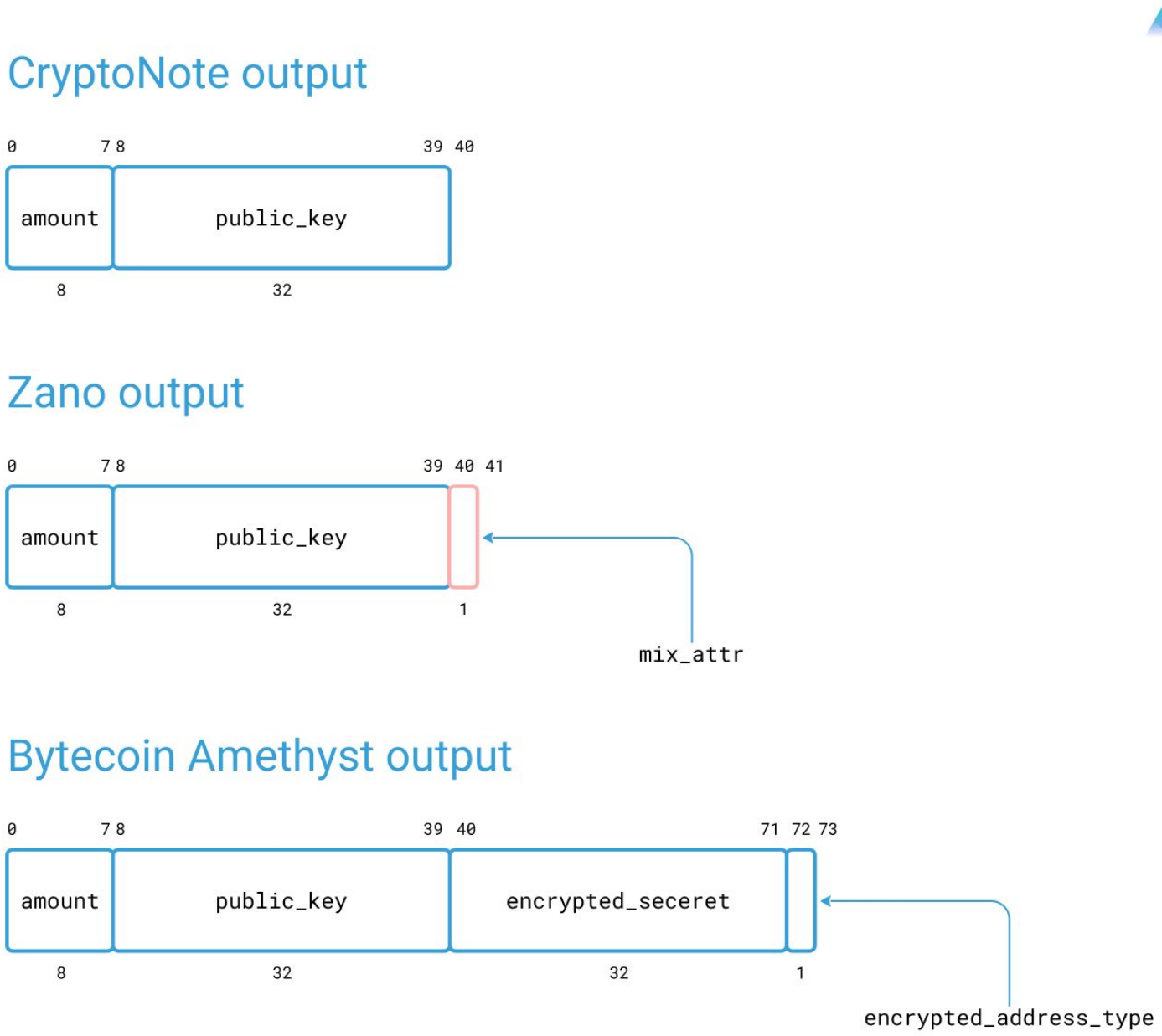

Fig. 6.1. Comparison of transaction output structure in CryptoNote, Zano and Bytecoin Amethyst

The rule restricting mixing according to mix_attr is this:

1. If mix_attr = 0, then there are no restrictions. This output can be used to mix with other outputs in any combination. This is the default value.
2. If mix_attr = 1, then this output can be spent only directly, without mixing, and cannot be used to mix with other outputs.
3. If mix_attr ≥ 2, then this output can be spent only by mixing others, while the total number of outputs used should not be less than the value of mix_attr.

The main feature of this innovation is item 3, which allows one to increase untraceability (untraceability of bonds) by eliminating situations where the output already used in mixing is spent by its owner directly (this was described in detail in [19]).

However, in the context of this article, we will be interested in item 2, that is, the situation where mix_attr = 1 and the output marked in this way can only be spent directly. This limitation is illustrated in Fig. 6.2.

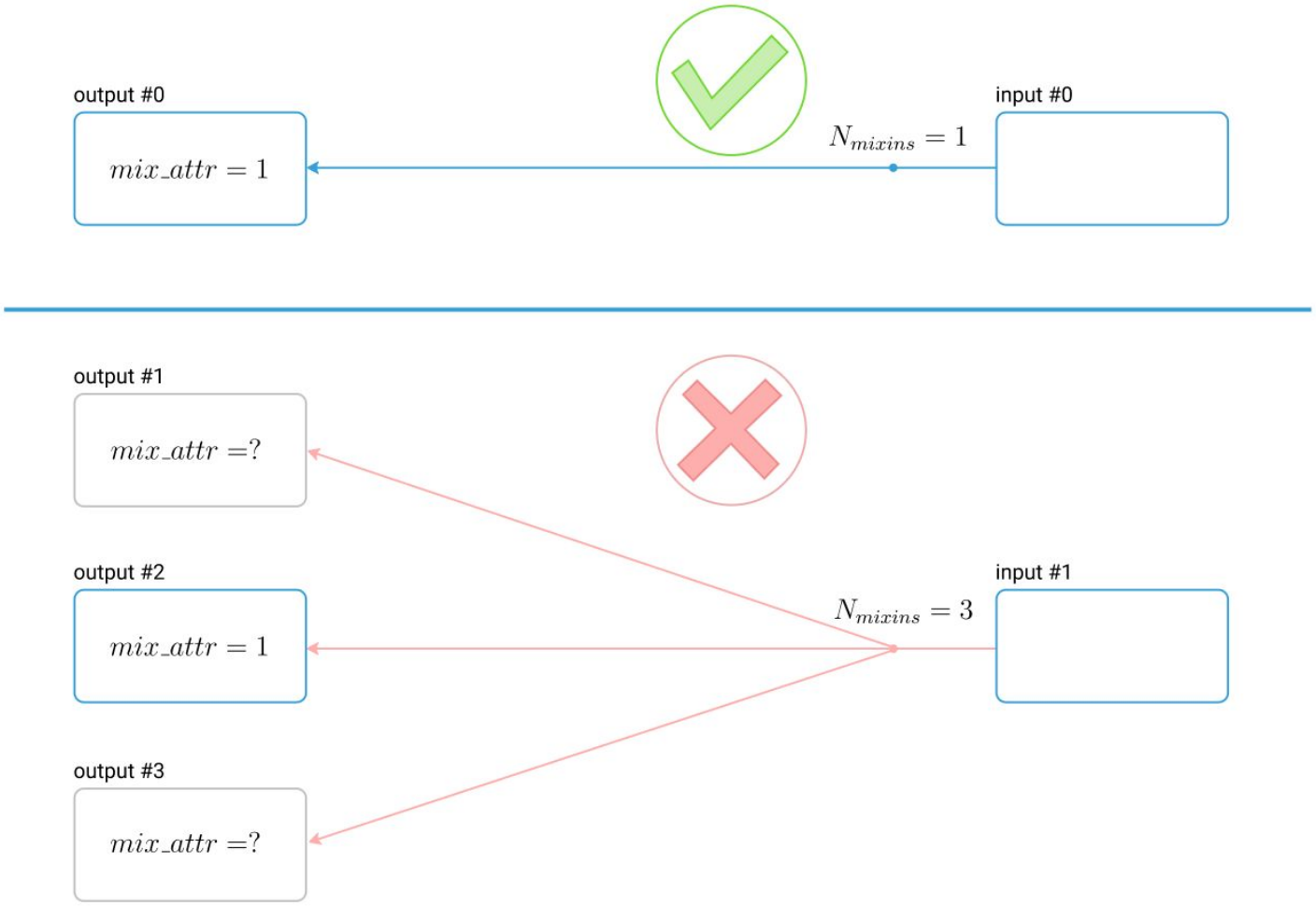

Fig. 6.2. Above: input # 0 refers only to output # 0 with mix_attr = 1 (direct spend) - a valid situation.
Bottom: input # 1 refers to three outputs: output # 2 with mix_attr = 1 and also to output # 1 and output # 3 - an invalid situation

Since output #2 has mix_attr = 1, it cannot be mixed with other outputs, regardless of their mix_attr attribute values. It can only be spent directly, as output #0.

This feature can be used for auditing.

Audit implementation using mix_attr = 1

As noted in Section 3, if, within the original CryptoNote protocol, Auditor Dan receives a secret key v from Bob, he will be able to recognize incoming transactions, but he will not be able to recognize outgoing transactions, so an exact balance calculation is impossible.

Nevertheless, since Dan will have full information about Bob's unspent outputs (UTXO), he will be able to track the fact that Bob's UTXO are being referred to by an input of some transaction in the blockchain. When using the mixing of other outputs, Dan will not be able to determine whether this transaction is spending Bob's output, which is achieved through the use of a ring signature. However, if there is no mixing and Bob’s UTXO is spent directly, then Auditor Dan can be sure that this transaction is Bob’s outgoing payment. This is the idea of this scheme.

Suppose Alice sends a transaction to Bob, and Bob wants Dan to see the fact that Alice received the money and that it was spent when Bob decides to spend it.

The whole process will look as follows (Fig. 6.3).

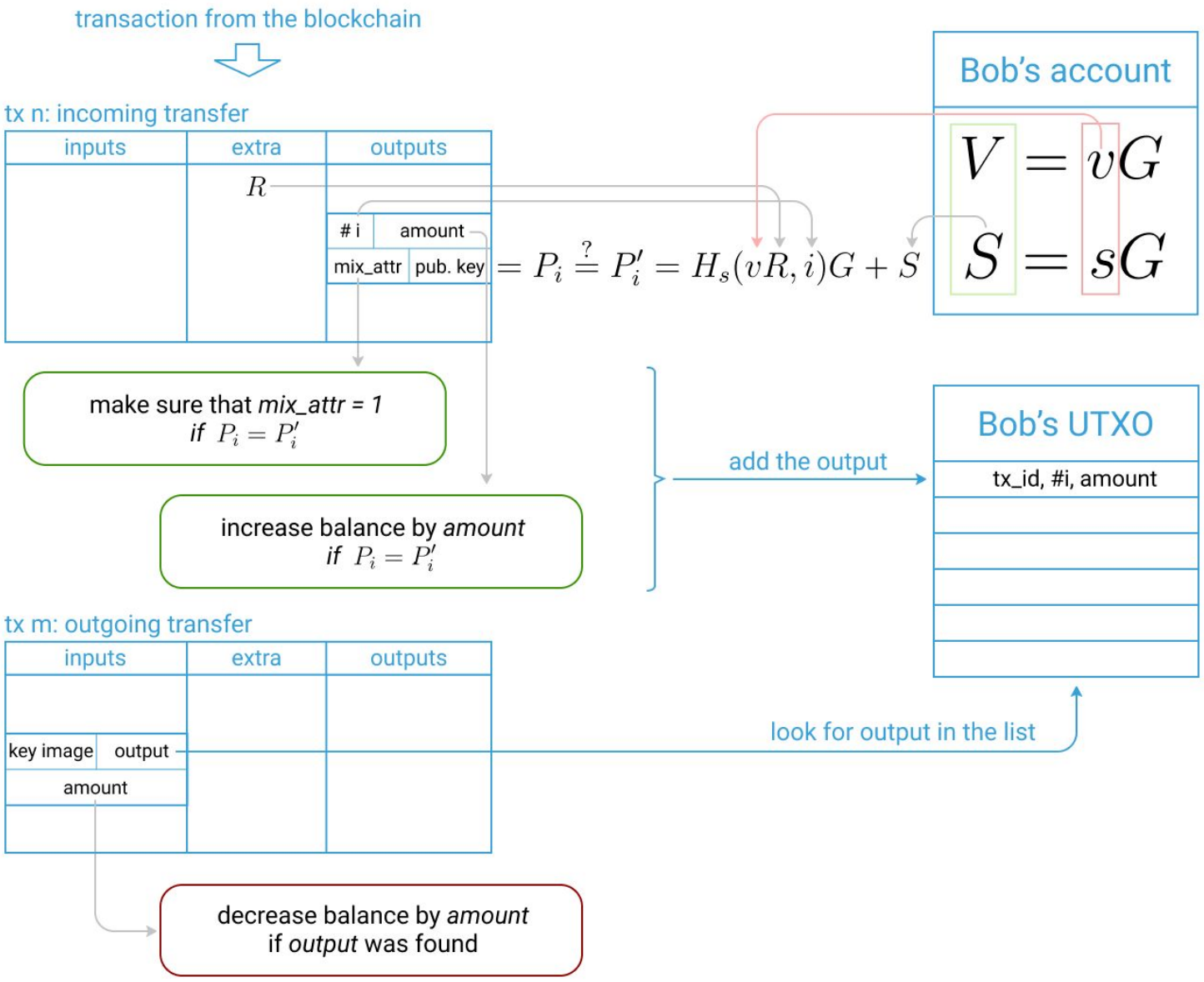

Fig. 6.3. Auditor using Bob's secret view key v determines incoming and outgoing transactions

1. Bob tells Alice his public address (V, S), and also asks her in all the outputs addressed to him to set the mix_attr attribute to 1 (see below on how this can be technically implemented).

2. Bob gives Dan his secret view key v.

3. Alice sends Bob a transaction using the usual CryptoNote scheme. In the outputs addressed to Bob, she sets mix_attr = 1.

4. Dan scans all transactions on the blockchain and, using Bob's secret view key, checks for each output P_i == P'_i = H_s (v * R, i) * G + S (where i is the index of the output, S is Bob's public spend key). If equality holds, then this output is addressed to Bob.
   Dan makes sure that mix_attr == 1, adds this output to the list and increases the balance by the value of the output.

5. Dan also checks all the inputs of all transactions, and if one of the inputs refers to Bob's UTXO from the list, this means that this transaction is Bob's outgoing payment. Thanks to the restriction mix_attr = 1, the input cannot mix other outputs, which means it spends Bob's output directly, and Dan can be sure that this

is Bob's outgoing payment.
Dan reduces the balance by the amount of the input value.

Thus, Dan will be able to calculate the correct balance of Bob’s wallet without gaining access to his secret spend key s.

Special notes and features

Feature 1. It is important that the sending party (Alice) set mix_attr to 1 when sending a transaction. If Alice doesn’t do this, then the funds will come to Bob, but in the future Dan will not be able to clearly say whether Bob spent them or not, since any other user will be able to use this output for mixing.

One solution is introducing a special type of address containing the same two public keys (V, S), but packed in a different way from the standard one. Alice, sending funds to an address of this type, will know about the need to set mix_attr = 1 for the corresponding outputs.

Feature 2. The use of addresses of a special type does not oblige Alice to set the desired attribute value for the outputs. She can send funds without doing this, however, it will be immediately revealed by both Bob and the auditor Dan.

Feature 3. In this scheme, audit capability is achieved at the price of the utmost reduction in untraceability. This does not present a particular problem if the audit opportunity is provided to an unlimited circle of people. For example, a charity fund can publish a donation address along with a secret view key, so that anyone can act as an auditor and calculate the current balance of his wallet. It is important that at the same time, the anonymity of incoming transfers remains standard for CryptoNote (a large number of N_{mixins} can be used). However, the anonymity of outgoing payments will be limited by the inability to mix arbitrary outputs.

It is worth noting that in the case when the audit opportunity is provided to a narrow circle of people, in contrast to the example above, a decrease in untraceability may be an undesirable effect.

The big advantages of this option are ease of implementation and the absence of additional load on calculations and data.

7. Conclusion

We examined three options for implementing audit to the CryptoNote protocol: Bytecoin Amethyst, the theoretical AS scheme by Anton Sokolov, and the use of mix_attr (denoted by MA).

The relative change in the size of the ring signature for all options except Bytecoin (BCN) does not depend on N_{inputs}, so consider N_{inputs} = 1 (minimum) and N_{inputs} = 3 (Fig. 7.1).

	N mixins						
1 input	**1**	**2**	**3**	**4**	**5**	**10**	**20**
CN vs BCN	150%	50%	17%	0%	-10%	-30%	-40%
CN vs AS	0%	-25%	-33%	-38%	-40%	-45%	-48%
CN vs MA	0%	0%	0%	0%	0%	0%	0%

	N mixins						
3 inputs	**1**	**2**	**3**	**4**	**5**	**10**	**20**
CN vs BCN	117%	33%	6%	-8%	-17%	-33%	-42%
CN vs AS	0%	-25%	-33%	-38%	-40%	-45%	-48%
CN vs MA	0%	0%	0%	0%	0%	0%	0%

Fig. 7.1. Signatures comparison when N_{inputs} = 1 and N_{inputs}= 3 cross all schemes

Note. N_{inputs} — the number of inputs in the transaction, N_{mixins} — the number of outputs associated with each input of the transaction. If N_{mixins} > 1, this means that to increase untraceability, each input refers to more than one output of some other transaction, and it is impossible to identify which one is real.

The best result is shown by the AS scheme, giving a reduction in the size of the ring signature in a wide range of the number of mixed outputs (N_{mixins}). Bytecoin also gives a good result with N_{mixins}> = 3 (as already noted, in Bytecoin Amethyst, a limit on the minimum number of N_{mixins} equal to 3 was introduced in order to increase untraceability).

The MA scheme does not offer any improvements regarding the size of the ring signature.

Let us now compare the computational complexity of the ring signature verification procedure. To do this, we will use the metric mentioned above, namely, we take the sum of the scalar multiplications and the calculation of the hash function H_p with weights proportional to their execution time on a typical modern processor:

$O(total) = 130 * O(*) + 15 * O(H_p)$

We have the following picture (Fig. 7.2).

	N mixins						
1 input	**1**	**2**	**3**	**4**	**5**	**10**	**20**
CN vs BCN	76%	38%	25%	19%	15%	8%	4%
CN vs AS	49%	49%	49%	49%	49%	49%	49%
CN vs MA	0%	0%	0%	0%	0%	0%	0%

Fig. 7.2. Comparison of signature validation complexity (N_{inputs} = 1)

With direct spending of funds (N_{mixins} = 1), AS and Bytecoin schemes require a significantly larger amount of computation to verify the signature.

When mixing outputs (N_{mixins} > 1), the Bytecoin Amethyst scheme is preferable to AS, however, the last MA variant considered remains unsurpassed.

Let us summarize the final result, taking into account factors such as an increase in the size of the address, inputs and outputs of transactions (Fig. 7.3).

criterion	CN	BCN	AS	MA
address size, bytes	64	64	96	64*
increase in output's size, bytes	0	32	32	0
increase in input's size, bytes	0	0	32	0
ring signature size, comparing to CN	0%	+17..-40%	0%..-48%	0%
signature verification complexity, comparing to CN	0%	4..76%	+49%	0%
untraceability descrease for the sake of audit	no	no	no	yes

Fig. 7.3. Final comparison of all the schemes

Bytecoin maintains the length of the address convenient for the end user, while the Anton Sokolov scheme increases it by 50%. This does not directly affect the blockchain size (although the sender address in encrypted form can be transmitted in extra transactions if the sender wishes, for example) and computational complexity, however, it does not significantly affect usability. In the MA scheme, the address size remains equal to 64 bytes; however, a special process is required to distinguish audit addresses from normal ones.

Audit options for Bytecoin and AS include adding one point (32 bytes) to each transaction output, however, the input size remains unchanged only for Bytecoin.

It is also worth noting that the Bytecoin Amethyst scheme has been implemented in the code for a long time and, judging by the lack of messages about problems that have passed since its implementation, it is well tested in practice. However, a whitepaper or a strict description could not be found, and therefore there is no formal evidence of its correctness.

The AS scheme, on the contrary, is strictly described and proposed for discussion in the crypto community [17].

The MA scheme does not have a strictly described formal proof; however, due to its extreme simplicity, it seems unnecessary.

References

1. “CryptoNote v 2.0”, Nicolas van Saberhagen https://cryptonote.org/whitepaper.pdf

2. Bytecoin ANN bitcointalk.org thread: https://bitcointalk.org/index.php?topic=512747.0

3. Bitmonero announcement thread at Bitcointalk https://bitcointalk.org/index.php?topic=563821.0

4. Bitmonero Github: https://github.com/bitmonero-project/bitmonero

5. “Ed25519: high-speed high-security signatures” https://ed25519.cr.yp.to/

6. "Elliptic Curve Cryptography: a gentle introduction", Andrea Corbellini

https://andrea.corbellini.name/2015/05/17/elliptic-curve-cryptography-a-gentle-introduction/

7. Understanding ge_fromfe_frombytes_vartime, Shen Noether
https://github.com/monero-project/research-lab/blob/master/whitepaper/ge_fromfe_writeup/ge_fromfe.pdf

8. “Ring confidential transactions”, Shen Noether, MRL https://eprint.iacr.org/2015/1098.pdf

9. Commit with hardcoded constant H in Monero:

https://github.com/monero-project/monero/commit/1b867e7f4087378a04a0b94d720d3bed8505e245

10. Bytecoin blog — Auditable coins https://bytecoin.org/blog/auditable-coins

11. "Cryptonote auditability. How to append a wallet balance audit."

https://medium.com/@coffeemas1/cryptonote-auditability-how-to-append-the-wallet-balance-audit-b2e5b47b69a6

12. "Discussion for the auditable wallets Variant 1 and 2"

https://medium.com/@coffeemas1/discussion-for-the-cryptonote-auditable-wallets-variant-1-3aed8261e34a

13. "The unlinkable auditable Variant 3"

https://medium.com/@coffeemas1/the-unlinkable-auditable-variant-3-7f259fb1d727

14. "The auditable variant 4. Memory efficiency and security question."

https://medium.com/@coffeemas1/the-unlinkable-auditable-variant-4-memory-efficiency-and-security-question-5e

5121ded839

15. "Multi-signature with LSAG. One more memory efficient approach to auditable wallets."

https://medium.com/@coffeemas1/multi-signature-within-lsag-one-more-memory-efficient-approach-to-unlinkable

-auditable-wallets-b70cc86d7c30

16. Abe, Ohkubo, Suzuki paper “1-out-of-n Signatures from a Variety of Keys” (AOS).
https://www.iacr.org/cryptodb/archive/2002/ASIACRYPT/50/50.pdf

17. Cryptonote auditability and efficient scheme for anonymous key vector proof.
https://bitcointalk.org/index.php?topic=5216247.0

18. "Practical approach for appending auditable wallets to the Cryptonote"
https://medium.com/@coffeemas1/practical-approach-for-appending-auditable-wallets-to-the-cryptonote-894030952d0

19. "Boolberry Solves CryptoNote Issues"
https://boolberry.com/files/Boolberry_Solves_CryptoNote_Issues.pdf

20. "Bytecoin Amethyst Stable Release Extended Technical Description"
https://bytecoin.org/blog/bytecoin-amethyst-stable-release-extended-technical-description