Judecoin

Version 1.0

August 26, 2020

Judecoin is a leading cryptocurrency that focuses on private and censored transactions. The publicly verifiable nature of most cryptography currencies allows anyone in the world to track your money. In addition, links between your financial records and your personal identity can compromise your security.

To avoid this, judecoin uses powerful cryptography to create a network that allows parties to interact without revealing the sender, receiver or transaction amount. Like other cryptography currencies, judecoin has a distributed ledger that all participants can download and verify by themselves.

However, judecoin uses a series of mathematical techniques to hide all privacy details and prevent any blockchain tracking. Judecoin's privacy function allows the network to evaluate the validity of the transaction and determine whether the sender has enough account balance, but without actually knowing the transaction amount or account balance! No one can view other people's account balances, and the transaction cannot disclose the source of the transferred funds.

Users can enjoy the transparency and security of the blockchain without taking all the risks of the financial system.

1 PREFACE

"Bitcoin" [1] has been a successful implementation of the concept of p2p electronic cash. Both professionals and the general public have come to appreciate the convenient combination of public transactions and proof-of-work as a trust model. Today, the user base of electronic cash is growing at a steady pace; customers are attracted to low fees and the anonymity provided by electronic cash and merchants value its predicted and decentralized emission. Bitcoin has effectively proved that electronic cash can be as simple as paper money and as convenient as credit cards.

Unfortunately, Bitcoin suffers from several deficiencies. For example, the system's distributed nature is inflexible, preventing the implementation of new features until almost all of the network users update their clients. Some critical flaws that cannot be fixed rapidly deter Bitcoin's widespread propagation. In such inflexible models, it is more efficient to roll-out a new project rather than perpetually fix the original project.

In this paper, we study and propose solutions to the main deficiencies of Bitcoin. We believe that a system taking into account the solutions we propose will lead to a healthy competition among different electronic cash systems. We also propose our own electronic cash, "CryptoNote", a name emphasizing the next breakthrough in electronic cash.

2 BITCOIN DRAWBACKS AND SOME POSSIBLE SOLUTIONS

2.1 TRACEABILITY OF TRANSACTIONS

Privacy and anonymity are the most important aspects of electronic cash. Peer-to-peer payments seek to be concealed from third party's view, a distinct difference when compared with traditional banking. In particular, T. Okamoto and K. Ohta described six criteria of ideal electronic cash, which included "privacy: relationship between the user and his purchases must be untraceable by anyone" [30]. From their description, we derived two properties which a fully anonymous electronic cash model must satisfy in order to comply with the requirements outlined by Okamoto and Ohta:

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.

Unfortunately, Bitcoin does not satisfy the untraceability requirement. Since all the transactions that take place between the network's participants are public, any transaction can be unambiguously traced to a unique origin and final recipient. Even if two participants exchange funds in an indirect way, a properly engineered path-finding method will reveal the origin and final recipient.

It is also suspected that Bitcoin does not satisfy the second property. Some researchers stated ([33, 35, 29, 31]) that a careful blockchain analysis may reveal a connection between the users of the Bitcoin network and their transactions. Although a number of methods are disputed [25], it is suspected that a lot of hidden personal information can be extracted from the public database.

Bitcoin's failure to satisfy the two properties outlined above leads us to conclude that it is not an anonymous but a pseudo-anonymous electronic cash system. Users were quick to develop solutions to circumvent this shortcoming. Two direct solutions were "laundering services" [2] and the development of distributed methods [3, 4]. Both solutions are based on the idea of mixing several public transactions and sending them through some intermediary address; which in turn suffers the drawback of requiring a trusted third party.

Recently, a more creative scheme was proposed by I. Miers et al. [28]: "Zerocoin". Zerocoin utilizes a cryptographic one-way accumulators and zero-knoweldge proofs which permit users to "convert" bitcoins to zerocoins and spend them using anonymous proof of ownership instead of explicit public-key based digital signatures. However, such knowledge proofs have a constant but inconvenient size - about 30kb (based on today's Bitcoin limits), which makes the proposal impractical. Authors admit that the protocol is unlikely to ever be accepted by the majority of Bitcoin users [5].

2.2 THE PROOF-OF-WORK FUNCTION

Bitcoin creator Satoshi Nakamoto described the majority decision making algorithm as "oneCPU-one-vote" and used a CPU-bound pricing function (double SHA-256) for his proof-of-work scheme. Since users vote for the single history of transactions order [1], the reasonableness and consistency of this process are critical conditions for the whole system.

The security of this model suffers from two drawbacks. First, it requires 51% of the network's mining power to be under the control of honest users. Secondly, the system's progress (bug fixes, security fixes, etc...) require the overwhelming majority of users to support and agree to the changes (this occurs when the users update their wallet software) [6].Finally this same voting mechanism is also used for collective polls about implementation of some features [7].

This permits us to conjecture the properties that must be satisfied by the proof-of-work pricing function. Such function must not enable a network participant to have a *significant* advantage over another participant; it requires a parity between common hardware and high cost of custom devices. From recent examples [8], we can see that the SHA-256 function used in the Bitcoin architecture does not posses this property as mining becomes more efficient on GPUs and ASIC devices when compared to high-end CPUs.

Therefore, Bitcoin creates favourable conditions for a large gap between the voting power of participants as it violates the "one-CPU-one-vote" principle since GPU and ASIC owners posses a much larger voting power when compared with CPU owners. It is a classical example of the Pareto principle where 20% of a system's participants control more than 80% of the votes.

One could argue that such inequality is not relevant to the network's security since it is not the small number of participants controlling the majority of the votes but the honesty of these participants that matters. However, such argument is somewhat flawed since it is rather the possibility of cheap specialized hardware appearing rather than the participants' honesty which poses a threat. To demonstrate this, let us take the following example. Suppose a malevolent individual gains significant mining power by creating his own mining farm through the cheap hardware described previously. Suppose that the global hashrate decreases significantly, even for a moment, he can now use his mining power to fork the chain and double-spend. As we shall see later in this article, it is not unlikely for the previously described event to take place.

2.3 IRREGULAR EMISSION

Bitcoin has a predetermined emission rate: each solved block produces a fixed amount of coins. Approximately every four years this reward is halved. The original intention was to create a limited smooth emission with exponential decay, but in fact we have a piecewise linear emission function whose breakpoints may cause problems to the Bitcoin infrastructure.

When the breakpoint occurs, miners start to receive only half of the value of their previous reward. The absolute difference between 12.5 and 6.25 BTC (projected for the year 2020) may seem tolerable. However, when examining the 50 to 25 BTC drop that took place on November 28 2012, felt inappropriate for a significant number of members of the mining community. Figure 1 shows a dramatic decrease in the network's hashrate in the end of November, exactly when the halving took place. This event could have been the perfect moment for the malevolent individual described in the proof-of-work function section to carry-out a double spending attack [36].

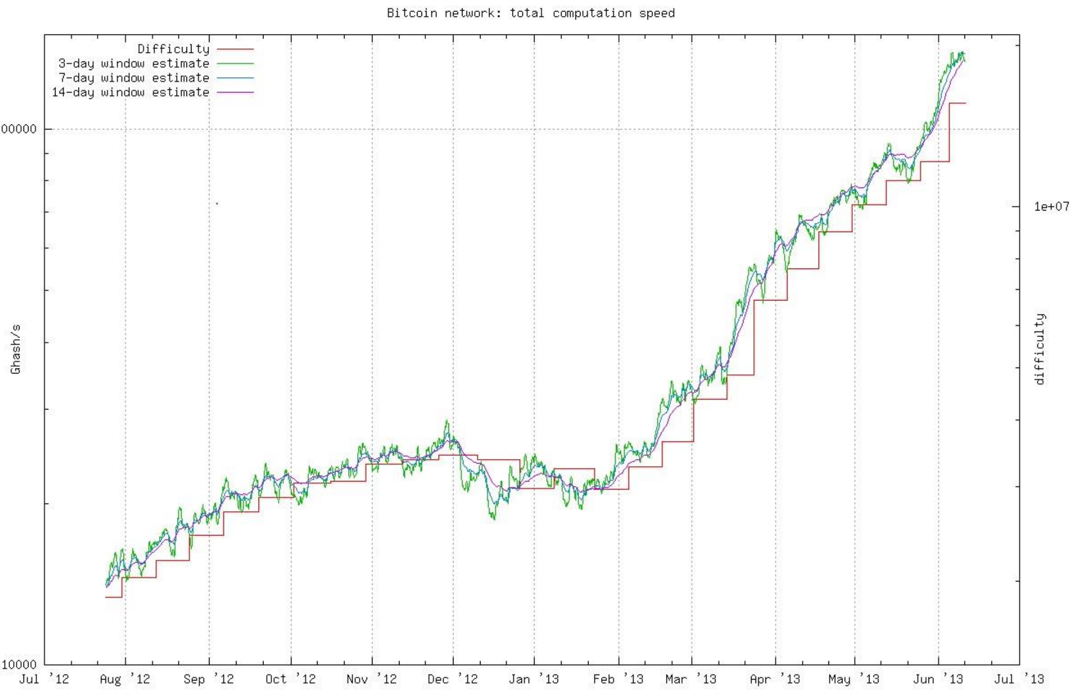

Fig. 1. Bitcoin hashrate chart

(source: http://bitcoin.sipa.be)

2.4 HARDCODED CONSTANTS

Bitcoin has many hard-coded limits, where some are natural elements of the original design (e.g. block frequency, maximum amount of money supply, number of confirmations) whereas other seem to be artificial constraints. It is not so much the limits, as the inability of quickly changing them if necessary that causes the main drawbacks. Unfortunately, it is hard to predict when the constants may need to be changed and replacing them may lead to terrible consequences.

A good example of a hardcoded limit change leading to disastrous consequences is the block size limit set to 250kb[1]. This limit was sufficient to hold about 10000 standard transactions. In early 2013, this limit had almost been reached and an agreement was reached to increase the limit. The change was implemented in wallet version 0.8 and ended with a 24-blocks chain split and a successful double-spend attack [9]. While the bug was not in the Bitcoin protocol, but rather in the database engine it could have been easily caught by a simple stress test if there was no artificially introduced block size limit.

[1] This is so-called “soft limit” — the reference client restriction for creating new blocks. Hard maximum of *possible* blocksize was 1 MB

Constants also act as a form of centralization point. Despite the peer-to-peer nature of Bitcoin, an overwhelming majority of nodes use the official reference client [10] developed by a small group of people. This group makes the decision to implement changes to the protocol and most people accept these changes irrespective of their "correctness". Some decisions caused heated discussions and even calls for boycott [11], which indicates that the community and the developers may disagree on some important points. It therefore seems logical to have a protocol with user-configurable and self-adjusting variables as a possible way to avoid these problems.

2.5 Bulky scripts

The scripting system in Bitcoin is a heavy and complex feature. It *potentially* allows one to create sophisticated transactions [12], but some of its features are disabled due to security concerns and some have never even been used [13]. The script (including both senders' and receivers' parts) for the most popular transaction in Bitcoin looks like this:

```
<sig> <pubKey> OP DUP OP HASH160 <pubKeyHash> OP EQUALVERIFY OP CHECKSIG.
```

The script is 164 bytes long whereas its only purpose is to check if the receiver possess the secret key required to verify his signature.

3 The CryptoNote Technology

Now that we have covered the limitations of the Bitcoin technology, we will concentrate on presenting the features of CryptoNote.

4 Untraceable Transactions

In this section we propose a scheme of fully anonymous transactions satisfying both untraceability and unlinkability conditions. An important feature of our solution is its autonomy: the sender is not required to cooperate with other users or a trusted third party to make his transactions; hence each participant produces a cover traffic independently.

4.1 Literature review

Our scheme relies on the cryptographic primitive called a *group signature*. First presented by D. Chaum and E. van Heyst [19], it allows a user to sign his message on behalf of the group.

After signing the message the user provides (for verification purposes) not his own single public key, but the keys of all the users of his group. A verifier is convinced that the real signer is a member of the group, but cannot exclusively identify the signer.

The original protocol required a trusted third party (called the Group Manager), and he was the only one who could trace the signer. The next version called a *ring signature*, introduced by

Rivest et al. in [34], was an autonomous scheme without Group Manager and anonymity revocation. Various modifications of this scheme appeared later: *linkable ring signature* [26, 27, 17] allowed to determine if two signatures were produced by the same group member, *traceable ring signature* [24, 23] limited excessive anonymity by providing possibility to trace the signer of two messages with respect to the same metainformation (or "tag" in terms of [24]).

A similar cryptographic construction is also known as a *ad-hoc group signature* [16, 38]. It emphasizes the arbitrary group formation, whereas group/ring signature schemes rather imply a fixed set of members.

For the most part, our solution is based on the work "Traceable ring signature" by E. Fujisaki and K. Suzuki [24]. In order to distinguish the original algorithm and our modification we will call the latter a *one-time ring signature*, stressing the user's capability to produce only one valid signature under his private key. We weakened the traceability property and kept the linkability only to provide one-timeness: the public key may appear in many foreign verifying sets and the private key can be used for generating a unique anonymous signature. In case of a double spend attempt these two signatures will be linked together, but revealing the signer is not necessary for our purposes.

4.2 DEFINITIONS

4.2.1 Elliptic curve parameters

As our base signature algorithm we chose to use the fast scheme EdDSA, which is developed and implemented by D.J. Bernstein et al. [18]. Like Bitcoin's ECDSA it is based on the elliptic curve discrete logarithm problem, so our scheme could also be applied to Bitcoin in future. Common parameters are:

q: a prime number; $q = 2^{255} - 19$; d:

an element of $\mathbb{F}_q$; $d = -121665/121666$;

E: an elliptic curve equation; $-x^2 + y^2 = 1 + dx^2y^2$; G: a base point; $G = (x, -4/5)$; l: a

prime order of the base point; $l = 2^{252} + 27742317777372353535851937790883648493$;

$\mathcal{H}_s$: a cryptographic hash function $\{0,1\}^* \to \mathbb{F}_q$;

$\mathcal{H}_p$: a deterministic hash function $E(\mathbb{F}_q) \to E(\mathbb{F}_q)$.

4.2.2 Terminology

Enhanced privacy requires a new terminology which should not be confused with Bitcoin entities.

private ec-key is a standard elliptic curve private key: a number $a \in [1,l -$ 1]; **public ec-key** is a standard elliptic curve public key: a point $A = aG$; **one-time keypair** is a pair of private and public ec-keys;

private user key is a pair (a,b) of two different private ec-keys; **tracking key** is a pair (a,B) of private and public ec-key (where $B = bG$ and a 6= b); **public user key** is a pair (A,B) of two public ec-keys derived from (a,b);

standard address is a representation of a public user key given into human friendly string with error correction;

truncated address is a representation of the second half (point B) of a public user key given into human friendly string with error correction.

The transaction structure remains similar to the structure in Bitcoin: every user can choose several independent incoming payments (transactions outputs), sign them with the corresponding private keys and send them to different destinations.

Contrary to Bitcoin's model, where a user possesses unique private and public key, in the proposed model a sender generates a one-time public key based on the recipient's address and some random data. In this sense, an incoming transaction for the same recipient is sent to a one-time public key (not directly to a unique address) and only the recipient can recover the corresponding private part to redeem his funds (using his unique private key). The recipient can spend the funds using a ring signature, keeping his ownership and actual spending anonymous. The details of the protocol are explained in the next subsections.

4.3 UNLINKABLE PAYMENTS

Classic Bitcoin addresses, once being published, become unambiguous identifier for incoming payments, linking them together and tying to the recipient's pseudonyms. If someone wants to receive an "untied" transaction, he should convey his address to the sender by a private channel. If he wants to receive different transactions which cannot be proven to belong to the same owner he should generate all the different addresses and never publish them in his own pseudonym.

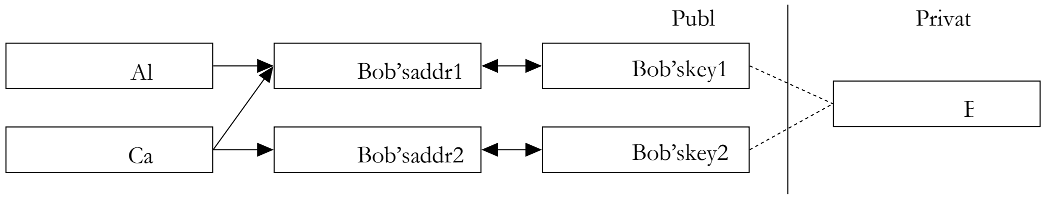

Fig. 2. Traditional Bitcoin keys/transactions model.

We propose a solution allowing a user to publish **a single address** and receive unconditional unlinkable payments. The destination of each CryptoNote output (by default) is a public key, derived from recipient's address and sender's random data. The main advantage against Bitcoin is that every destination key is unique by default (unless the sender uses the same data for each of his transactions to the same recipient). Hence, there is no such issue as "address reuse" by design and no observer can determine if any transactions were sent to a specific address or link two addresses together.

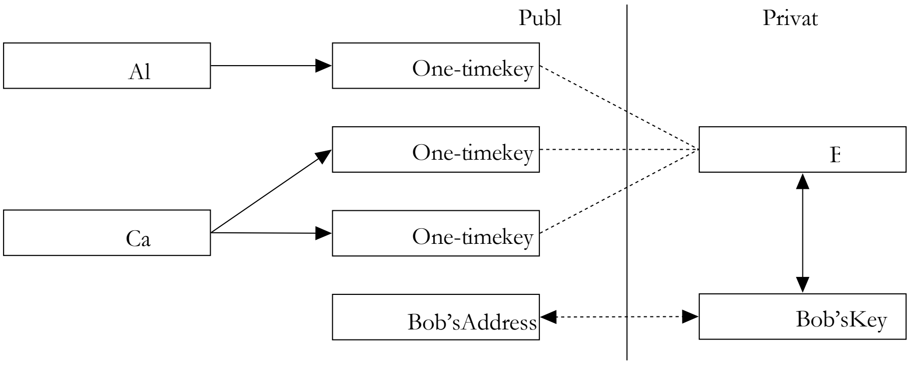

Fig. 3. CryptoNote keys/transactions model.

First, the sender performs a Diffie-Hellman exchange to get a shared secret from his data and half of the recipient's address. Then he computes a one-time destination key, using the shared secret and the second half of the address. Two different ec-keys are required from the recipient for these two steps, so a standard CryptoNote address is nearly twice as large as a Bitcoin wallet address. The receiver also performs a Diffie-Hellman exchange to recover the corresponding secret key.

A standard transaction sequence goes as follows:

1. Alice wants to send a payment to Bob, who has published his standard address. She unpacks the address and gets Bob's public key (A,B).

2. Alice generates a random $r \in [1,l-1]$ and computes a one-time public key $P =$
$H_s(rA)G+$
B.

3. Alice uses P as a destination key for the output and also packs value $R = rG$ (as a part of the Diffie-Hellman exchange) somewhere into the transaction. Note that she can create other outputs with unique public keys: different recipients' keys (A_i,B_i) imply different P_i even with the same r.

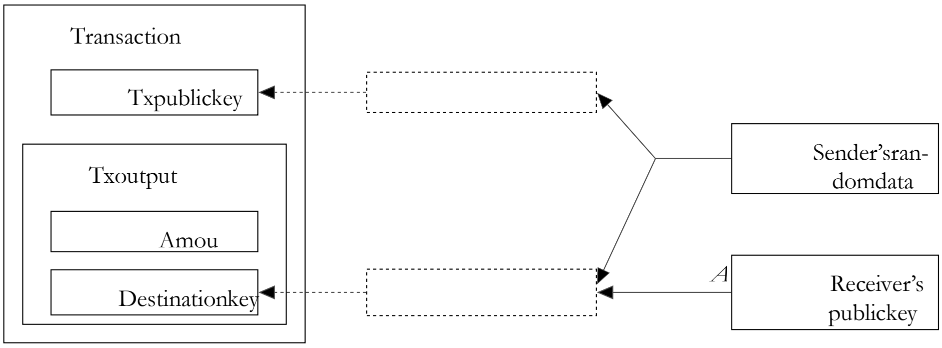

Fig. 4. Standard transaction structure.

4. Alice sends the transaction.

5. Bob checks every passing transaction with his private key (a,b), and computes $P^0 = H_s(aR)G + B$. If Alice's transaction for with Bob as the recipient was among them, then $aR = arG = rA$ and $P^0 = P$.

6. Bob can recover the corresponding one-time private key: $x = H_s(aR) + b$, so as $P = xG$. He can spend this output at any time by signing a transaction with x.

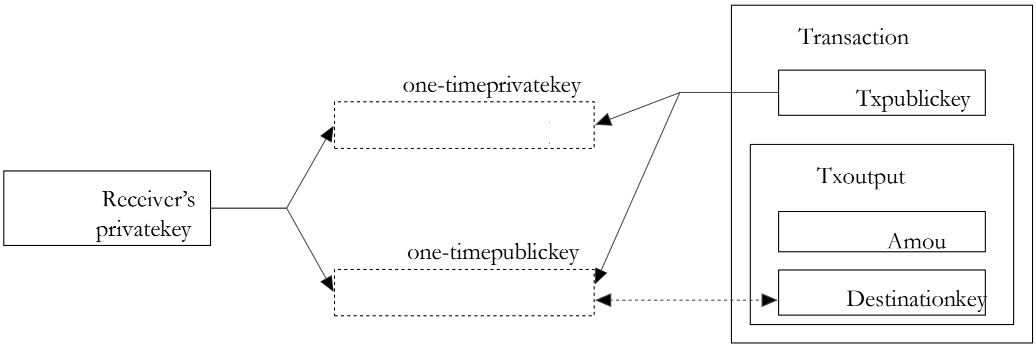

Fig. 5. Incoming transaction check.

As a result Bob gets incoming payments, associated with one-time public keys which are **unlinkable** for a spectator. Some additional notes:

- When Bob "recognizes" his transactions (see step 5) he practically uses only half of his private information: (a,B). This pair, also known as the **tracking key**, can be passed to a third party (Carol). Bob can delegate her the processing of new transactions. Bob doesn't need to explicitly trust Carol, because she can't recover the one-time secret key p without Bob's full private key (a,b). This approach is useful when Bob lacks bandwidth or computation power (smartphones, hardware wallets etc.).

- In case Alice wants to prove she sent a transaction to Bob's address she can either disclose r or use any kind of zero-knowledge protocol to prove she knows r (for example by signing the transaction with r).

- If Bob wants to have an audit compatible address where all incoming transaction are linkable, he can either publish his tracking key or use a **truncated address**. That address represent only one public ec-key B, and the remaining part required by the protocol is derived from it as follows: $a = H_s(B)$ and $A = H_s(B)G$. In both cases every person is able to "recognize" all of Bob's incoming transaction, but, of course, none can spend the funds enclosed within them without the secret key b.

4.4 ONE-TIME RING SIGNATURES

A protocol based on one-time ring signatures allows users to achieve unconditional unlinkability. Unfortunately, ordinary types of cryptographic signatures permit to trace transactions to their respective senders and receivers. Our solution to this deficiency lies in using a different signature type than those currently used in electronic cash systems.

We will first provide a general description of our algorithm with no explicit reference to electronic cash.

A one-time ring signature contains four algorithms: (**GEN**, **SIG**, **VER**, **LNK**):

GEN: takes public parameters and outputs an ec-pair (P,x) and a public key I.

SIG: takes a message m, a set S^0 of public keys $\{P_i\}_{i6=s}$, a pair (P_s,x_s) and outputs a signature σ and a set $S = S^0 \cup \{P_s\}$.

VER: takes a message m, a set S, a signature σ and outputs "true" or "false".

LNK: takes a set $I = \{I_i\}$, a signature σ and outputs "linked" or "indep".

The idea behind the protocol is fairly simple: a user produces a signature which can be checked by a set of public keys rather than a unique public key. The identity of the signer is indistinguishable from the other users whose public keys are in the set until the owner produces a second signature using the same keypair.

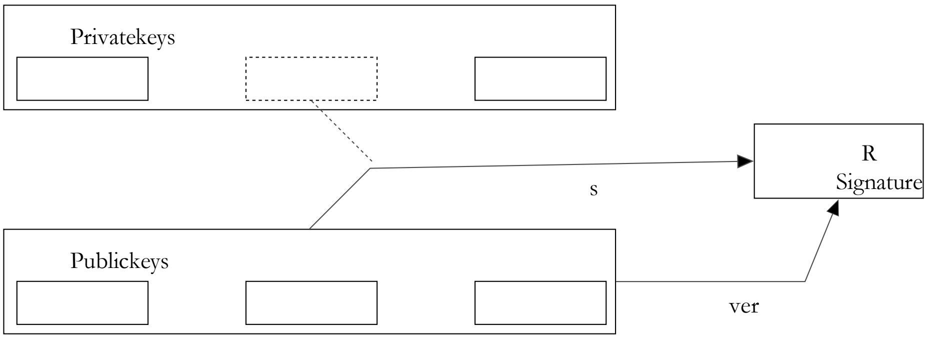

Fig. 6. Ring signature anonymity.

GEN: The signer picks a random secret key $x \in [1, l - 1]$ and computes the corresponding public key $P = xG$. Additionally he computes another public key $I = x\mathrm{H}_p(P)$ which we will call the "key image".

SIG: The signer generates a one-time ring signature with a non-interactive zero-knowledge proof using the techniques from [21]. He selects a random subset S^0 of n from the other users' public keys P_i, his own keypair (x,P) and key image I. Let $0 \leq s \leq n$ be signer's secret index in S (so that his public key is P_s).

He picks a random $\{q_i \mid i = 0...n\}$ and $\{w_i \mid i = 0...n, i\ 6 = s\}$ from $(1...l)$ and applies the following *transformations*:

$$L_i = \begin{cases} q_i G, & \text{if } i = s \\ & i \neq s \end{cases} q_iG + w_iP_i, \quad \text{if}$$

$$\left(q_i\mathrm{H}_p(P_i), \text{if } i = s \right.$$

$$\mathrm{R}_i =$$

$$q_i\mathrm{H}_p(P_i) + w_iI, \quad \text{if } i\ 6 = s$$

The next step is getting the non-interactive *challenge*:

$c = \mathrm{H}_s(m, L_1, ..., L_n, \mathrm{R}_1, ..., \mathrm{R}_n)$ Finally the signer computes the

response:

$$c_i = \begin{cases} w_i, & \text{if } i \neq s \\ c - \sum\limits_{i=0}^{n} c_i \mod l, & \text{if } i = s \end{cases}$$

$$($$

$$q_i, \qquad \text{if } i\ 6 = s$$

$$r_i = q_s - c_sx \qquad \mod l,$$

$$\text{if } i = s$$

The resulting signature is $\sigma = (I, c_1, ..., c_n, r_1, ..., r_n)$.

VER: The verifier checks the signature by applying the inverse transformations:

$$(L0_i = r_iG + c_iP_i$$

$$\mathrm{R}_{i0} = r_i\mathrm{H}_p(P_i) + c_iI\ {}_n$$

Finally, the verifier checks if $\mathrm{P}_{i=0}\ c_i =^? \mathrm{H}_s(m, L^0_0, ..., L^0_n, \mathrm{R}_0{}^0, ..., \mathrm{R}_n{}^0) \mod l$

If this equality is correct, the verifier runs the algorithm **LNK**. Otherwise the verifier rejects the signature.

LNK: The verifier checks if I has been used in past signatures (these values are stored in the set I). Multiple uses imply that two signatures were produced under the same secret key.

The meaning of the protocol: by applying L-transformations the signer proves that he knows such x that at least one $P_i = xG$. To make this proof non-repeatable we introduce the key image as $I = x\mathrm{H}_p(P)$. The signer uses the same coefficients (r_i,c_i) to prove almost the same statement:

he knows such x that at least one $\mathrm{H}_p(P_i) =$

$I \cdot x^{-1}$. If the mapping $x \to I$ is an injection:

1. Nobody can recover the public key from the key image and identify the signer;

2. The signer cannot make two signatures with different I's and the same x.

A full security analysis is provided in Appendix A.

4.5 STANDARD CRYPTONOTE TRANSACTION

By combining both methods (unlinkable public keys and untraceable ring signature) Bob achieves new level of privacy in comparison with the original Bitcoin scheme. It requires him to store only one private key (a,b) and publish (A,B) to start receiving and sending anonymous transactions.

While validating each transaction Bob additionally performs only two elliptic curve multiplications and one addition per output to check if a transaction belongs to him. For his every output Bob recovers a one-time keypair (p_i,P_i) and stores it in his wallet. Any inputs can be *circumstantially proved* to have the same owner only if they appear in a single transaction. In fact this relationship is much harder to establish due to the one-time ring signature.

With a ring signature Bob can effectively hide every input among somebody else's; all possible spenders will be equiprobable, even the previous owner (Alice) has no more information than any observer.

When signing his transaction Bob specifies n foreign outputs with the same amount as his output, mixing all of them without the participation of other users. Bob himself (as well as anybody else) does not know if any of these payments have been spent: an output can be used in thousands of signatures as an ambiguity factor and never as a target of hiding. The double spend check occurs in the **LNK** phase when checking against the used key images set.

Bob can choose the ambiguity degree on his own: $n = 1$ means that the probability he has spent the output is 50% probability, $n = 99$ gives 1%. The size of the resulting signature increases linearly as $O(n+1)$, so the improved anonymity costs to Bob extra transaction fees. He also can set $n = 0$ and make his ring signature to consist of only one element, however this will instantly reveal him as a spender.

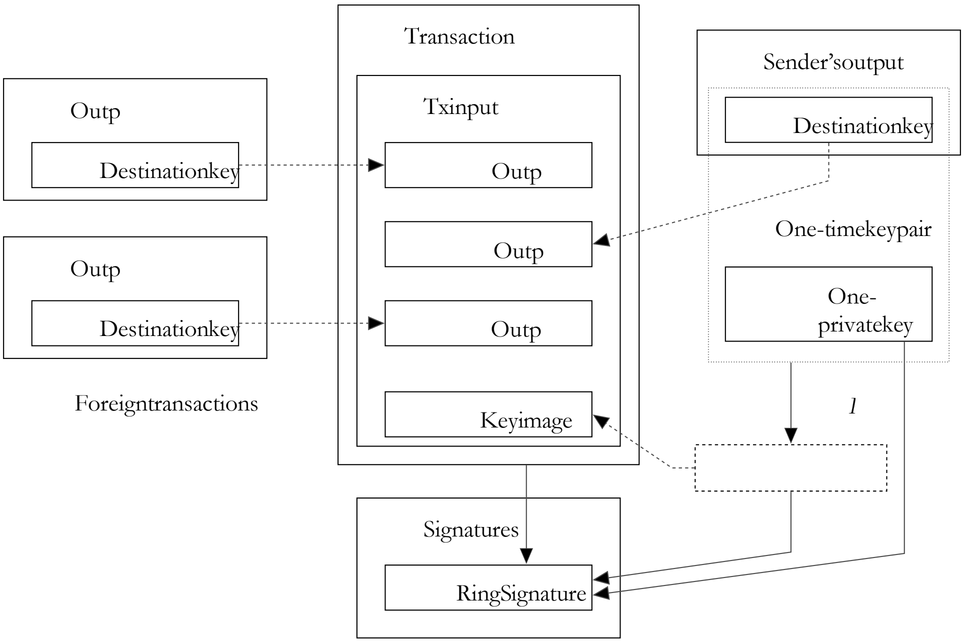

Fig. 7. Ring signature generation in a standard transaction.

5 EGALITARIAN PROOF-OF-WORK

In this section we propose and ground the new proof-of-work algorithm. Our primary goal is to close the gap between CPU (majority) and GPU/FPGA/ASIC (minority) miners. It is appropriate that some users can have a certain advantage over others, but their investments should grow at least linearly with the power. More generally, producing special-purpose devices has to be as less profitable as possible.

5.1 RELATED WORKS

The original Bitcoin proof-of-work protocol uses the CPU-intensive pricing function SHA-256. It mainly consists of basic logical operators and relies solely on the computational speed of processor, therefore is perfectly suitable for multicore/conveyer implementation.

However, modern computers are not limited by the number of operations per second alone, but also by memory size. While some processors can be substantially faster than others [8], memory sizes are less likely to vary between machines.

Memory-bound price functions were first introduced by Abadi et al and were defined as "functions whose computation time is dominated by the time spent accessing memory" [15]. The main idea is to construct an algorithm allocating a large block of data ("scratchpad") within memory that can be accessed relatively slowly (for example, RAM) and "accessing an unpredictable sequence of locations" within it. A block should be large enough to make preserving the data more advantageous than recomputing it for each access. The algorithm also should prevent internal parallelism, hence N simultaneous threads should require N times more memory at once.

Dwork et al [22] investigated and formalized this approach leading them to suggest another variant of the pricing function: “Mbound”. One more work belongs to F. Coelho [20], who proposed the most effective solution: “Hokkaido”.

To our knowledge the last work based on the idea of pseudo-random searches in a big array is the algorithm known as “scrypt” by C. Percival [32]. Unlike the previous functions it focuses on key derivation, and not proof-of-work systems. Despite this fact scrypt can serve our purpose: it works well as a pricing function in the partial hash conversion problem such as SHA-256 in

Bitcoin.

By now scrypt has already been applied in Litecoin [14] and some other Bitcoin forks. However, its implementation is not really memory-bound: the ratio “memory access time / overall time” is not large enough because each instance uses only 128 KB. This permits GPU miners to be roughly 10 times more effective and continues to leave the possibility of creating relatively cheap but highly-efficient mining devices.

Moreover, the scrypt construction itself allows a *linear* trade-off between memory size and CPU speed due to the fact that every block in the scratchpad is derived only from the previous. For example, you can store every second block and recalculate the others in a lazy way, i.e. only when it becomes necessary. The pseudo-random indexes are assumed to be uniformly distributed, hence the expected value of the *additional* blocks’ recalculations is $\frac{1}{2} \cdot N$, where N is the number of iterations. The overall computation time increases less than by half because there are also time independent (constant time) operations such as preparing the scratchpad and hashing on every iteration. Saving 2/3 of the memory costs $\frac{1}{3} \cdot N + \frac{1}{3} \cdot 2 \cdot N = N$ additional recalculations; 9/10 results in $\frac{1}{10} \cdot N + \ldots + \frac{1}{10} \cdot 9 \cdot N = 4.5N$. It is easy to show that storing only $\frac{1}{s}$ of all blocks increases the time less than by a factor of $\frac{s-1}{2}$. This in turn implies that a machine with a CPU 200 times faster than the modern chips can store only 320 bytes of the scratchpad.

5.2 THE PROPOSED ALGORITHM

We propose a new memory-bound algorithm for the proof-of-work pricing function. It relies on random access to a slow memory and emphasizes latency dependence. As opposed to scrypt every new block (64 bytes in length) depends on *all* the previous blocks. As a result a hypothetical “memory-saver” should increase his calculation speed exponentially.

Our algorithm requires about 2 Mb per instance for the following reasons:

1. It fits in the L3 cache (per core) of modern processors, which should become mainstream in a few years;

2. A megabyte of internal memory is an almost unacceptable size for a modern ASIC pipeline;

3. GPUs may run hundreds of concurrent instances, but they are limited in other ways: GDDR5 memory is slower than the CPU L3 cache and remarkable for its bandwidth, not random access speed.

4. Significant expansion of the scratchpad would require an increase in iterations, which in turn implies an overall time increase. "Heavy" calls in a trust-less p2p network may lead to serious vulnerabilities, because nodes are obliged to check every new block's proof-of-work. If a node spends a considerable amount of time on each hash evaluation, it can be easily DDoSed by a flood of fake objects with arbitrary work data (nonce values).

6 FURTHER ADVANTAGES

6.1 SMOOTH EMISSION

The upper bound for the overall amount of CryptoNote digital coins is: $\text{MSupply} = 8.7 * 10^{16} / 10^{9}$ jude units. This is a natural restriction based only on implementation limits, not on intuition such as "*N* coins ought to be enough for anybody".

To ensure the smoothness of the emission process we use the following formula for block rewards:

$$BaseReward = 120/ceil(A/365462) * 10^{9},$$

where *A* is amount of previously generated blocks.

6.2 ADJUSTABLE PARAMETERS

6.2.1 Difficulty

CryptoNote contains a targeting algorithm which changes the difficulty of every block. This decreases the system's reaction time when the network hashrate is intensely growing or shrinking, preserving a constant block rate. The original Bitcoin method calculates the relation of actual and target time-span between the last 2016 blocks and uses it as the multiplier for the current difficulty. Obviously this is unsuitable for rapid recalculations (because of large inertia) and results in oscillations.

The general idea behind our algorithm is to sum all the work completed by the nodes and divide it by the time they have spent. The measure of work is the corresponding difficulty values in each block. But due to inaccurate and untrusted timestamps we cannot determine the exact time interval between blocks. A user can shift his timestamp into the future and the next time intervals might be improbably small or even negative. Presumably there will be few incidents of this kind, so we can just sort the timestamps and cut-off the outliers (i.e. 20%). The range of the rest values is the time which was spent for 80% of the corresponding blocks.

6.2.2Size limits

Users pay for storing the blockchain and shall be entitled to vote for its size. Every miner deals with the trade-off between balancing the costs and profit from the fees and sets his own “soft-limit” for creating blocks. Also the core rule for the maximum block size is necessary for preventing the blockchain from being flooded with bogus transaction, however this value should not be hard-coded.

Let M_N be the median value of the last N blocks sizes. Then the “hard-limit” for the size of accepting blocks is $2 \cdot M_N$. It averts the blockchain from bloating but still allows the limit to slowly grow with time if necessary.

Transaction size does not need to be limited explicitly. It is bounded by the size of a block; and if somebody wants to create a huge transaction with hundreds of inputs/outputs (or with the high ambiguity degree in ring signatures), he can do so by paying sufficient fee.

6.2.3Excess size penalty

A miner still has the ability to stuff a block full of his own zero-fee transactions up to its maximum size $2 \cdot M_b$. Even though only the majority of miners can shift the median value, there is still a possibility to bloat the blockchain and produce an additional load on the nodes. To discourage malevolent participants from creating large blocks we introduce a penalty function:

$$NewReward = BaseReward \cdot \left(\frac{BlkSize}{M_N} - 1\right)^2 \underline{\qquad\qquad}$$

This rule is applied only when $BlkSize$ is greater than minimal free block size which should be close to $\mathsf{max}(10kb, M_N \cdot 110\%)$. Miners are permitted to create blocks of “usual size” and even exceed it with profit when the overall fees surpass the penalty. But fees are unlikely to grow quadratically unlike the penalty value so there will be an equilibrium.

6.3 TRANSACTION SCRIPTS

CryptoNote has a very minimalistic scripting subsystem. A sender specifies an expression $\Phi = f(x_1, x_2, \ldots, x_n)$, where n is the number of destination public keys$\{P_i\}_{i=1}^n$. Only five binary operators are supported: min, max, sum, mul and cmp. When the receiver spends this payment, he produces $0 \le k \le n$ signatures and passes them to transaction input. The verification process simply evaluates Φ with $x_i = 1$ to check for a valid signature for the public key P_i, and $x_i = 0$.

A verifier accepts the proof iff $\Phi > 0$.

Despite its simplicity this approach covers every possible case:

- **Multi-/Threshold signature.** For the Bitcoin-style “M-out-of-N” multi-signature (i.e. the receiver should provide at least $0 \le M \le N$ valid signatures) $\Phi = x_1 + x_2 + \ldots + x_N \ge M$ (for clarity we are using common algebraic notation). The weighted threshold signature (some keys can be more important than other) could be expressed as $\Phi = w_1 \cdot x_1 + w_2 \cdot x_2 + \ldots + w_N \cdot x_N \ge w_M$. And scenario where the master-key corresponds to $\Phi = \mathsf{max}(M \cdot x, x_1 + x_2 + \ldots + x_N) \ge M$. It is easy to show that any sophisticated case can be expressed with these operators, i.e. they form basis.

- **Password protection.** Possession of a secret password s is equivalent to the knowledge of a private key, deterministically derived from the password: $k = \mathsf{KDF}(s)$. Hence, a receiver can prove that he knows the password by providing another signature under the key k. The sender simply adds the corresponding public key to his own output. Note that this method is much more secure than the "transaction puzzle" used in Bitcoin [13], where the password is explicitly passed in the inputs.

- **Degenerate cases.** $\Phi = 1$ means that anybody can spend the money; $\Phi = 0$ marks the output as not spendable forever.

In the case when the output script combined with public keys is too large for a sender, he can use special output type, which indicates that the recipient will put this data in his input while the sender provides only a hash of it. This approach is similar to Bitcoin's "pay-to-hash" feature, but instead of adding new script commands we handle this case at the data structure level.

6.4 AN EFFICIENT IMPLEMENTATION OF JUDECOIN SUBADDRESSES

Users of the judecoin cryptocurrency who wish to reuse wallet addresses in an unlinkable way must maintain separate wallets, which necessitates scanning incoming transactions for each one. We document a new address scheme that allows a user to maintain a single master wallet address and generate an arbitrary number of unlinkable subaddresses. Each transaction needs to be scanned only once to determine if it is destinated for any of the user's subaddresses. The scheme additionally supports multiple outputs to other subaddresses, and is as efficient as traditional wallet transactions.

6.5 DUAL LINKABLE RING SIGNATURES

This bulletin describes a modification to judecoin's linkable ring signature scheme that permits dual-key outputs as ring members. Key images are tied to both output one-time public keys in a dual, preventing both keys in that transaction from being spent separately. This method has applications to non-interactive refund transactions. We discuss the security implications of the scheme.

6.6 CONCISE LINKABLE RING SIGNATURES AND FORGERY AGAINST ADVERSARIAL KEYS

We demonstrate that a version of non-slanderability is a natural definition of unforgeability for linkable ring signatures. We present a linkable ring signature construction with concise signatures and multi-dimensional keys that is linkably anonymous if a variation of the decisional Diffie-Hellman problem with random oracles is hard, linkable if key aggregation is a one-way function, and non-slanderable if a one-more variation of the discrete logarithm problem is hard. We remark on some applications in signer-ambiguous confidential transaction models without trusted setup.

7 RANDOMX DESIGN

To minimize the performance advantage of specialized hardware, a proof of work (PoW) algorithm must achieve *device binding* by targeting specific features of existing general-purpose hardware. This is a complex task because we have to target a large class of devices with different architectures from different manufacturers.

There are two distinct classes of general processing devices: central processing units (CPUs) and graphics processing units (GPUs). RandomX targets CPUs for the following reasons:

- CPUs, being less specialized devices, are more prevalent and widely accessible. A CPU-bound algorithm is more egalitarian and allows more participants to join the network. This is one of the goals stated in the original CryptoNote whitepaper [1].

- A large common subset of native hardware instructions exists among different CPU architectures. The same cannot be said about GPUs. For example, there is no common integer multiplication instruction for NVIDIA and AMD GPUs [2].

- All major CPU instruction sets are well documented with multiple open source compilers available. In comparison, GPU instruction sets are usually proprietary and may require vendor specific closed-source drivers for maximum performance.

7.1 1. DESIGN CONSIDERATIONS

The most basic idea of a CPU-bound proof of work is that the "work" must be dynamic. This takes advantage of the fact that CPUs accept two kinds of inputs: *data* (the main input) and *code* (which specifies what to perform with the data).

Conversely, typical cryptographic hashing functions [3] do not represent suitable work for the CPU because their only input is *data*, while the sequence of operations is fixed and can be performed more efficiently by a specialized integrated circuit.

7.1.1**1.1 Dynamic proof of work**

A dynamic proof of work algorithm can generally consist of the following 4 steps:

1) Generate a random program.
2) Translate it into the native machine code of the CPU.
3) Execute the program.
4) Transform the output of the program into a cryptographically secure value.

The actual 'useful' CPU-bound work is performed in step 3, so the algorithm must be tuned to minimize the overhead of the remaining steps.

7.1.1.1 1.1.1 Generating a random program

Early attempts at a dynamic proof of work design were based on generating a program in a high-level language, such as C or Javascript [4, 5]. However, this is very inefficient for two main reasons:

- High level languages have a complex syntax, so generating a valid program is relatively slow since it requires the creation of an abstract syntax tree (ASL).
- Once the source code of the program is generated, the compiler will generally parse the textual representation back into the ASL, which makes the whole process of generating source code redundant.

The fastest way to generate a random program is to use a *logic-less* generator - simply filling a buffer with random data. This of course requires designing a syntaxless programming language (or instruction set) in which all random bit strings represent valid programs.

7.1.1.2 1.1.2 Translating the program into machine code

This step is inevitable because we don't want to limit the algorithm to a specific CPU architecture. In order to generate machine code as fast as possible, we need our instruction set to be as close to native hardware as possible, while still generic enough to support different architectures. There is not enough time for expensive optimizations during code compilation.

7.1.1.3 1.1.3 Executing the program

The actual program execution should utilize as many CPU components as possible. Some of the features that should be utilized in the program are:

- multi-level caches (L1, L2, L3)
- μop cache [6]
- arithmetic logic unit (ALU)
- floating point unit (FPU)
- memory controller
- instruction level parallelism [7]
  - superscalar execution [8]
  - out-of-order execution [9]
  - speculative execution [10]
  - register renaming [11]

Chapter 2 describes how the RandomX VM takes advantages of these features.

7.1.1.4 1.1.4 Calculating the final result

Blake2b [12] is a cryptographically secure hashing function that was specifically designed to be fast in software, especially on modern 64-bit processors, where it's around three times faster than SHA-3 and can run at a speed of around 3 clock cycles per byte of input. This function is an ideal candidate to be used in a CPU-friendly proof of work.

For processing larger amounts of data in a cryptographically secure way, the Advanced Encryption Standard (AES) [13] can provide the fastest processing speed because many modern CPUs support hardware acceleration of these operations. See chapter 3 for more details about the use of AES in RandomX.

7.1.2**1.2 The "Easy program problem"**

When a random program is generated, one may choose to execute it only when it's favorable. This strategy is viable for two main reasons:

1. The runtime of randomly generated programs typically follows a log-normal distribution [14] (also see Appendix C). A generated program may be quickly analyzed and if it's likely to have above-average runtime, program execution may be skipped and a new program may be generated instead. This can significantly boost performance especially in case the runtime distribution has a heavy tail (many long-running outliers) and if program generation is cheap.
2. An implementation may choose to optimize for a subset of the features required for program execution. For example, the support for some operations (such as division) may be dropped or some instruction sequences may be implemented more efficiently. Generated programs would then be analyzed and be executed only if they match the specific requirements of the optimized implementation.

These strategies of searching for programs of particular properties deviate from the objectives of this proof of work, so they must be eliminated. This can be achieved by requiring a sequence of *N* random programs to be executed such that each program is generated from the output of the previous one. The output of the final program is then used as the result.

```
          +---------------+     +---------------+               +-------
--------+     +---------------+
          |               |     |               |               |
        |     |               |
input --> |   program 1   | --> |   program 2   | -->  ...  --> | progra
m (N-1) | --> |   program N   | --> result
          |               |     |               |               |
        |     |               |
          +---------------+     +---------------+               +-------
--------+     +---------------+
```

The principle is that after the first program is executed, a miner has to either commit to finishing the whole chain (which may include unfavorable programs) or start over and waste the effort expended on the unfinished chain. Examples of how this affects the hashrate of different mining strategies are given in Appendix A.

Additionally, this chained program execution has the benefit of equalizing the runtime for the whole chain since the relative deviation of a sum of identically distributed runtimes is decreased.

7.1.3**1.3 Verification time**

Since the purpose of the proof of work is to be used in a trustless peer-to-peer network, network participants must be able to quickly verify if a proof is valid or not. This puts an upper bound on the complexity of the proof of work algorithm. In particular, we set a goal for RandomX to be at least as fast to verify as the CryptoNight hash function [15], which it aims to replace.

7.1.4**1.4 Memory-hardness**

Besides pure computational resources, such as ALUs and FPUs, CPUs usually have access to a large amount of memory in the form of DRAM [16]. The performance of the memory subsystem is typically tuned to match the compute capabilities, for example [17]:

- single channel memory for embedded and low power CPUs
- dual channel memory for desktop CPUs
- triple or quad channel memory for workstation CPUs
- six or eight channel memory for high-end server CPUs

In order to utilize the external memory as well as the on-chip memory controllers, the proof of work algorithm should access a large memory buffer (called the "Dataset"). The Dataset must be:

1. larger than what can be stored on-chip (to require external memory)
2. dynamic (to require writable memory)

The maximum amount of SRAM that can be put on a single chip is more than 512 MiB for a 16 nm process and more than 2 GiB for a 7 nm process [18]. Ideally, the size of the Dataset should be at least 4 GiB. However, due to constraints on the verification time (see below), the size used by RandomX was selected to be 2080 MiB. While a single chip can theoretically be made with this amount of SRAM using current technology (7 nm in 2019), the feasibility of such solution is questionable, at least in the near future.

7.1.4.1 1.4.1 Light-client verification

While it's reasonable to require >2 GiB for dedicated mining systems that solve the proof of work, an option must be provided for light clients to verify the proof using a much lower amount of memory.

The ratio of memory required for the 'fast' and 'light' modes must be chosen carefully not to make the light mode viable for mining. In particular, the area-time (AT) product of the light mode should not be smaller than the AT product of the fast mode. Reduction of the AT product is a common way of measuring tradeoff attacks [19].

Given the constraints described in the previous chapters, the maximum possible performance ratio between the fast and the light verification modes was empirically determined to be 8. This is because:

1. Further increase of the light verification time would violate the constraints set out in chapter 1.3.

2. Further decrease of the fast mode runtime would violate the constraints set out in chapter 1.1, in particular the overhead time of program generation and result calculation would become too high.

Additionally, 256 MiB was selected as the maximum amount of memory that can be required in the light-client mode. This amount is acceptable even for small single-board computers such as the Raspberry Pi.

To keep a constant memory-time product, the maximum fast-mode memory requirement is:

```
8 * 256 MiB = 2048 MiB
```

This can be further increased since the light mode requires additional chip area for the SuperscalarHash function (see chapter 3.4 and chapter 6 of the Specification). Assuming a conservative estimate of 0.2 mm2 per SuperscalarHash core and DRAM density of 0.149 Gb/mm2 [20], the additional memory is:

```
8 * 0.2 * 0.149 * 1024 / 8 = 30.5 MiB
```

or 32 MiB when rounded to the nearest power of 2. The total memory requirement of the fast mode can be 2080 MiB with a roughly constant AT product.

7.2 2. Virtual machine architecture

This section describes the design of the RandomX virtual machine (VM).

7.2.1 2.1 Instruction set

RandomX uses a fixed-length instruction encoding with 8 bytes per instruction. This allows a 32-bit immediate value to be included in the instruction word. The interpretation of the instruction word bits was chosen so that any 8-byte word is a valid instruction. This allows for very efficient random program generation (see chapter 1.1.1).

7.2.1.1 2.1.1 Instruction complexity

The VM is a complex instruction set machine that allows both register and memory addressed operands. However, each RandomX instructions translates to only 1-7 x86 instructions (1.8 on average). It is important to keep the instruction complexity relatively low to minimize the efficiency advantage of specialized hardware with a tailored instruction set.

7.2.2**2.2 Program**

The program executed by the VM has the form of a loop consisting of 256 random instructions.

- 256 instructions is long enough to provide a large number of possible programs and enough space for branches. The number of different programs that can be generated is limited to 2512 = 1.3e+154, which is the number of possible seed values of the random generator.

- 256 instructions is short enough so that high-performance CPUs can execute one iteration in similar time it takes to fetch data from DRAM. This is advantageous because it allows Dataset accesses to be synchronized and fully prefetchable (see chapter 2.9).

- Since the program is a loop, it can take advantage of the μop cache [6] that is present in some x86 CPUs. Running a loop from the μop cache allows the CPU to power down the x86 instruction decoders, which should help to equalize the power efficiency between x86 and architectures with simple instruction decoding.

7.2.3**2.3 Registers**

The VM uses 8 integer registers and 12 floating point registers. This is the maximum that can be allocated as physical registers in x86-64, which has the fewest architectural registers among common 64-bit CPU architectures. Using more registers would put x86 CPUs at a disadvantage since they would have to use memory to store VM register contents.

7.2.4**2.4 Integer operations**

RandomX uses all primitive integer operations that have high output entropy: addition (IADD*RS, IADD*M), subtraction (ISUB*R, ISUB*M, INEG*R), multiplication (IMUL*R, IMUL*M, IMULH*R, IMULH*M, ISMULH*R, ISMULH*M, IMUL*RCP), exclusive or (IXOR*R, IXOR*M) and rotation (IROR*R, IROL*R).

7.2.4.1 2.4.1 IADD_RS

The IADD_RS instruction utilizes the address calculation logic of CPUs and can be performed in a single hardware instruction by most CPUs (x86 `lea`, arm `add`).

7.2.4.2 2.4.2 IMUL_RCP

Because integer division is not fully pipelined in CPUs and can be made faster in ASICs, the IMUL_RCP instruction requires only one division per program to calculate the reciprocal. This forces an ASIC to include a hardware divider without giving them a performance advantage during program execution.

7.2.4.3 2.4.3 IRORR/IROLR

Rotation instructions are split between rotate right and rotate left with a 4:1 ratio. Rotate right has a higher frequency because some architecures (like ARM) don't support rotate left natively (it must be emulated using rotate right).

7.2.4.4 2.4.4 ISWAP_R

This instruction can be executed efficiently by CPUs that support register renaming/move elimination.

7.2.5**2.5 Floating point operations**

RandomX uses double precision floating point operations, which are supported by the majority of CPUs and require more complex hardware than single precision. All operations are performed as 128-bit vector operations, which is also supported by all major CPU architectures.

RandomX uses five operations that are guaranteed by the IEEE 754 standard to give correctly rounded results: addition, subtraction, multiplication, division and square root. All 4 rounding modes defined by the standard are used.

7.2.5.1 2.5.1 Floating point register groups

The domains of floating point operations are separated into "additive" operations, which use register group F and "multiplicative" operations, which use register group E. This is done to prevent addition/subtraction from becoming no-op when a small number is added to a large number. Since the range of the F group registers is limited to around `±3.0e+14`, adding or subtracting a floating point number with absolute value larger than 1 always changes at least 5 fraction bits.

Because the limited range of group F registers would allow the use of a more efficient fixed-point representation (with 80-bit numbers), the FSCAL instruction manipulates the binary representation of the floating point format to make this optimization more difficult.

Group E registers are restricted to positive values, which avoids `NaN` results (such as square root of a negative number or `0 * ∞`). Division uses only memory source operand to avoid being optimized into multiplication by constant reciprocal. The exponent of group E memory operands is set to a value between -255 and 0 to avoid division and multiplication by 0 and to increase the range of numbers that can be obtained. The approximate range of possible group E register values is `1.7E-77` to `infinity`.

Approximate distribution of floating point register values at the end of each program loop is shown in these figures (left - group F, right - group E):

(Note: bins are marked by the left-side value of the interval, e.g. bin marked `1e-40` *contains values from* `1e-40` *to* `1e-20`*.)*

The small number of F register values at `1e+14` is caused by the FSCAL instruction, which significantly increases the range of the register values.

Group E registers cover a very large range of values. About 2% of programs produce at least one `infinity` value.

To maximize entropy and also to fit into one 64-byte cache line, floating point registers are combined using the XOR operation at the end of each iteration before being stored into the Scratchpad.

7.2.62.6 Branches

Modern CPUs invest a lot of die area and energy to handle branches. This includes:

- Branch predictor unit [21]
- Checkpoint/rollback states that allow the CPU to recover in case of a branch misprediction.

To take advantage of speculative designs, the random programs should contain branches. However, if branch prediction fails, the speculatively executed instructions are thrown away, which results in a certain amount of wasted energy with each misprediction. Therefore we should aim to minimize the number of mispredictions.

Additionally, branches in the code are essential because they significantly reduce the amount of static optimizations that can be made. For example, consider the following x86 instruction sequence:

```
    ...
branch_target_00:
    ...
    xor r8, r9
    test r10, 2088960
    je branch_target_00
    xor r8, r9
    ...
```

The XOR operations would normally cancel out, but cannot be optimized away due to the branch because the result will be different if the branch is taken. Similarly, the ISWAP_R instruction could be always statically optimized out if it wasn't for branches.

In general, random branches must be designed in such way that:

1. Infinite loops are not possible.
2. The number of mispredicted branches is small.
3. Branch condition depends on a runtime value to disable static branch optimizations.

7.2.6.1 2.6.1 Branch prediction

Unfortunately, we haven't found a way how to utilize branch prediction in RandomX. Because RandomX is a consensus protocol, all the rules must be set out in advance, which includes the rules for branches. Fully predictable branches cannot depend on the runtime value of any VM register (since register values are pseudorandom and unpredictable), so they would have to be static and therefore easily optimizable by specialized hardware.

7.2.6.2 2.6.2 CBRANCH instruction

RandomX therefore uses random branches with a jump probability of 1/256 and branch condition that depends on an integer register value. These branches will be predicted as "not taken" by the CPU. Such branches are "free" in most CPU designs unless they are taken. While this doesn't take advantage of the branch predictors, speculative designs will see a significant performance boost compared to non-speculative branch handling - see Appendix B for more information.

The branching conditions and jump targets are chosen in such way that infinite loops in RandomX code are impossible because the register controlling the branch will never be modified in the repeated code block. Each CBRANCH instruction can jump up to twice in a row. Handling CBRANCH using predicated execution [22] is impractical because the branch is not taken most of the time.

7.2.72.7 Instruction-level parallelism

CPUs improve their performance using several techniques that utilize instruction-level parallelism of the executed code. These techniques include:

- Having multiple execution units that can execute operations in parallel (*superscalar execution*).
- Executing instruction not in program order, but in the order of operand availability (*out-of-order execution*).
- Predicting which way branches will go to enhance the benefits of both superscalar and out-of-order execution.

RandomX benefits from all these optimizations. See Appendix B for a detailed analysis.

7.2.82.8 Scratchpad

The Scratchpad is used as read-write memory. Its size was selected to fit entirely into CPU cache.

7.2.8.1 2.8.1 Scratchpad levels

The Scratchpad is split into 3 levels to mimic the typical CPU cache hierarchy [23]. Most VM instructions access "L1" and "L2" Scratchpad because L1 and L2 CPU caches are located close to the CPU execution units and provide the best random access latency. The ratio of reads from L1 and L2 is 3:1, which matches the inverse ratio of typical latencies (see table below).

CPU μ-architecture	L1 latency	L2 latency	L3 latency	source
ARM Cortex A55	2	6	-	[24]
AMD Zen+	4	12	40	[25]
Intel Skylake	4	12	42	[26]

The L3 cache is much larger and located further from the CPU core. As a result, its access latencies are much higher and can cause stalls in program execution.

RandomX therefore performs only 2 random accesses into "L3" Scratchpad per program iteration (steps 2 and 3 in chapter 4.6.2 of the Specification). Register values from a given iteration are written into the same locations they were loaded from, which guarantees that the required cache lines have been moved into the faster L1 or L2 caches.

Additionally, integer instructions that read from a fixed address also use the whole "L3" Scratchpad (Table 5.1.4 of the Specification) because repetitive accesses will ensure that the cache line will be placed in the L1 cache of the CPU. This shows that the Scratchpad level doesn't always directly correspond to the same CPU cache level.

7.2.8.2 2.8.2 Scratchpad writes

There are two ways the Scratchpad is modified during VM execution:

1. At the end of each program iteration, all register values are written into "L3" Scratchpad (see Specification chapter 4.6.2, steps 9 and 11). This writes a total of 128 bytes per iteration in two 64-byte blocks.

2. The ISTORE instruction does explicit stores. On average, there are 16 stores per program, out of which 2 stores are into the "L3" level. Each ISTORE instruction writes 8 bytes.

The image below shows an example of the distribution of writes to the Scratchpad. Each pixel in the image represents 8 bytes of the Scratchpad. Red pixels represent portions of the Scratchpad that have been overwritten at least once during hash calculation. The "L1" and

"L2" levels are on the left side (almost completely overwritten). The right side of the scratchpad represents the bottom 1792 KiB. Only about 66% of it are overwritten, but the writes are spread uniformly and randomly.

See Appendix D for the analysis of Scratchpad entropy.

7.2.8.3 2.8.3 Read-write ratio

Programs make, on average, 39 reads (instructions IADD*M, ISUB*M, IMUL*M, IMULH*M, ISMULH*M, IXOR*M, FADD*M, FSUB*M, FDIV_M) and 16 writes (instruction ISTORE) to the Scratchpad per program iteration. Additional 128 bytes are read and written implicitly to initialize and store register values. 64 bytes of data is read from the Dataset per iteration. In total:

- The average amount of data read from memory per program iteration is: 39 * 8 + 128 + 64 = **504 bytes**.
- The average mount of data written to memory per program iteration is: 16 * 8 + 128 = **256 bytes**.

This is close to a 2:1 read/write ratio, which CPUs are optimized for.

7.2.9**2.9 Dataset**

Since the Scratchpad is usually stored in the CPU cache, only Dataset accesses utilize the memory controllers.

RandomX randomly reads from the Dataset once per program iteration (16384 times per hash result). Since the Dataset must be stored in DRAM, it provides a natural parallelization limit, because DRAM cannot do more than about 25 million random accesses per second per bank group. Each separately addressable bank group allows a throughput of around 1500 H/s.

All Dataset accesses read one CPU cache line (64 bytes) and are fully prefetched. The time to execute one program iteration described in chapter 4.6.2 of the Specification is about the same as typical DRAM access latency (50-100 ns).

7.2.9.1 2.9.1 Cache

The Cache, which is used for light verification and Dataset construction, is about 8 times smaller than the Dataset. To keep a constant area-time product, each Dataset item is constructed from 8 random Cache accesses.

Because 256 MiB is small enough to be included on-chip, RandomX uses a custom high-latency, high-power mixing function ("SuperscalarHash") which defeats the benefits of using low-latency memory and the energy required to calculate SuperscalarHash makes light mode very inefficient for mining (see chapter 3.4).

Using less than 256 MiB of memory is not possible due to the use of tradeoff-resistant Argon2d with 3 iterations. When using 3 iterations (passes), halving the memory usage increases computational cost 3423 times for the best tradeoff attack [27].

7.3 3. Custom functions

7.3.1 3.1 AesGenerator1R

AesGenerator1R was designed for the fastest possible generation of pseudorandom data to fill the Scratchpad. It takes advantage of hardware accelerated AES in modern CPUs. Only one AES round is performed per 16 bytes of output, which results in throughput exceeding 20 GB/s in most modern CPUs.

AesGenerator1R gives a good output distribution provided that it's initialized with a sufficiently 'random' initial state (see Appendix F).

7.3.2 3.2 AesGenerator4R

AesGenerator4R uses 4 AES rounds to generate pseudorandom data for Program Buffer initialization. Since 2 AES rounds are sufficient for full avalanche of all input bits [28], AesGenerator4R has excellent statistical properties (see Appendix F) while maintaining very good performance.

The reversible nature of this generator is not an issue since the generator state is always initialized using the output of a non-reversible hashing function (Blake2b).

7.3.3 3.3 AesHash1R

AesHash was designed for the fastest possible calculation of the Scratchpad fingerprint. It interprets the Scratchpad as a set of AES round keys, so it's equivalent to AES encryption with 32768 rounds. Two extra rounds are performed at the end to ensure avalanche of all Scratchpad bits in each lane.

The reversible nature of AesHash1R is not a problem for two main reasons:

- It is not possible to directly control the input of AesHash1R.
- The output of AesHash1R is passed into the Blake2b hashing function, which is not reversible.

7.3.43.4 SuperscalarHash

SuperscalarHash was designed to burn as much power as possible while the CPU is waiting for data to be loaded from DRAM. The target latency of 170 cycles corresponds to the usual DRAM latency of 40-80 ns and clock frequency of 2-4 GHz. ASIC devices designed for light-mode mining with low-latency memory will be bottlenecked by SuperscalarHash when calculating Dataset items and their efficiency will be destroyed by the high power usage of SuperscalarHash.

The average SuperscalarHash function contains a total of 450 instructions, out of which 155 are 64-bit multiplications. On average, the longest dependency chain is 95 instructions long. An ASIC design for light-mode mining, with 256 MiB of on-die memory and 1-cycle latency for all operations, will need on average 95 * 8 = 760 cycles to construct a Dataset item, assuming unlimited parallelization. It will have to execute 155 * 8 = 1240 64-bit multiplications per item, which will consume energy comparable to loading 64 bytes from DRAM.

7.4 Appendix

7.4.1 A. The effect of chaining VM executions

Chapter 1.2 describes why `N` random programs are chained to prevent mining strategies that search for 'easy' programs. RandomX uses a value of `N = 8`.

Let's define `Q` as the ratio of acceptable programs in a strategy that uses filtering. For example `Q = 0.75` means that 25% of programs are rejected.

For `N = 1`, there are no wasted program executions and the only cost is program generation and the filtering itself. The calculations below assume that these costs are zero and the only real cost is program execution. However, this is a simplification because program generation in RandomX is not free (the first program generation requires full Scratchpad initialization), but it describes a best-case scenario for an attacker.

For `N > 1`, the first program can be filtered as usual, but after the program is executed, there is a chance of `1-Q` that the next program should be rejected and we have wasted one program execution.

For `N` chained executions, the chance is only QN that all programs in the chain are acceptable. However, during each attempt to find such chain, we will waste the execution of some programs. For `N = 8`, the number of wasted programs per attempt is equal to (1-Q)*(1+2*\Q+3*Q2+4*Q3+5*Q4+6*Q5+7*Q6) (approximately 2.5 for `Q = 0.75`).

Let's consider 3 mining strategies:

7.4.1.1 Strategy I

Honest miner that doesn't reject any programs (`Q = 1`).

7.4.1.2 Strategy II

Miner that uses optimized custom hardware that cannot execute 25% of programs (`Q = 0.75`), but supported programs can be executed 50% faster.

7.4.1.3 Strategy III

Miner that can execute all programs, but rejects 25% of the slowest programs for the first program in the chain. This gives a 5% performance boost for the first program in the chain (this matches the runtime distribution from Appendix C).

7.4.1.4 Results

The table below lists the results for the above 3 strategies and different values of `N`. The columns **N(I)**, **N(II)** and **N(III)** list the number of programs that each strategy has to execute on average to get one valid hash result (this includes programs wasted in rejected chains). Columns **Speed(I)**, **Speed(II)** and **Speed(III)** list the average mining performance relative to strategy I.

N	N(I)	N(II)	N(III)	Speed(I)	Speed(II)	Speed(III)
1	1	1	1	1.00	1.50	1.05
2	2	2.3	2	1.00	1.28	1.02
4	4	6.5	4	1.00	0.92	1.01
8	8	27.0	8	1.00	0.44	1.00

For `N = 8`, strategy II will perform at less than half the speed of the honest miner despite having a 50% performance advantage for selected programs. The small statistical advantage of strategy III is negligible with `N = 8`.

7.4.2B. Performance simulation

As discussed in chapter 2.7, RandomX aims to take advantage of the complex design of modern high-performance CPUs. To evaluate the impact of superscalar, out-of-order and speculative execution, we performed a simplified CPU simulation. Source code is available in perf-simulation.cpp.

7.4.2.1 CPU model

The model CPU uses a 3-stage pipeline to achieve an ideal throughput of 1 instruction per cycle:

```
        (1)                         (2)                     (3)
+------------------+        +----------------+       +----------------+
|   Instruction    |        |                |       |                |
|      fetch       | --->   | Memory access  | --->  |    Execute     |
|    + decode      |        |                |       |                |
+------------------+        +----------------+       +----------------+
```

The 3 stages are:

1. Instruction fetch and decode. This stage loads the instruction from the Program Buffer and decodes the instruction operation and operands.

2. Memory access. If this instruction uses a memory operand, it is loaded from the Scratchpad in this stage. This includes the calculation of the memory address. Stores are also performed in this stage. The value of the address register must be available in this stage.

3. Execute. This stage executes the instruction using the operands retrieved in the previous stages and writes the results into the register file.

Note that this is an optimistically short pipeline that would not allow very high clock speeds. Designs using a longer pipeline would significantly increase the benefits of speculative execution.

7.4.2.2 Superscalar execution

Our model CPU contains two kinds of components:

- Execution unit (EXU) - it is used to perform the actual integer or floating point operation. All RandomX instructions except ISTORE must use an execution unit in the 3rd pipeline stage. All operations are considered to take only 1 clock cycle.

- Memory unit (MEM) - it is used for loads and stores into Scratchpad. All memory instructions (including ISTORE) use a memory unit in the 2nd pipeline stage.

A superscalar design will contain multiple execution or memory units to improve performance.

7.4.2.3 Out-of-order execution

The simulation model supports two designs:

1. **In-order** - all instructions are executed in the order they appear in the Program Buffer. This design will stall if a dependency is encountered or the required EXU/MEM unit is not available.

2. **Out-of-order** - doesn't execute instructions in program order, but an instruction can be executed when its operands are ready and the required EXU/MEM units are available.

7.4.2.4 Branch handling

The simulation model supports two types of branch handling:

1. **Non-speculative** - when a branch is encountered, the pipeline is stalled. This typically adds a 3-cycle penalty for each branch.

2. **Speculative** - all branches are predicted not taken and the pipeline is flushed if a misprediction occurs (probability of 1/256).

7.4.2.5 Results

The following 10 designs were simulated and the average number of clock cycles to execute a RandomX program (256 instructions) was measured.

design	superscalar config.	reordering	branch handling	execution time [cycles]	IPC
#1	1 EXU + 1 MEM	in-order	non-speculative	293	0.87
#2	1 EXU + 1 MEM	in-order	speculative	262	0.98
#3	2 EXU + 1 MEM	in-order	non-speculative	197	1.3
#4	2 EXU + 1 MEM	in-order	speculative	161	1.6
#5	2 EXU + 1 MEM	out-of-order	non-speculative	144	1.8
#6	2 EXU + 1 MEM	out-of-order	speculative	122	2.1
#7	4 EXU + 2 MEM	in-order	non-speculative	135	1.9
#8	4 EXU + 2 MEM	in-order	speculative	99	2.6
#9	4 EXU + 2 MEM	out-of-order	non-speculative	89	2.9
#10	4 EXU + 2 MEM	out-of-order	speculative	64	4.0

The benefits of superscalar, out-of-order and speculative designs are clearly demonstrated.

7.4.3C. RandomX runtime distribution

Runtime numbers were measured on AMD Ryzen 7 1700 running at 3.0 GHz using 1 core. Source code to measure program execution and verification times is available in runtime-distr.cpp. Source code to measure the performance of the x86 JIT compiler is available in jit-performance.cpp.

7.4.3.1 Fast mode - program execution

The following figure shows the distribution of the runtimes of a single VM program (in fast mode). This includes: program generation, JIT compilation, VM execution and Blake2b hash of the register file. Program generation and JIT compilation was measured to take 3.6 μs per program.

AMD Ryzen 7 1700 can calculate 625 hashes per second in fast mode (using 1 thread), which means a single hash result takes 1600 μs (1.6 ms). This consists of (approximately):

- 1480 μs for VM execution (8 programs)
- 45 μs for initial Scratchpad fill (AesGenerator1R).
- 45 μs for final Scratchpad hash (AesHash1R).
- 30 μs for program generation and JIT compilation (8 programs)

This gives a total overhead of 7.5% (time per hash spent not executing VM).

7.4.3.2 Light mode - verification time

The following figure shows the distribution of times to calculate 1 hash result using the light mode. Most of the time is spent executing SuperscalarHash to calculate Dataset items (13.2 ms out of 14.8 ms). The average verification time exactly matches the performance of the CryptoNight algorithm.

7.4.4**D. Scratchpad entropy analysis**

The average entropy of the Scratchpad after 8 program executions was approximated using the LZMA compression algorithm:

1. Hash resuls were calculated and the final scratchpads were written to disk as files with '.spad' extension (source code: scratchpad-entropy.cpp)
2. The files were compressed using 7-Zip [29] in Ultra compression mode: `7z.exe a -t7z -m0=lzma2 -mx=9 scratchpads.7z *.spad`

The size of the resulting archive is approximately 99.98% of the uncompressed size of the scratchpad files. This shows that the Scratchpad retains high entropy during VM execution.

7.4.5E. SuperscalarHash analysis

SuperscalarHash is a custom function used by RandomX to generate Dataset items. It operates on 8 integer registers and uses a random sequence of instructions. About 1/3 of the instructions are multiplications.

The following figure shows the sensitivity of SuperscalarHash to changing a single bit of an input register:

This shows that SuperscalaHash has quite low sensitivity to high-order bits and somewhat decreased sensitivity to the lowest-order bits. Sensitivity is highest for bits 3-53 (inclusive).

When calculating a Dataset item, the input of the first SuperscalarHash depends only on the item number. To ensure a good distribution of results, the constants described in section 7.3 of the Specification were chosen to provide unique values of bits 3-53 for *all* item numbers in the range 0-34078718 (the Dataset contains 34078719 items). All initial register values for all Dataset item numbers were checked to make sure bits 3-53 of each register are unique and there are no collisions (source code: superscalar-init.cpp). While this is not strictly necessary to get unique output from SuperscalarHash, it's a security precaution that mitigates the non-perfect avalanche properties of the randomly generated SuperscalarHash instances.

7.4.6F. Statistical tests of RNG

Both AesGenerator1R and AesGenerator4R were tested using the TestU01 library [30] intended for empirical testing of random number generators. The source code is available in rng-tests.cpp.

The tests sample about 200 MB ("SmallCrush" test), 500 GB ("Crush" test) or 4 TB ("BigCrush" test) of output from each generator. This is considerably more than the amounts generated in RandomX (2176 bytes for AesGenerator4R and 2 MiB for AesGenerator1R), so failures in the tests don't necessarily imply that the generators are not suitable for their use case.

7.4.6.1 AesGenerator4R

The generator passes all tests in the "BigCrush" suite when initialized using the Blake2b hash function:

```
$ bin/rng-tests 1
state0 = 67e8bbe567a1c18c91a316faf19fab73
state1 = 39f7c0e0a8d96512c525852124fdc9fe
state2 = 7abb07b2c90e04f098261e323eee8159
state3 = 3df534c34cdfbb4e70f8c0e1826f4cf7
```

```
...

========= Summary results of BigCrush =========

 Version:          TestU01 1.2.3
 Generator:        AesGenerator4R
 Number of statistics:  160
 Total CPU time:   02:50:18.34

 All tests were passed
```

The generator passes all tests in the "Crush" suite even with an initial state set to all zeroes.

```
$ bin/rng-tests 0
state0 = 00000000000000000000000000000000
state1 = 00000000000000000000000000000000
state2 = 00000000000000000000000000000000
state3 = 00000000000000000000000000000000

...

========= Summary results of Crush =========

 Version:          TestU01 1.2.3
 Generator:        AesGenerator4R
 Number of statistics:  144
 Total CPU time:   00:25:17.95

 All tests were passed
```

7.4.6.2 AesGenerator1R

The generator passes all tests in the "Crush" suite when initialized using the Blake2b hash function.

```
$ bin/rng-tests 1
state0 = 67e8bbe567a1c18c91a316faf19fab73
state1 = 39f7c0e0a8d96512c525852124fdc9fe
state2 = 7abb07b2c90e04f098261e323eee8159
state3 = 3df534c34cdfbb4e70f8c0e1826f4cf7

...

========= Summary results of Crush =========

 Version:          TestU01 1.2.3
 Generator:        AesGenerator1R
 Number of statistics:  144
 Total CPU time:   00:25:06.07

 All tests were passed
```

When the initial state is initialized to all zeroes, the generator fails 1 test out of 144 tests in the "Crush" suite:

```
$ bin/rng-tests 0
state0 = 00000000000000000000000000000000
state1 = 00000000000000000000000000000000
state2 = 00000000000000000000000000000000
state3 = 00000000000000000000000000000000

...

========= Summary results of Crush =========

 Version:          TestU01 1.2.3
 Generator:        AesGenerator1R
 Number of statistics:  144
 Total CPU time:   00:26:12.75
 The following tests gave p-values outside [0.001, 0.9990]:
 (eps  means a value < 1.0e-300):
 (eps1 means a value < 1.0e-15):

       Test                          p-value
 ----------------------------------------------
 12  BirthdaySpacings, t = 3        1 -  4.4e-5
 ----------------------------------------------
 All other tests were passed
```

7.5 References

[1] CryptoNote whitepaper - https://cryptonote.org/whitepaper.pdf

[2] ProgPoW: Inefficient integer multiplications - https://github.com/ifdefelse/ProgPOW/issues/16

[3] Cryptographic Hashing function - https://en.wikipedia.org/wiki/Cryptographic*hash*function

[4] randprog - https://github.com/hyc/randprog

[5] RandomJS - https://github.com/tevador/RandomJS

[6] µop cache - https://en.wikipedia.org/wiki/CPU*cache#Micro-operation*(%CE%BCop*or*uop)_cache

[7] Instruction-level parallelism - https://en.wikipedia.org/wiki/Instruction-level_parallelism

[8] Superscalar processor - https://en.wikipedia.org/wiki/Superscalar_processor

[9] Out-of-order execution - https://en.wikipedia.org/wiki/Out-of-order_execution

[10] Speculative execution - https://en.wikipedia.org/wiki/Speculative_execution

[11] Register renaming - https://en.wikipedia.org/wiki/Register_renaming

[12] Blake2 hashing function - https://blake2.net/

[13] Advanced Encryption Standard - https://en.wikipedia.org/wiki/Advanced*Encryption*Standard

[14] Log-normal distribution - https://en.wikipedia.org/wiki/Log-normal_distribution

[15] CryptoNight hash function - https://cryptonote.org/cns/cns008.txt

[16] Dynamic random-access memory - https://en.wikipedia.org/wiki/Dynamic*random-access*memory

[17] Multi-channel memory architecture - https://en.wikipedia.org/wiki/Multi-channel*memory*architecture

[18] Obelisk GRN1 chip details - https://www.grin-forum.org/t/obelisk-grn1-chip-details/4571

[19] Biryukov et al.: Tradeoff Cryptanalysis of Memory-Hard Functions - https://eprint.iacr.org/2015/227.pdf

[20] SK Hynix 20nm DRAM density - http://en.thelec.kr/news/articleView.html?idxno=20

[21] Branch predictor - https://en.wikipedia.org/wiki/Branch_predictor

[22] Predication - https://en.wikipedia.org/wiki/Predication*(computer*architecture)

[23] CPU cache - https://en.wikipedia.org/wiki/CPU_cache

[24] Cortex-A55 Microarchitecture - https://www.anandtech.com/show/11441/dynamiq-and-arms-new-cpus-cortex-a75-a55/4

[25] AMD Zen+ Microarchitecture - https://en.wikichip.org/wiki/amd/microarchitectures/zen%2B#Memory_Hierarchy

[26] Intel Skylake Microarchitecture - https://en.wikichip.org/wiki/intel/microarchitectures/skylake*(client)#Memory*Hierarchy

[27] Biryukov et al.: Fast and Tradeoff-Resilient Memory-Hard Functions for Cryptocurrencies and Password Hashing - https://eprint.iacr.org/2015/430.pdf Table 2, page 8

[28] J. Daemen, V. Rijmen: AES Proposal: Rijndael - https://csrc.nist.gov/csrc/media/projects/cryptographic-standards-and-guidelines/documents/aes-development/rijndael-ammended.pdf page 28

[29] 7-Zip File archiver - https://www.7-zip.org/

[30] TestU01 library - http://simul.iro.umontreal.ca/testu01/tu01.html

A SECURITY

We shall give a proof for our one-time ring signature scheme. At some point it coincides with the parts of the proof in [24], but we decided to rewrite them with a reference rather than to force a reader to rush about from one paper to another. These are the properties to be established:

- **Linkability.** Given all the secret keys$\{x_i\}_{i=1}^{n}$ for a set S it is impossible to produce $n+1$ valid signatures $\sigma_1,\sigma_2,\ldots,\sigma_{n+1}$, such that all of them pass the **LNK** phase (i.e. with $n+1$ different key images I_i). This property implies the double spending protection in the context of CryptoNote.

- **Exculpability.** Given set S, at most $n-1$ corresponding private keys x_i (excluding $i = j$) and the image I_j of the keys x_j it is impossible to produce a valid signature σ with I_j. This property implies theft protection in the context of CryptoNote.

- **Unforgeability.** Given only a public keys set S it is impossible to produce a valid signature σ.

- **Anonymity.** Given a signature σ and the corresponding set S it is impossible to determine the secret index j of the signer with a probability$p > \frac{1}{n}$.

LINKABILITY

Theorem 1. *Our one-time ring signature scheme is linkable under the random oracle model.*

Proof. Suppose an adversary can produce $n + 1$ valid signatures σ_i with key images $I_i 6= I_j$ for any $i,j \in [1\ldots n]$. Since $\#S = n$, at least one $I_i 6= x_i\mathrm{H}_p(P_i)$ for every i. Consider the corresponding signature $\sigma = (I,c_1,\ldots,c_n,r_1,\ldots,r_n)$. **VER**$(\sigma)$ = "true", this means that

$$LR_{0ii0} == rr_{ii}G\mathrm{H}_p+(Pc_{ii}) + P_i\, c_i I$$

$$\mathop{\mathrm{P}}_{i=1}^{n} c_i = \mathrm{H}_s(m,L^0{}_1,\ldots,L^0{}_n,R_1{}^0,\ldots,R_n{}^0) \qquad \bmod l$$

The first two equalities imply

$$(\log_G L_{0i} = r_i + c_i x_i \log_{\mathrm{H}_p(P_i)}$$

$$R_i^0 = r_i + c_i \log_{H_p(P_i)} I$$

where $\log_A B$ informally denotes the discrete logarithm of B to the base A.

As in [24] we note that @i : $x_i = \log_{H_p(P_i)} I$ implies that all c_i's are uniquely determined. The third equality forces the adversary to find a pre-image of H_s to succeed in the attack, an event whose probability is considered to be negligible. □

EXCULPABILITY

Theorem 2. *Our one-time ring signature scheme is exculpable under the discrete logarithm assumption in the random oracle model.*

Proof. Suppose an adversary can produce a valid signature $\sigma = (I, c_1, ..., c_n, r_1, ..., r_n)$ with $I = x_j H_P(P_j)$ with given $\{x_i \mid i = 1, ..., j-1, j+1, ..., n\}$. Then, we can construct an algorithm A which solves the discrete logarithm problem in $E(\mathsf{F}_q)$.

Suppose inst = $(G,P) \in E(\mathsf{F}_q)$ is a given instance of the DLP and the goal is to get s, such that $P = sG$. Using the standard technique (as in [24]), A simulates the random and signing oracles and makes the adversary produce two valid signatures with $P_j = P$ in the set S:

$\sigma = (I, c_1, ..., c_n, r_1, ..., r_n)$ and $\sigma' = (I, c'_1, \ldots, c'_n, r'_1, \ldots, r'_n)$. ′

Since $I = x_j H_p(P_j)$ in both signatures we compute $x_j = \log_{\mathcal{H}_p(P_j)} I = \frac{r_j - r_j}{c'_j - c_j} \bmod l$

A outputs x_j because $L_j = r_j G + c_j P_j = r'_j G + c'_j P_j$ and $P_j = P$. □

UNFORGEABILITY

It has been shown in [24] that unforgeability is just an implication of both linkability and exculpability.

Theorem 3. *If a one-time ring signature scheme is linkable and exculpable, then it is unforgeable.*

Proof. Suppose an adversary can forge a signature for a given set S: $\sigma_0 = (I_0, ...)$. Consider all valid signatures (produced by the honest signers) for the same message m and the set S: $\sigma_1, \sigma_2, ..., \sigma_n$. There are two possible cases:

1. $I_0 \in \{I_i\}_{i=1}^n$. Which contradicts exculpability.
2. $I_0 \notin \{I_i\}_{i=1}^n$. Which contradicts linkability. □

ANONYMITY

Theorem 4. *Our one-time ring signature scheme is anonymous under the decisional DiffieHellman assumption in the random oracle model.*

Proof. Suppose an adversary can determine the secret index j of the Signer with a probability $p = \frac{1}{n} + \epsilon$. Then, we can construct algorithm A which solves the decisional Diffie-Hellman problem in $E(\mathsf{F}_q)$ with the probability $\frac{1}{2} + \frac{\epsilon}{2}$.

Let inst = $(G_1,G2,Q_1,Q_2) \in E(\mathsf{F}_q)$ be the instance of DDH and the goal to determine if $\log_{G_1} Q_1 = \log_{G_2} Q_2$. A feeds the adversary with valid signature $\sigma_0 = (I,...)$, where $P_j = x_j G_1 = Q_1$ and $I = Q_2$ and simulates oracle H_p, returning G_2 for query $\mathrm{H}_p(P_j)$.

The adversary returns k as his guess for the index i: $I = x_i \mathrm{H}_P(P_i)$. If $k = j$, then A returns 1 (for "yes") otherwise a random $r \in \{1,0\}$. The probability of the right choice is computed as in [24]: $\frac{1}{2} + \Pr(1 \mid \text{inst} \in DDH) - \Pr(1 \mid \text{inst} \notin DDH) = \frac{1}{2} + \Pr(k = j \mid \text{inst} \in DDH) +$

$\Pr(k\ 6{=} j \mid \text{inst} \in DDH) \cdot \Pr(r = 1) - \Pr(k = j \mid \text{inst} \in / \ DDH) - \Pr(k = 6\, j \mid \text{inst} \in / \ DDH) \cdot \Pr(r = 0) =$

$\frac{1}{2} + \frac{1}{n} + \epsilon + (\frac{-1}{n} - \epsilon) \cdot \frac{1}{2} - \frac{1}{n} - \frac{-1}{n} \cdot \frac{1}{2} = \frac{1}{2} + \frac{\epsilon}{2n}$ n

In fact, the result should be reduced by the probability of collision in H_s, but this value is considered to be negligible. □

NOTES ON THE HASH FUNCTION H_P

We defined H_p as deterministic hash function $E(\mathsf{F}_q) \to E(\mathsf{F}_q)$. None of the proofs demands H_p to be an ideal cryptographic hash function. It's main purpose is to get a pseudo-random base for image key $I = x\mathrm{H}_p(xG)$ in some determined way.

With fixed base ($I = xG_2$) the following scenario is possible:

1. Alice sends two standard transactions to Bob, generating one-time tx-keys: $P_2 = \mathrm{H}_s(r_1A)G + B$ and $P_1 = \mathrm{H}_s(r_2A)G + B$.

2. Bob recovers corresponding one-time private tx-keys x_1 and x_2 and spends the outputs with valid signatures and images keys $I_1 = x_1G_2$ and $I_2 = x_2G_2$.

3. Now Alice can link these signatures, checking the equality $I_1 - I_2 = (^? \mathrm{H}_s(r_1A) - \mathrm{H}_s(r_2A))G_2$.

The problem is that Alice knows the linear correlation between public keys P_1 and P_2 and in case of fixed base G_2 she also gets the same correlation between key images I_1 and I_2. Replacing G_2 with $\mathrm{H}_p(xG_2)$, which does not preserve linearity, fixes that flaw.

For constructing deterministic H_p we use algorithm presented in [37].

REFERENCES

[1] http://bitcoin.org.

[2] https://en.bitcoin.it/wiki/Category:Mixing Services.

[3] http://blog.ezyang.com/2012/07/secure-multiparty-bitcoin-anonymization.

[4] https://bitcointalk.org/index.php?topic=279249.0.

[5] http://msrvideo.vo.msecnd.net/rmcvideos/192058/dl/192058.pdf.

[6] https://github.com/bitcoin/bips/blob/master/bip-0034.mediawiki#Specification.

[7] https://github.com/bitcoin/bips/blob/master/bip-0016.mediawiki#Backwards_Compatibility.

[8] https://en.bitcoin.it/wiki/Mining hardware comparison.

[9] https://github.com/bitcoin/bips/blob/master/bip-0050.mediawiki.

[10] http://luke.dashjr.org/programs/bitcoin/files/charts/branches.html.

[11] https://bitcointalk.org/index.php?topic=196259.0.

[12] https://en.bitcoin.it/wiki/Contracts.

[13] https://en.bitcoin.it/wiki/Script.

[14] http://litecoin.org.

[15] Mart´ın Abadi, Michael Burrows, and Ted Wobber. Moderately hard, memory-bound functions. In *NDSS*, 2003.

[16] Ben Adida, Susan Hohenberger, and Ronald L. Rivest. Ad-hoc-group signatures from hijacked keypairs. In *in DIMACS Workshop on Theft in E-Commerce*, 2005.

[17] Man Ho Au, Sherman S. M. Chow, Willy Susilo, and Patrick P. Tsang. Short linkable ring signatures revisited. In *EuroPKI*, pages 101–115, 2006.

[18] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *J. Cryptographic Engineering*, 2(2):77–89, 2012.

[19] David Chaum and Eug`ene van Heyst. Group signatures. In *EUROCRYPT*, pages 257–265, 1991.

[20] Fabien Coelho. Exponential memory-bound functions for proof of work protocols. *IACR Cryptology ePrint Archive*, 2005:356, 2005.

[21] Ronald Cramer, Ivan Damg˚ard, and Berry Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. In *CRYPTO*, pages 174–187, 1994.

[22] Cynthia Dwork, Andrew Goldberg, and Moni Naor. On memory-bound functions for fighting spam. In *CRYPTO*, pages 426–444, 2003.

[23] Eiichiro Fujisaki. Sub-linear size traceable ring signatures without random oracles. In *CTRSA*, pages 393–415, 2011.

[24] Eiichiro Fujisaki and Koutarou Suzuki. Traceable ring signature. In *Public Key Cryptography*, pages 181–200, 2007.

[25] Jezz Garzik. Peer review of "quantitative analysis of the full bitcoin transaction graph". https://gist.github.com/3901921, 2012.

[26] Joseph K. Liu, Victor K. Wei, and Duncan S. Wong. Linkable spontaneous anonymous group signature for ad hoc groups (extended abstract). In *ACISP*, pages 325–335, 2004.

[27] Joseph K. Liu and Duncan S. Wong. Linkable ring signatures: Security models and new schemes. In *ICCSA (2)*, pages 614–623, 2005.

[28] Ian Miers, Christina Garman, Matthew Green, and Aviel D. Rubin. Zerocoin: Anonymous distributed e-cash from bitcoin. In *IEEE Symposium on Security and Privacy*, pages 397– 411, 2013.

[29] Micha Ober, Stefan Katzenbeisser, and Kay Hamacher. Structure and anonymity of the bitcoin transaction graph. *Future internet*, 5(2):237–250, 2013.

[30] Tatsuaki Okamoto and Kazuo Ohta. Universal electronic cash. In *CRYPTO*, pages 324–337, 1991.

[31] Marc Santamaria Ortega. The bitcoin transaction graph — anonymity. Master's thesis, Universitat Oberta de Catalunya, June 2013.

[32] Colin Percival. Stronger key derivation via sequential memory-hard functions. Presented at BSDCan'09, May 2009.

[33] Fergal Reid and Martin Harrigan. An analysis of anonymity in the bitcoin system. *CoRR*, abs/1107.4524, 2011.

[34] Ronald L. Rivest, Adi Shamir, and Yael Tauman. How to leak a secret. In *ASIACRYPT*, pages 552–565, 2001.

[35] Dorit Ron and Adi Shamir. Quantitative analysis of the full bitcoin transaction graph. *IACR Cryptology ePrint Archive*, 2012:584, 2012.

[36] Meni Rosenfeld. Analysis of hashrate-based double-spending. 2012.

[37] Maciej Ulas. Rational points on certain hyperelliptic curves over finite fields. *Bulletin of the Polish Academy of Sciences. Mathematics*, 55(2):97–104, 2007.

[38] Qianhong Wu, Willy Susilo, Yi Mu, and Fangguo Zhang. Ad hoc group signatures. In *IWSEC*, pages 120–135, 2006.