VILNIUS UNIVERSITY
FACULTY OF MATHEMATICS AND INFORMATICS
INSTITUTE OF COMPUTER SCIENCE
DEPARTMENT OF COMPUTATIONAL AND DATA MODELING

Master Thesis Project

Anonimity-first time locks using Proof of Work

Done by:
Martynas Maciulevičius signature

Supervisor:
doc. dr. Vilius Stakėnas

Vilnius
2020

Contents

Keywords

ECC – Elliptic curve cryptography
ECIES – Elliptic curve integrated encryption scheme
JSON – JavaScript Object Notation
LAC – Lattice-based Cryptosystems
LAC.CPA – Public key encryption scheme based on Lattice LWE problem
LWE – Lattice learning with errors problem
NIST – American National Institute of Standards and Technology
NP – Non-polynomial time computational complexity problem.
PoW – Proof of Work consensus mechanism
SHA256 – Cryptographic hash function designed by the United States National Security Agency.
SMA – Simple moving average
SPV – Lightweight Bitcoin client
SVP – Lattice shortest vector problem
Time-release cryptography – Cryptography intended for time-based release of data
VDF – Verifiable Delay Function
WIF – Wallet import format
WT – Wavelet transform

Abstract

Proof of Work (PoW) is a system that's used by Bitcoin to keep it's state update intervals at around 10 minutes. PoW uses hashing techniques to produce validation of state which are required by PoW mechanism. This hash-based mechanism has it's pros, cons and inefficiencies. One of the inefficiencies would be that hashes are not reusable anywhere else. This thesis presents an alternative to a hash-based validation that's based on asymmetric cryptography. Several difficulty calculation and block validation methods are explored to send messages into the future by encrypting them with temporary keys. Public key pool is maintained by the system in such a way that it can be done publically and without trust and without knowing any private keys. Later private keys are found and revealed and only then time-lock message's decryption becomes available.

Santrauka

Anoniminis žinučių įšaldymas laike naudojant Proof of Work

Atsiradus Proof of Work (PoW) technologijai, kurią naudoja Bitcoin sistema 10 minučių laiko intervalams išlaikyti, tapo prieinama konstruoti panašias sistemas. PoW naudoja Hash validacijos schemą, kuri turi savų privalumų, trūkumų ir neefektyvumų. Vienas iš neefektyvumų yra menka galimybė panaudoti Hash validaciją kitiems tikslams, pvz.: prieinamam duomenų šifravimui. Šiame darbe nagrinėjami keli metodai leidžiantys pakeisti PoW veikimą iš Hash algoritmo į asimetrinę kriptosistemą. Tai leidžia sukurti trumpalaikius atvirus raktus ir juos panaudoti žinučių siuntimui į netolimą ateitį. Raktai yra generuojami fiksuotais laiko intervalais ir tokiu būdu, kad neprivaloma slėpti sistemos komponentų. Galiausiai tinklas atranda raktus ir juos gali atskleisti. Tik tada yra prieinamas duomenų atšifravimas.

Introduction

The goal of time-release cryptography is to send a message into the future so that it could be decrypted after a certain time. Strategies for this range from secure escrow for holding the key to processor-intensive tasks that give an answer after a specified amount of time. The usecases range from short-term time escrow such as blind no-authority auctions or elections to long-term storage like wealth preservation for cryogenically cooled people [26].

This kind of escrow system could be constructed in several ways, ranging from trusted storage to private key brute-force. It's possible to design a trust-based service to reveal the key but then trust would be an important factor because key would be known in advance.

Alternatively an a priori unknown private key would allow for more public components in the system. Publically computing the key would require costly computations and various security assumptions (see chapter 9), but in return it would reduce trusted authorities to the minimum.

With recent development in Proof of Work (PoW) that's used by Bitcoin and similar networks time-release cryptography may get another push forward. As Bitcoin manages to keep block intervals around 10 minutes and adjusts it's difficulty every 2016 blocks (which is about two weeks) it looks promising that it may be used for a system such as a short-term time-release network. The system uses exhaustive amounts of computing power in it's progress validation which allows coordination of a large amount of interconnected parties without a trusted authority[1].

This thesis extends the computing-based approach of PoW and explores an alternative to hash-based validation that's based on asymmetric cryptography. Weak asymmetric cryptosystems are not immune against a sufficient amount of computing power. This may mean that block validation and difficulty calculation mechanisms could be refactored to use asymmetric cryptography.

This thesis proposes several core changes to PoW mechanism. Block validation on Bitcoin network uses SHA256 hashing for block validation and the proposal of this thesis is to use asymmetric keys instead of cryptographic hashes. It's not enough to blindly change validation into asymmetric puzzles (defined in chapter 2) as it wouldn't give any benefits in this primitive form. Difficulty adjustment freeze interval (which lasts for 2016 blocks in case of Bitcoin[1]) can be used to provide a constantly refillable pool of uncracked public keys (see chapters 4, 5 and a minimal implementation on chapter 7). This key pool can then be used to encrypt secret sharing schemes which would then become the actual time-locks (more in chapter 6).

The mechanism would yield a side benefit of time-release messaging that wouldn't have any time-lock creation frequency constraints. The messages would become decryptable only after network computes and publishes the corresponding keys. Additionally each time-lock would be completely anonymous because time-lock encryption users would not use this network for message hosting or exchange.

This kind of timestamping scheme could be used to implement a fork of Bitcoin or a completely new system. As it's an extension to PoW it could also be used to create privacy oriented blockchains[1] or even smart contract platforms[2].

[1] Similar to ZCash, Monero or Dash.

[2] Ethereum is a PoW-based smart contract platform.

1 Related work

1.1 Proof of work

Proof of work is a specific type of brute force coordination. It uses a network (i.e. botnet) of connected devices to parallelize brute-force computations. The algorithm tries to guess the next difficulty of the problem using previous executions as a benchmark. This allows dynamic addition and removal of nodes from the network by adjusting the difficulty of brute-force problems over time. The difficulty always lags behind and is only changed by adding or removing the amount of computation [1].

1.2 Public-key cryptography

Public key cryptography is a cryptographic protocol that allows two parties exchange information without negotiating anything in advance. It uses a mathematical problem to hide the transferred information. The receiver party has an ability to decipher the transferred data.

1.3 El Gamal encryption scheme

El Gamal encryption scheme is based on a difficulty of discrete logarithm computation problem over finite fields [5].

1.3.1 Key generation

Each participating entity should do the following [18]:

- Choose a group G and a generator g for that group.
- Generate a key pair.

1.3.2 Choosing a group and a generator

Group $G : \{g^j\}; j \in \{0, ..., p-1\}$ can be chosen by choosing a large integer prime p with order $n = p - 1$. Prime p choice depends on the application. Badly chosen group prime p can increase weaknesses such as easier computation of discrete logarithms using Pohlig-Hellman algorithm [18].

This algorithm is general for cyclic group cryptosystems [18]:

Algorithm 1. Finding generator of a cyclic group [18].

Require: A cyclic group G of order n, and the prime factorization $n = p_1^{e1} p_2^{e2} ... p_k^{ek}$.
Ensure: : A generator of G.

1: Choose a random element g in G.
2: **for** i from 1 to k **do**
3: Compute $b \leftarrow g^{n/p_i}$.
4: If $b = 1$ then go to step 1.
5: **end for**
6: **return** g.

The algorithm 1 filters out elements of a group that generate the whole group $\mathbb{Z}_p^*$.

Example: Let's choose $p = 11$ as our definition of a cyclic group. p is prime and $n = 10$ so it has only two factors: 2 and 5.

Let's choose a random group entry $g = 3$. To check if it's a valid generator we have to perform line 3 of algorithm 1:

$$\begin{aligned} &3^2 = 9 \ (mod\ 11) \\ &3^5 = 243 = 1 \ (mod\ 11) \end{aligned} \tag{1.1}$$

Equation 1.1 shows order factor check using algorithm 1. One of the results evaluated to 1 so 3 is not a group generator.

Let's choose $g = 2$.

$$\begin{aligned} &2^2 = 4 \ (mod\ 11) \\ &2^5 = 32 = 10 \ (mod\ 11) \end{aligned} \tag{1.2}$$

Equation 1.2 didn't produce any results of 1 so 2 is a valid generator.

Example with a non-generator: Let's choose $p = 11$ and a non whole-group generator $g = 3$ and list all possible elements of a group for this generator. After moving through first five powers of g the values start to repeat. As 3 is not an efficient generator only 5 out of all possible elements of group G are generated:

$$\begin{aligned} &G_{11,3} : \{g^n\}; n \in \{0, ..., p-1\} \\ &G_{11,3} : \{1, 3, 9, 5, 4, 1, 3, 9, ...\} \\ &G_{11,3} : \{1, 3, 9, 5, 4\} \end{aligned} \tag{1.3}$$

Example with a valid generator: Let's choose $p = 11$ with a generator $g = 2$ and list all possible elements of a group for this generator. After moving through the powers of g the values start to repeat only after all of the values of G are exhausted:

$$\begin{aligned} &G_{11,2} : \{1, 2, 4, 8, 5, 10, 9, 7, 3, 6, 1, ...\} \\ &G_{11,2} : \{1, 2, 4, 8, 5, 10, 9, 7, 3, 6\} \\ &sorted(G_{11,2}) : \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\} \end{aligned} \tag{1.4}$$

We've just generated all possible public keys using generator 2 (except 0 and $p = 11$). I.e. if any of these group elements would be chosen as public keys (with generator $g = 2$) then it would be possible to eventually find out the private key.

1.3.3 Key pair generation

Derivation of a public-private key pair is performed using algorithm 2.

Algorithm 2. El Gamal public key generation [18].

1: Choose a description of a cyclic group G of prime order p with generator $g \in \mathbb{Z}_p^* : \{0, ..., p-1\}$.
2: Choose an integer $x \in \{1, ..., p-1\}$.
3: Compute $h := g^x$.
4: **return** (p, g, h) and x.

The public key consists of the values (p, g, h) and x is the private key. Note: Algorithm 1 doesn't have parameter p and uses $n = p - 1$ for El Gamal's case.

Example: Let's choose $p = 11$ and a generator $g = 3$. To create a key pair we have to choose a private key x:

$$\begin{aligned} &x = 5 \\ &h = g^x = 3^5 = 243 = 1 \ (mod\ 11) \\ &(h = 1; x = 5) \ (mod\ 11) \end{aligned} \tag{1.5}$$

1.3.4 Encryption

ElGamal encryption is performed by executing algorithm 3.

Algorithm 3. B encrypts a message m for A (algorithm 8.26 from [18]):

1: Obtain A's authentic public key $(g; g^x)$.
2: Represent the message as an element m of the group G.
3: Select a random integer k, $1 \leq k \leq n - 1$.
4: Compute $\gamma = g^k$ and $\delta = m \cdot (g^x)^k$.
5: Send $c = (\gamma, \delta)$ to A.

1.3.5 Decryption

ElGamal decryption is performed by executing algorithm 4.

Algorithm 4. A recovers plaintext m from c (algorithm 8.26 from [18]):

1: Use the private key x to compute γ^x and then compute γ^{-x}.
2: Recover m by computing $(\gamma^{-x}) \cdot \delta$.

1.4 Elliptic curve cryptography

Elliptic curve cryptography is based on Elliptic curve discrete logarithm problem. Currently it's the standard choice for secure communication [9]. Elliptic curves offer many well-understood and widely used cryptographic concepts.

1.5 Lattice-based cryptography

Lattice-based cryptography uses hard problems in lattices for security. This mathematical concept is different from discrete logarithm problem that is used in El Gamal and Elliptic curve cryptosystems. One could imagine a lattice as a way to define multiple linearly independent vectors or different multivariate polynomials and their relations in multidimensional space. These relations can be interpreted as points of a space. This is where the name "Lattice" comes from.

Lattices have several hard problems that are used in modern cryptography that include:

- Shortest vector problem (SVP) – Find shortest vector of γ times shortest non-zero lattice vector [19]
- Closest vector problem (CVP) – Find a lattice point near given non-lattice point [19]
- Learning with errors (LWE) problem which is derived from SVP [25]

and others.

Multiple lattice-based asymmetric encryption algorithms are proposed to NIST's Post-Quantum Cryptography project which include:

- CRYSTALS-KYBER (LWE)
- FrodoKEM (LWE)
- LAC.CPA (LWE)
- NTRUEncrypt (SVP [10])
- Saber (module-LWR [13])
- Three Bears (LWE).

1.6 LAC CPA public-key encryption scheme

LAC suite offers multiple cryptographic primitives including key exchange and public key cryptosystem. LAC.CPA [17] is a lattice-based public key cryptosystem based on LAC key exchange mechanism. It's based on a hard lattice problem known as Learning With Errors (LWE or Ring-LWE).

LAC.CPA encryption algorithm doesn't use complicated operations (ex. modulo inverse[3]) so it may be a good candidate for AVDFs. It uses only addition, multiplication and sampling from error distributions. Errors that get introduced during the steps obfuscate operations of the mechanism and can hide messages. They are sampled from distributions around points in a lattice and from general randomness. Encryption and decryption is performed with help of error correcting codes[4] to recover obfuscated parameters from random errors.

1.6.1 Key generation

Distribution $S : \{0,1\}^{l_s}$ is a space of random seeds where l_s is a positive length of a seed. l_s doesn't participate in the key generation algorithm.

Operation $x \leftarrow Samp(D, seed)$ is an abstract random generator that returns a sample from a distribution D for a given seed. Equivalently $(x_1, x_2, \ldots, x_t) \leftarrow Samp(D_1, D_2, \ldots, D_t; seed)$ returns t samples.

Let q be a modulus and define a polynomial ring $R_q = \mathbb{Z}_q/(x^n - 1)$. $U(R_q)$ a uniform distribution over R_q.

$\Psi_\sigma^{n,h}$ is a distribution that is constructed in such a way that some of the space around a specific point can be sampled for points. Authors of LAC.CPA provide binomial and Gaussian distributions as examples and suggest to use binomial distribution to improve performance.

Dollar sampling notation from step 1 refers to coordinate-wise sampling[5]. Operation in step 5 is vector multiplication that results in an element of field R_q. It produces a public key b.

[3] NTRU uses modulo inverse[10].

[4] Authors use BCH[24] error correction codes.

[5] Coordinate-wise sampling: https://crypto.stackexchange.com/a/37797.

Algorithm 5. LAC.CPA.KG() [17]

Ensure: A pair of public key and secret key (pk, sk).

1: $seed_a \xleftarrow{\$} S$
2: $a \leftarrow Samp(U(R_q); seed_a) \in R_q$
3: $s \xleftarrow{\$} \Psi_\sigma^{n,h}$
4: $e \xleftarrow{\$} \Psi_\sigma^{n,h}$
5: $b \leftarrow as + e \in R_q$
6: **return** $(pk := (seed_a, b), sk := s)$

1.6.2 Encryption

Algorithm 6 describes the steps for encryption. Encryption is performed by encoding the message m into a error correcting code $\widehat{m}$ with l_v as it's length. The payload is then adjusted for decryption decoding and multiplied by the public key b and other random parameters.

Algorithm 6. LAC.CPA.Enc$(pk = (seed_a, b), m \in \{0,1\}^{l_m}; seed \in S)$ [17]

Ensure: A ciphertext c.

1: $a \leftarrow Samp(U(R_q); seed_a) \in R_q$
2: $\widehat{m} \leftarrow ECCEnc(m) \in \{0,1\}^{l_v}$
3: $(r, e_1, e_2) \leftarrow Samp(\Psi_\sigma^{n,h}, \Psi_\sigma^{n,h}, \Psi_\sigma^{l_v}; seed)$
4: $c_1 \leftarrow ar + e_1 \in R_q$
5: $c_2 \leftarrow (br)_{l_v} + e_2 + \lfloor \frac{q}{2} \rceil \cdot \widehat{m} \in \mathbb{Z}_q^{l_v}$
6: **return** $c := (c_1, c_2) \in R_q \times \mathbb{Z}_q^{l_v}$

1.6.3 Decryption

Decryption algorithm 7 derives parameter u from secret key component and removes ciphering components from the ciphertext. This is done by matching between the error values in a range because $\frac{q}{2}$ was added to every component of ciphertext during encryption. Line 10 of the algorithm uses error correcting decoding to recover the final plaintext.

Algorithm 7. LAC.CPA.Dec$(sk = s, c = (c_1, c_2))$ [17]

Ensure: A plaintext m.

1: $u \leftarrow c_1 s \in R_q$
2: $\widetilde{m} \leftarrow c_2 - (u)_{l_v} \in Z_q^{l_v}$
3: **for** $i = 0$ to $l_v - 1$ **do**
4: **if** $\frac{q}{4} \leq \widetilde{m}_i < \frac{3q}{4}$ **then**
5: $\widehat{m}_i \leftarrow 1$
6: **else**
7: $\widehat{m}_i \leftarrow 0$
8: **end if**
9: **end for**
10: $m \leftarrow ECCDec(\widehat{m})$
11: **return** m

1.7 Delay mechanisms

Delay mechanisms ensure that computation cannot continue until specific period of time passes. This can be achieved in several ways:

- Reliance on a third party
- Classic slow function
- Verifiable delay function (VDF)

Reliance on a third party is a trust-based scheme for unveiling data. Third party can agree to hold keys for specific time and needs to be trusted not to disclose them. It can be combined with a secret sharing mechanism so that the risk of premature disclosure would be reduced by spreading the trust between multiple third parties [28].

Classic slow functions are CPU-intensive tasks that occupy a single core and run for a specified amount of time. The execution yields a value that can be used as a solution of the domain problem (e.g. a secret key for exchange of company data). The main drawback of this kind of approach is that validation of the performed computation is expensive [26].

VDFs are similar to slow functions but they add fast verification of the solution. This means that the verifier doesn't need to reevaluate the work performed by the solver [3].

1.8 Verifiable delay function

Verifiable delay function is a busy wait puzzle function that yields a value. To call function a Verifiable Delay Function (VDF) it has to allow efficient verification of its result [3]:

$$\begin{aligned} puzzle &= Setup(D) \\ answer &= Eval(D, puzzle) \\ result &= Verify(D, puzzle, answer) \end{aligned} \tag{1.6}$$

Where $Setup$ is puzzle creation with D initial parameter(s) for the puzzle, $Eval$ is a resource-intensive step and $Verify$ is verification function for the $answer$.

1.9 Sending messages into the future

It's very easy to take a cryptographic key and store it in a vault for some time. One could construct a system which allows users to rely on a third party. This party then would hold the keys and would be responsible to reveal them at specified moment.

This kind of naive system may work for some usecases but it wouldn't be private enough for very sensitive applications. The sensitivity of information would eventually incur that some form of insider would be able to read the data.

Prevention of this kind of behavior could be to have a specific contract with the provider but even then nobody could be sure. To increase this kind of certainty the private keys shouldn't exist anywhere before the specified time.

This concept of sending messages into the future is already known for a while[26]. And there are many already explored ways to do it [23, 16].

One of the concepts that may deliver at least short-term results would be a proof of work public brute-force system [12, 4, 1]. Using this kind of mechanism the constructed system would not allow anybody to decrypt the information without producing enough effort for PoW verifications.

1.10 Secret sharing scheme

Secret sharing schemes allow sharing a secret piece of information among a group of participants. This kind of schemes[29, 28] allow to split secret information into several parts with ability to recover it when enough participants cooperate.

1.10.1 Shamir's secret sharing

Shamir's secret sharing scheme is based on interpolation of a polynomial over a field $GF(q)$[28]. Every participant is given a point and the final polynomial is recovered using interpolation. Output of sharing scheme secret's recovery is the constant coefficient of the polynomial that isn't multiplied by variable x. The scheme has these properties:

- Valid threshold scheme – Any set of t participants can reconstruct the secret value s when it's constructed for t out of n total shares.
- Perfect – shares of $t - 1$ participants don't reveal any information about the secret s.
- Ideal – every share has the same length as the secret.
- Linear – shares are linear combinations[6] of the secret and random values.
- Multiplicative – product from two different scheme secrets can be obtained by multiplying shares of both secrets.

Additionally it supports weighted participation for the users in the scheme. It can be done in an unorthodox way by issuing more of the shares to a single participating entity.

1.10.2 Schemes with complex access structures

Complex ownership models in secret sharing schemes are called access structures. Schemes that use these ownership models use mathematical relations to reduce the size of weighted shares [7, 22, 15]. Multipartite access schemes allow efficient groups of share weights while ensuring validity and perfectness (see chapter 1.10.1). Bipartite[22, 15], Tripartite[7] and Multipartite access schemes represent two, three and many weight partitions respectively[7]. Schemes that use access structures with single weight for all shares are already provided by schemes similar to Shamir's secret sharing scheme (see chapter 1.10.1).

[6] Addition or multiplication.

[7] For instance bipartite access structure allows two kinds of weights to be used at the same time in a single sharing scheme with any share combination for secret recovery.

1.11 Threshold encryption scheme

Threshold encryption schemes are cryptographic techniques to reduce communication between multiple involved peers for decryption. There are numerous such encryption schemes [6] for different cryptosystems. Not all cryptosystems support it.

2 AVDF puzzles

Asymmetric key VDF puzzle is a computational problem that enforces the solver to compute until the solution is found. It's defined as a cryptographic key pair that has an unknown private key but that key exists. It's difficulty level depends on the key's cryptosystem's parameters.

Asymmetric key VDF (AVDF) is constructed as a public key from an asymmetric cryptosystem where public key is generated without a private one. AVDF has two additional functions -- *Encrypt* and *Decrypt*. Both of those functions are asymmetric encryption functions from the underlying cryptosystem.

2.1 AVDF cryptosystem

AVDF cryptosystem is an abstract cryptosystem that allows public key generation (or verification) without reliance on private keys.

By choosing parameters for the cryptosystem we can adjust the length of resulting asymmetric public keys (and change difficulties of puzzles).

In order to make a specific asymmetric cryptosystem behave like a VDF function we have to make sure that:

- At least one of (or leaning towards both[8]) is true:
  - All public keys resemble a private key [5]
  - It's possible to verify that private key exists for a random public key (I.E. it's part of the field [14])
- It's possible to verify that private and public keys match. Verification of the public-private key pair should be a relatively easy task. Most of the cryptosystems generate private key first and then derive the public key from it.
- Private key derivation from public one is a hard task and it's security depends on the parameters of the cryptosystem.

2.2 AVDF puzzle workflow

Let D be valid puzzle difficulty parameters[9], c be a cryptosystem created using parameters D and let R be an abstract random generator.

[8] ECIES doesn't strongly follow this rule. Details can be found in section **??**.

[9] Cryptosystems may expect specially picked numbers (e.g. prime) as construction parameters.

Then VDF workflow (from equations 1.6) would result in:

$$\begin{aligned} public &= Setup(R, c) \\ private &= Eval(c, public) \\ v &= Verify(c, public, private) \end{aligned} \tag{2.1}$$

Where:

- $Setup$ involves generation of a new public key without knowing the private key.
- $Eval$ is a guess stage which eventually yields a matching private key.
- $Verify$ is performed by deriving a public key from the private one and checking for equality.
- Newly acquired boolean parameter v is $true$ when found private key matches the public one.

2.3 AVDF cryptosystem setup

To construct a cryptosystem for AVDF puzzle creation it's needed to know the computation difficulty of $Eval$ and time target. This difficulty then should be converted into cryptosystem's parameters. It's useful to start with a small difficulty and increase until the difficulty matches the resources on hand.

2.4 AVDF puzzle encryption workflow

Let $public$ be a public key in a cryptosystem c which is valid AVDF puzzle, and m a plaintext message. Then encryption and decryption would be performed as follows:

$$m' = Decrypt(Eval(c, public), Encrypt(c, public, m)) \tag{2.2}$$

And m' would be equal to m

2.5 Randomness in $Eval$

$Eval$ may use an encapsulated random generator (not connected to the $Setup$ step). This means that it could add additional randomness to the system in the form of a generated private key.

2.6 Difficulty adjustment

For every cryptosystem type the adjustment of difficulty may be a different kind of process. Generally cryptosystem's difficulty of an already functioning cryptosystem can't be changed. Security level and key length is defined by field power, modulo or other parameters of the cryptosystem [2], [5]. If we want to produce cryptosystems with different security levels (i.e. different public key lengths) we have to construct a new cryptosystem for exact adjusted key length.

3 Cryptosystem suitability for AVDF

Not all public key cryptosystems are suitable for AVDF puzzle construction. The construct expects one additional property over public key cryptosystems — trustless public key setup without a private key.

This section lists only changes in public key derivation methods because the remaining parts of the cryptosystem shouldn't be changed. Changing them would result in unintentional deviation from original cryptosystem designs and that wouldn't be desirable from security point of view. Changes to public key derivation and usage in AVDF scheme may have an impact on overall security of cryptosystems.

3.1 El Gamal asymmetric encryption scheme

El Gamal's cryptosystem parameters can be adjusted to include all possible entries of it's group (See chapter 1.4). This property allows to generate public keys with assurance that private keys will exist every time. It not only allows to use this scheme with key existence but does it in a lossless way – all public keys resemble a private key.

3.2 ECIES asymmetric encryption scheme

Public key in ECIES is a point on an elliptic curve.

It's easy to generate points and verify that they exist on the curve, but point verification is not enough to use them as valid public keys. Trustless public key generation can't be done reliably because ECC doesn't work with whole group of points [14] (in contrast with El Gamal). So it will either force AVDF creation to stick with inefficient curves or result in increased network calculation costs to find (and verify) the desired complexity of generated curves.

3.3 LAC CPA public-key encryption scheme

Algorithm 5 defines a way to generate a key pair for LAC.CPA scheme. Trustless key generation requires adjustment of parameters of this cryptosystem:

1. Trustless public key generation shouldn't involve a private key or $seed$.
2. Error distributions $\Psi_{\sigma}^{n,h}$ may not allow to pick all possible elements of the R_q.

The distributions are constructed in a specific way such that if one would overlay them onto a n dimensional lattice in $n + 1$ dimension space the whole field would be (mostly) uniformly covered. Therefore this means that all (or most) points from R_q are parts of a distribution near a lattice point.

3.3.1 LAC.CPA: public key generation without a private key

If we'd take algorithms 5, 6 and 7 and extract all transformations of the public key and plaintext we'd get the following:

$a = Samp(U(R_q); seed_a) \in R_q$	Initial randomness from $seed_a$.
$b = as + e_{kg}$	Public key generation with error $e_{kg} \xleftarrow{\$} \Psi_\sigma^{n,h}$.
$(r, e_1, e_2) = Samp(\Psi_\sigma^{n,h}, \Psi_\sigma^{n,h}, \Psi_\sigma^{l_v}; seed)$	New randomness during encryption.
$c_1 = ar + e_1 \in R_q$	Ciphertext (part 1)
$c_2 = (br)_{l_v} + e_2 + \lfloor \frac{q}{2} \rceil \cdot \widehat{m} \in \mathbb{Z}_q^{l_v}$	Ciphertext (part 2).
$u = c_1 s \in R_q$	Produced during decryption.
$\widetilde{m} = c_2 - (u)_{l_v} \in Z_q^{l_v}$	Recoverable plaintext. Used in later steps (See alg. 7.).

Let's remove the components of public key to see what part errors play in $\widetilde{m}$:

$$
\begin{aligned}
c_2 &= (br)_{l_v} + e_2 + \lfloor \frac{q}{2} \rceil \cdot \widehat{m} && \text{Ciphertext (part 2).} \\
&= (r(as + e_{kg}))_{l_v} + e_2 + \lfloor \frac{q}{2} \rceil \cdot \widehat{m} && \text{Elimination of } b. \\
u &= c_1 s = s(ar + e_1) \\
\widetilde{m} &= c_2 - (u)_{l_v} \\
&= (r(as + e_{kg}))_{l_v} + e_2 + \lfloor \frac{q}{2} \rceil \cdot \widehat{m} - (u)_{l_v} && \text{Elimination of } c_2. \\
&= (r(as + e_{kg}))_{l_v} + e_2 + \lfloor \frac{q}{2} \rceil \cdot \widehat{m} - (s(ar + e_1))_{l_v} && \text{Elimination of } u. \\
&= (ars + re_{kg})_{l_v} + e_2 + \lfloor \frac{q}{2} \rceil \cdot \widehat{m} - (ars + se_1)_{l_v} && \text{Expansion of nested elements.} \\
&= (re_{kg})_{l_v} + e_2 + \lfloor \frac{q}{2} \rceil \cdot \widehat{m} - (se_1)_{l_v} && \text{Elimination of } ars.
\end{aligned}
\tag{3.1}
$$

The cryptosystem's public key b is composed of components a, s and e of which a and e are random errors taken from errors distributions (see algorithm 5). Equations 3.1 show that expanded version of $\widetilde{m}$ doesn't involve a. One of the interesting things is that all of the coefficients are multiplied by an error sample in some way except of the message — r, e_{kg}, e_2 and e_1 are all additionally produced errors. This means that the algorithm doesn't break if error distribution $\Psi_\sigma^{n,h}$ is chosen together with constant q (see algorithms 6 and 7) is used. And this means that it's possible to find lattice and error distribution pairs that allow any public key to be used in encryption.

Cryptosystem's use of error-correcting codes can adjust the recoverability of the message even further. AVDF produced from LAC.CPA structure would be able to additionally adjust message's recoverability by adjusting parameters of error-correcting code.

4 Naïve AVDF puzzle time-lock chain

One of the simplest variations of time lock can be created by chaining multiple AVDF puzzles together. This would involve one or more computers brute-forcing the keys for the chain to advance.

An implementation for a single CPU can be scaled up by using a PoW mechanism such as in Bitcoin[1]. The mechanism provides means to stop premature advancement of the global state (chain) by forcing all of the participants to solve a hard problem. Once any of the participants solves and shares the solution the chain can advance further [1].

Bitcoin's implementation uses SHA256 hash with a threshold (difficulty) that is used to change the amount of computation to find a problem solution [1]. It is possible to change that problem into something else. For this chapter SHA256 hash will be replaced by an abstract AVDF problem.

To define a simplistic time-lock mechanism and send an encrypted message it is needed to have two execution contexts: Loop and Encryption. Loop context will belong to miners of the chain and Encryption context will belong to the individual user.

4.1 VDF Loop

Loop context is worked on by miners to advance the chain. This phase of the algorithm produces new keys so that data encryption user could use them.

Let R_t be a seeded random generator; $createCryptosystem(r : R_t, d : \mathbb{P})$ be a function to produce an AVDF-compatible cryptosystem; t and t_0 – sequence numbers; $Seed$ – seeding function to produce a new random generator; d – difficulty of a cryptosystem; $\mathbb{P}$ – valid AVDF cryptosystem parameters.

Initialization step:

$$c_{t_0}, R'_0 = createCryptosystem(R_0, d)$$

This mechanism can be used to connect many AVDF puzzles:

$$\begin{aligned} public_t, R'_t &= Setup(c_t, R_t) \\ private_t &= Eval(public_t) \\ R_{t+1} &= Seed(R'_t, private_t) \end{aligned}$$

R_t generation seed has to be immutable i.e. the results have to be verifiable if validating with same input parameters. *Eval* may generate keys using a random generator, but it's not supplied in these equations because the order of keys is not relevant. Produced random generators (R'_0 and R'_t) are advancements of the initial random generator because only the advanced version contains new randomness.

4.2 Locking

In order to make lock ($Encrypt$) and reveal ($Decrypt$) mechanisms work the public key has to be generated first. This means that not only $public_t$ has to be publically disclosed, but all other parameters too, even the randomness generators. The parameters for cryptosystem c_t should be chosen such that it would take some[10] time until $Eval$ finishes it's lookup. Encryption would be performed simply by:

$$ciphertext = Encrypt(c_t, public_t, plaintext)$$

This would mean that message would be undecipherable in the context of the executing machine[11] until it calculates the private key and reveals it.

[10] Strength of $public_t$ is adjustable by choosing cryptosystem's complexity

[11] $Eval$ step is paralellizable and fixed in complexity so attacker could use more resources and find the private key in advance.

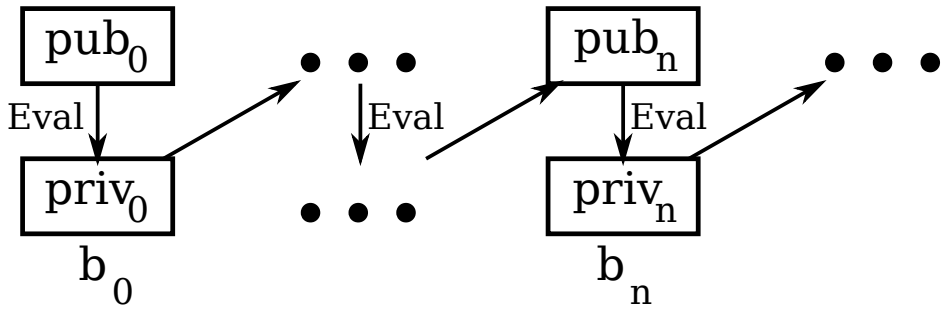


Figure 1. Diagram of naïve locking mechanism.

Figure 1 shows the relations between chain-connected AVDF puzzles. Parameter b_n shows a single PoW block with a sequence number n. Public and private asymmetric key sequence is presented as pub_n and $priv_n$ and their parameter n shows which block the keypair belongs to. Key pub_n can't be known before executing $Eval$ because derivation of pub_n uses $priv_{n-1}$ as a seed.

4.3 Difficulty adjustment

Bitcoin network adjusts it's difficulty using SMA of previous block times [1]. As there is only one key per block the difficulty has to be adjusted by changing the complexity of the underlying cryptosystem (AVDF).

4.4 Feasibility and shortcomings

The scheme 4.1 is a valid Proof of Work protocol but it has at least three problems:

- Difference in difficulty target size.
- Proof of Work and block finality [1].
- Only one public key exists per single block.

Difference in difficulty target size. There is a difference between brute-forcing of a thresholded SHA256 hash and of an asymmetric key. There is only one key as a target for an asymmetric key and threshold that is used in Bitcoin allows brute-forcing many hashes at the same time[12]. At first this may look trivial but it becomes tricky when chain returns to a similar kind of difficulty. When previously visited difficulty is reached again the system has a possibility to generate the same public key for brute-forcing. This problem can be solved by a history search or ordering of the public keys. The existence of previously mined key would reveal the message of the sender.

The problem won't occur with miner's block rewards because the block's AVDF puzzle's cryptosystem and public key randomness should be seeded from previous blocks.

PoW block finality. In the event of a temporary fork some calculations are lost. It is considered that weaker chain never happened [1]. Users of the weaker fork will not get their decryption key because that key won't be saved onto the main chain. This means that PoW mechanism doesn't ensure that all users will be able to decrypt their data. It's known as 51% attack against PoW network [1] and it is one of the weak points of PoW mechanism.

[12]Thresholded brute-force is the way that Bitcoin adjusts it's difficulty [1]

Key granularity It's a concern about usability of the system. At any given moment in time there exists only a single key to be discovered by PoW network. This means that user of the system will only be able to send messages to a single moment in time. That moment will come after the network will publish the private key.

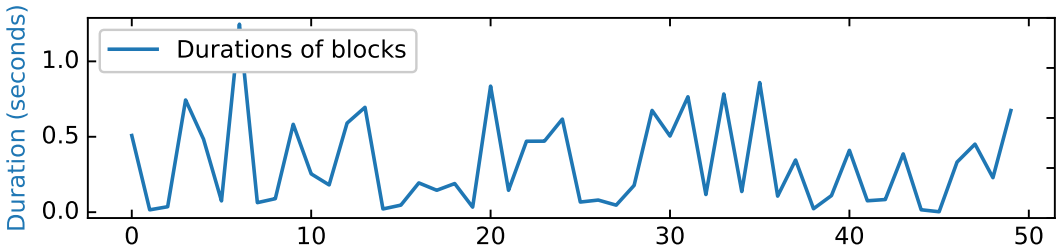


Figure 2. Single block duration in each epoch

Figure 2 presents a graph of 50 mined block times. The duration deviation from the mean is larger than the mean itself. This means that single key target doesn't smooth out the differences enough between guesses.

5 Chain of multiple key batches

To improve usability of naïve time-lock mechanism from chapter 4 more keys have to be generated to be brute-forced at the same time. Batching of more than one key would allow users to choose decryption times for their messages using threshold encryption schemes (read more in chapter 6).

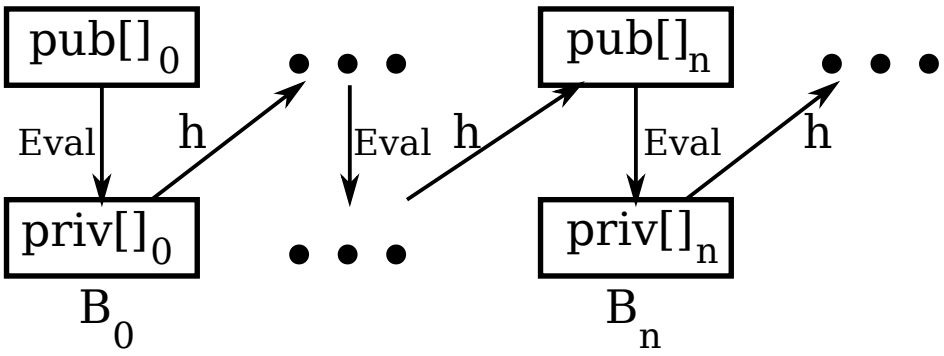


Figure 3. Chained multiple-keypair batches.

Figure 3 depicts a scheme of AVDF batches that extends the key list when private keys are revealed. It's an extension of design in Figure 1. The scheme shows use of multiple public keys per multiple-block epoch B_n. $pub[]$ and $priv[]$ are public and private asymmetric key lists. Arrows $Eval$ is a step of multiple AVDF puzzles and arrows h show randomness seeding from previous multi-block epoch outcomes to the next generation of AVDFs. They additionally show finalization of a private key calculation – advancement can be made only after the end of this calculation.

It's not enough to epoch-batch the keys in the system. Batches have to be stored and their contents have to be ordered in some way.

This can be achieved in several ways:

- External finalization of key batches
- AVDF for PoW block finalization

5.1 External finalization of key batches

Choice of external ledger for key snapshot validation would allow use of smart contracts of non-PoW blockchains and even more efficient (but arguably safer in worst-case scenarios) technologies like centralized storage. It would be flexible to do it this way but the design has at least these flaws:

- External storage system.
- Blockchain with two different kinds of PoW.

5.1.1 External storage system

External storage system[13] can be used for key sequence validation and ordering. As it doesn't force to use a blockchain it may be a good choice to save resources. Stored data could be any sequence of keys and the keys would be able to be cleaned up after some amount of time. This kind of activity would be governed by policy of the external system's supervising party. The design would improve partition-tolerance (compared to PoW blockchain)[8] as it would allow cheap and flexible way to synchronize AVDF puzzle outcomes.

The risk is that the owner of the external validation system can rollback or change previous work of the miners that solve AVDF puzzles and save outcomes on the ledger. It also means that puzzles can be created from any keys that are published by the system provider – even maliciously created ones.

Additional risk may involve a decrease of miner or user anonimity as external system would require some kind of registration, policy or fee.

5.1.2 Blockchain with two different kinds of PoW

Two different kinds of PoW is a working design choice but wouldn't be too beneficial in the long term. It would function the same if an external smart contract would be used to store the mined keys and external system would validate their existence. The only difference is that miners would have to balance their resources between AVDF mining and regular block-validation mining. This would be confusing as reward mechanism would be triggered at different rates for each miner group.

Comparison with Bitcoin PoW network Change in key production mechanism changes the semantics of the block. Validation mechanism from chapter 4 included a private key calculation. In this case the AVDF puzzle solution is regarded more as a transaction than a validation hash digest. It is a message of proof that calculation happened, but it doesn't finalize the block. In Bitcoin these transactions would exist in mempool [1] until they are incorporated into a block.

5.2 AVDF for PoW block finalization

In order to be used as a PoW chain and avoid reliance on external systems for key batch lifecycle a change of PoW validation problem is needed. To use AVDF as PoW block's computational problem[14] it's randomness has to be seedable and it should support deterministic verification.

[13] Not related to AVDF solving.

[14] Bitcoin's implementation uses SHA256 as it's computational [1] problem.

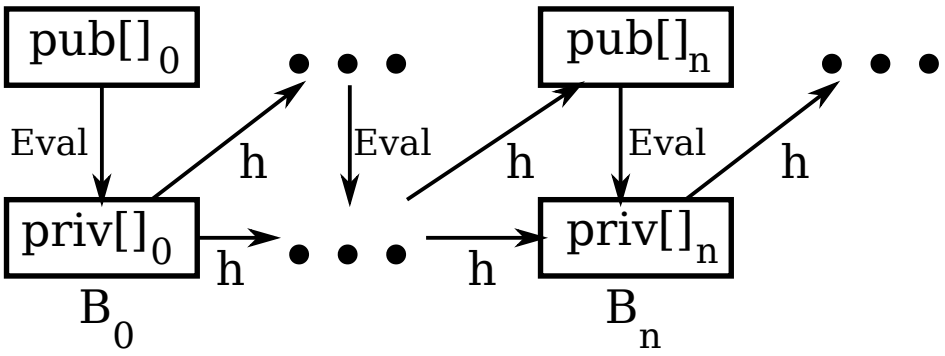


Figure 4. Chained verifiable multiple-keypair batches

Figure 4 contains one additional arrow h compared to Fig. 3. It is an additional seed that prevents the system to change it's previously calculated private keys. Private keys of cryptosystems are generally shorter than their private keys so collision is inevitable[15].

5.2.1 Mining incentivization

Bitcoin is a value transaction system powered by PoW[1]. To incentivize mining of the AVDF puzzles for data time-locks the system should reuse this model.

In Bitcoin's case miners prepend an UTXO to block's transaction list that rewards them. If the mined block is valid [1] then the reward is carried through.

To use same kind of mechanism it is needed to have similar wallets to Bitcoin's ones. This way after a miner produces a successive block she will be able to get rewards for the work.

5.2.2 Security and external keys

External key cracking attack would decrease security of the whole system for the time-lock users. To prevent this kind of attack the awarding mechanism should involve hash checks of key origin. Any non-chain related private keys should not be awarded even though they would decrypt correctly. I.e. blocks of data can only be accepted if they correctly contain hashes of previous block data. This way an external attacker would be forced to either attack the system without incentivization or commit the resources to the system.

5.2.3 Usability for the time-lock users

In order to use this system on a broad scale an SPV [1] client implementation is needed. This way users would be able to run light clients on weaker devices (Such as smartphones or even IoT devices). SPV node would trust a list of chosen (or privately hosted) nodes to get the newest unmined private keys.

5.2.4 Tests

Duration difference between blocks It would be optimal to maintain similar duration in all blocks without adding any extra computing power and without changing any difficulty. To test that the blocks don't change their difficulty randomly ANOVA one-way test was performed for different data sets.

[15]Two private keys can end up having the same public key.

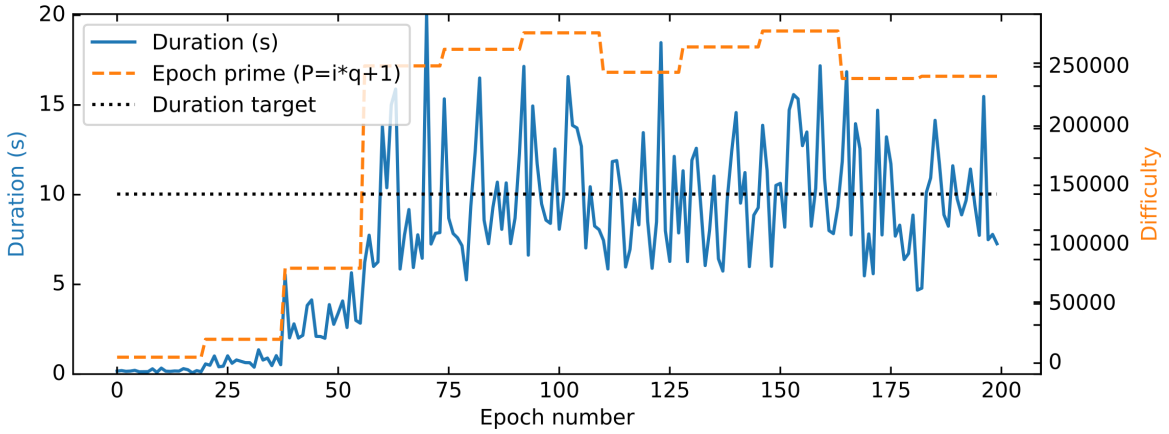


Figure 5. Multiple keys as targets. All keys are mandatory to advance. Block prime is a prime number used in El Gamal cryptosystem. 200 block epochs with 60 keys in each of them.

The distribution of the results is similar to the graph 2.

All data sets contain 5 blocks of El Gamal cryptosystem benchmarks with different key counts in each of them. P value is the one used in public key derivation.

ANOVA one-way results:

Prime P	Keys in a block	ANOVA F value	ANOVA p value
60037	4	~ 1.6	0.23
60037	9	~ 2.738	$4.19 * 10^{-2}$
60037	18	~ 1.22	0.31
60037	36	~ 1.8	$1.31 * 10^{-1}$
60037	360	~ 3.9	$3.47 * 10^{-3}$
154043	36000	~ 201.18	$1.76 * 10^{-172}$

The tests show that not only the blocks are not similar, but as the keys count and P increases the difference may increase (because p value nears to zero). This analysis shows that current approach for key targeting is not good enough if perfect time division intervals are desired.

Duration dependence on block difficulty Figure 5 shows difficulty changes of one hundred blocks containing 360 keys each (Every data point is time of 360 keys). The difficulty is changed every 20 blocks. The prime number was derived first and all of the public keys were generated from it. Then brute force mechanism was used to produce durations.

There is some visible dependence between duration and block prime (difficulty) but the signal is still mostly chaotic.

5.3 Block validation public key set

Bitcoin uses a difficulty value for it's block hash calculation. It is a number that leads to calculate hash that has many leading zeros. The remaining part of the hash is not restricted. So Bitcoin network may accept many hashes, but only one[16] would be added to a block as validation proof [1].

[16]In the occurrence of a temporary fork several keys can coexist, but after the fork resolution the redundant keys get destroyed

This is very similar to the list of public keys in section 5. It is similar in a way that the key list will exist.

The key list can be used to finalize a single block and if difficulty doesn't change then it can be reused again.

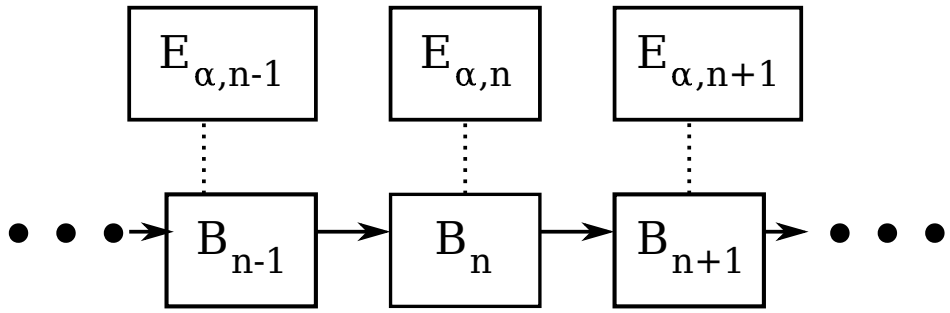


Figure 6. Block key target progression.

Figure 6 shows a way to share and reuse unbruteforced keys between different block mining sessions. α is the difficulty of the block and n is block number. Each successive block epoch B_n is assigned a target key list $E_{\alpha,n}$. This way miners can know which public keys are known and which are useful to pick for next round of work.

$E_{\alpha,n+1}$ is produced by removing the key that was used for validation of a previous block from it's target key pool and adding a new one:

$$\begin{aligned} \xi &\in E_{\alpha,n} \\ \xi' &\notin E_{\alpha,n} \\ E_{\alpha,n+1} &\leftarrow \forall x(E_{\alpha,n} \wedge x \neq \xi) \cup \{\xi'\} \end{aligned}$$

Where ξ' is previously unused public key matching difficulty α.

This scheme is sensitive to PoW chain forks. In the event of a PoW fork the key ξ would be a different entry of $E_{\alpha,n}$.

5.3.1 Impact on SPV clients

Public key storage would pose a challenge because the network will need to calculate currently accepted key set. This would mean that light nodes (SPV [1]) won't be able to be as light as they could be. Most probably the keys would be stored in separate blocks and included into a block if needed.

5.3.2 Comparison to Bitcoin PoW network

This scheme will still work in PoW mode, but it's possible that block time will be changed. Also the keys will have to be part of the block (current target keys should be stored in older blocks) so that it would be possible to calculate.

5.3.3 Security

Node has to guess a private key from a public key list to perform block validation. Network's security will depend on very weak keys. This will not only have the bad effects when forking, but also it will pose a risk for encryption users. Security of encrypted messages will be lowered:

- Chain forks may render encrypted data undecipherable by destroying targeted used keys.
- Forks will also reveal multiple keys at once by merging keys from non-included blocks.
- Individual keys may be cracked with lesser effort because they will be weak [17]

5.3.4 Tests

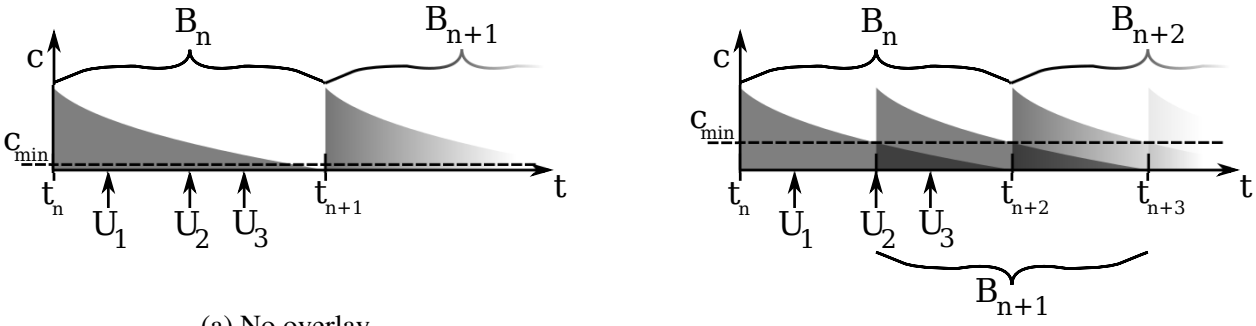
Similar ANOVA one-way analysis was performed to a different set of key blocks. This time blocks were generated with more targeted keys but it was allowed to mine less of them to proceed:

Prime P	Keys in a block	Accept threshold	ANOVA F value	ANOVA p value
60037	4000	36	$\sim 4.76 * 10^{-2}$	0.9957
60037	400	36	~ 0.91	0.4576
60037	40	36	~ 2	$9.6547 * 10^{-2}$

This time the output contains some schemes where blocks are not significantly different statistically ($p > 0.05$). This strategy may work to control the randomness in a block, but it will add additional block size and other significant computations if used in PoW[18].

5.4 Avoiding the paradox of bus arrival

Previously presented chain design has a flaw that prevents it from being useful for continuous use. To ensure usability system is supposed to continuously offer a set of public keys for user data encryption. Time-lock has to be achievable at any time.



(a) No overlay

(b) With block overlay

Figure 7. Overlaying of blocks to provide new key availability for users.

To solve this problem a macro-level design choice has to be made. In order to allow key availability at any time miners shouldn't completely exhaust every target key batch but preserve at least some keys for the future.

Figure 7b shows how design in Fig. 7a can be improved by overlaying blocks onto each other. B_n are consecutive block epochs. Parameter t_n shows their respective mining start times. Gray curve-shaped areas denote public keys that are left to brute-force in each key target batch. They are shown as continuous but in real-world case they would be discrete. c axis shows publically available remaining keys as total remaining multi-block difficulty. c_{min} and it's dashed line shows

[17] Bitcoin's block hash #580204 contains 19 leading zeros. This may grant at most 76 bits of security.

[18] Aditionally one may use a bloom filter to reduce the size of the key set.

the lowest possible key set size (remaining minimal difficulty) when user joins and decides to encrypt a message.

U_1, U_2, U_3 are examples of three users and their message encryption times. User key requests can occur at any moment so the system has to ensure key existence.

Using the design of Figure 7a users who join the network at precise time moment of block start are able to choose between all possible keys. But users who decide to join between block change periods have less or none available keys to choose from.

In Figure 7b darker curved areas that overlap between two consecutive blocks show key sets that match between two target key sets. This scheme of block overlay reuses the keys from previous block.

Two consecutive-block key reuse strategies exist:

- Implicit target key set reuse
- Blocking target key set reuse

5.4.1 Implicit target key set reuse

Implicitly sharing keys between consecutive blocks is easier from implementation point of view. Keys are stored as a single thresholded pool that is given for a brute-force algorithm. It has a caveat that impacts time-lock users though – some keys can remain unbruteforced for a very long time or not brute-forced at all. This would result in a time-lock that has a chance of not being predictably recoverable. Some time-lock messages would end up completely unrecoverable.

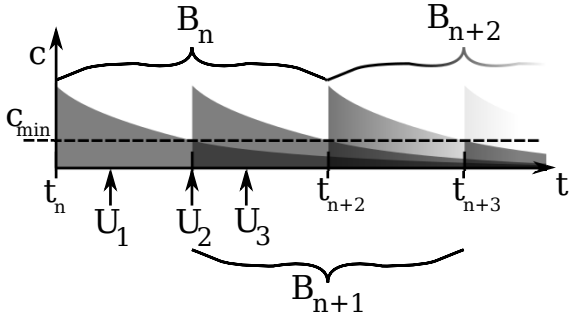


Figure 8. Implicit overlay key trail.

Figure 8 shows a trail of overlap between multiple target key sets. The area near the t axis between t_{n+2} and t_{n+3} involves leakage of keys from epoch B_n. Keys can leak into a very far future. This would be the result of continuously adding new keys to the key batches without completely finishing the previous ones.

5.4.2 Blocking target key set reuse

Figure 7b implies that keys become mined in each consecutive epoch. This way the chain can advance in a safe manner.

From implementation perspective this is achievable by enforcing the miners to mine specific keys at the right time. Keys should be divided into two groups: mandatory and threshold-mandatory. Miners would be expected to mine on a batch of keys as in chapter 5.4.1 but acceptance criteria would depend on keys which are mined. Thresholded-mandatory group wouldn't need to be fully mined whereas mandatory group would need to be fully mined. No new keys should be issued before finishing the mandatory group even if threshold-mandatory group is exhausted.

5.5 Difficulty adjustment

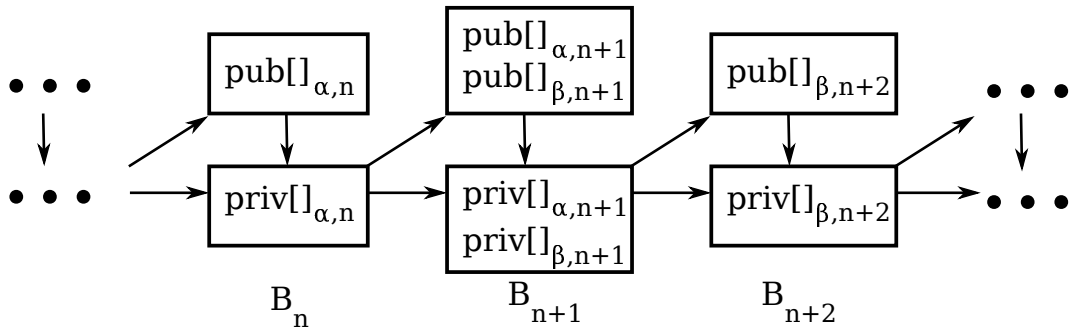


Figure 9. Adjustment of PoW difficulty from α to β through multi-block epochs.

Every PoW blockchain adjusts it's difficulty in some way. A difficulty-adjustable version of a protocol was presented in chapter 4. It has ability to adjust it's difficulty at every PoW block.

Usability improvements that were defined previously are not compatible with difficulty adjustment at every block[19]. As block-sets share their brute-force target keys between their neighbors user messages won't be recoverable after a sudden change in difficulty of blockchain. The design has to be adjusted to allow the difficulty changes.

Figure 9 presents three public key batches that are used to mine multiple block epochs each (B_n, B_{n+1}, B_{n+2}). The epochs contain same structure as it's shown in figure 4. The middle batch allows two kinds of key difficulties blend into a single key set. This allows adjustment of the difficulty on a macro-block level.

5.5.1 Key set sizes during transition

Target key set B_{n+1} is composed of two different public key sets: $pub_{\alpha,n+1}$ and $pub_{\beta,n+1}$. To ensure the properties of chapter 5.4.2 all keys targeted in epoch B_n have to be brute-forced in epoch B_{n+1}. It is achievable by calculating total needed work power c (See figure 7b) and allocating respective amount of keys of difficulty β.

5.5.2 Multiple difficulty adjustments in a row

It's possible to adjust difficulty multiple times in a row. It can be done by copying the structure of figure 9 of epoch B_{n+1} multiple times in a row. The work power calculation c is similar to the single-transition mode.

5.5.3 Time-lock threshold encryption during change of difficulty

During a change of the mining difficulty two cryptosystems are presented on the system at the same time. Change of the difficulty is a complex action from cryptographical point of view. There is no easy way to know which cryptosystems are compatible between each other even if they are based on the same concept. It can also mean that they can't be compatible at all. This means that threshold scheme for two different cryptosystems may not exist.

Previous concepts allow guaranteed decryption and choice of the decryption time. Difficulty changes decrease mining certainty because miners have to leverage two key target lists composed

[19]PoW allows it. Bitcoin adjusts difficulty every 2016 blocks [1].

of two different cryptosystems. To retain previously defined properties without the risk of changing the decryption time user of the system has to construct encrypted time-lock package is a specific way. When two cryptosystems are present at the same time the user is forced to combine two threshold encryption schemes or use a secret sharing scheme with bipartite access structure. It's further discussed in chapter 6.2.

6 Message decryption time targeting

To freeze a message using combination of AVDF and PoW frameworks user has to perform a threshold encryption using the provided keys. User has to be able to choose his threshold encryption/sharing scheme to reduce size of time-frozen messages.

For most of the block difficulty epochs[20] it may not be possible to use a single scheme that could be similar to Shamir's secret sharing scheme with AVDF encryption on top. This chapter presents multiple options of thresholded secret sharing for different access structure needs.

6.1 Time targeting for homogeneous difficulty epoch

Homogeneous difficulty epoch is the simplest case where only one cryptosystem is present on the system. User messages can be split using any secret sharing scheme without any consideration about weights[6].

6.2 Time targeting during difficulty-change epoch

Difficulty change epoch contains two different incompatible cryptosystems (see fig. 9). To allow predictable decryption of the value it has to be split into parts and shared between two different-weight threshold locks. It can be achieved in two ways:

- Threshold encryption with code-based access structure
- Threshold secret sharing with bipartite access structure.

6.2.1 Code-based bipartite access structure using threshold encryption

Code-based approach combines multiple threshold encrypted[6] versions of the same message into one large payload. It can be achieved by taking each group of keys and producing threshold-encrypted message for each valid sub-combination. It is not a straightforward task because both epoch key pools contain keys of different difficulties: α and β.

$$T \subseteq \left\{ (i_\alpha, i_\beta) \middle| \begin{array}{c} i_\alpha \in \{0, ..., l_\alpha\}, \\ i_\beta \in \{0, ..., l_\beta\}, \\ w_\alpha i_\alpha + w_\beta i_\beta \geqslant w_d \end{array} \right\} \tag{6.1}$$

Figure 10. Bipartite threshold access structure for difficulty-adjusted epoch.

[20]For instance Bitcoin's difficulty is changed almost every time it's possible to do it: https://bitinfocharts.com/comparison/bitcoin-difficulty.html

Equation 6.1 shows a set of threshold locks T for encryption in difficulty adjustment epoch. Each threshold lock will be created from returned indexes. For instance $(5, 2)$ would allow secret retrieval with 5 keys from first set and with any 2 of the other key set.

It is not an optimized version of lock-choosing algorithm. This equation chooses all thresholds that are stronger or equal with w_d so it includes interleaving locks too. Result-optimized code snippet version is listed in appendix B.

The set is composed using an iterative sum over two integer sequences where l_α and l_β are pool sizes of both epochs where indexes are used to produce combinations of the keys. These pool sizes may be different from each-other if difficulty algorithm decides it ($l_\alpha \simeq l_\beta$). Parameter w_d is a total weight needed for decryption using threshold-scheme. w_α and w_β are weights of a single pool key each.

Example: If epochs share 50% of their keys and target size of each epoch is 10 then parameter $c_{min} = 10 * 0.5 = 5$ (see c_{min} from figure 7b) and lengths $l_\alpha = l_\beta = c_{min} = 5$. Let's say difficulty algorithm decided to change the difficulty from $\alpha = 10$ to $\beta = 11$. Then total difficulty of the whole transition epoch would become:

$$w_B = c_{min}(\alpha + \beta) = 5 * (10 + 11) = 105.$$

Let's say user wants to ensure their time-lock message's decryptability after 40% of epoch's elapsed time. Then decryption threshold weight would be equal to:

$$w_d = w_B * 40\% = 105 * 0.4 = 42.$$

User needs to perform multiple threshold encryption locks. Threshold encryption lock count is calculable by using equation 6.1. Calculation code snippet is presented in appendix B.

The code snippet from appendix B listed 6 possible encryption schemes for this given example:

```
([0  4]  [1  3]  [2  2]  [3  2]  [4  1]  [5  0])
```

Figure 11. Sample multi-lock tuple list of $[l_\alpha\ l_\beta]$ for difficulty change epoch time-lock threshold lock creation.

At this stage real-world application would construct multiple threshold-encrypted versions of the same message using thresholds from figure 11.

Threshold lock composition is performed by nesting two locks inside of each-other. It's done by creating a first lock and using it as an input for the second one. Nesting example is shown in figure 12.

Every item from this list is a blueprint for a threshold signature scheme of two cryptosystems. See appendix

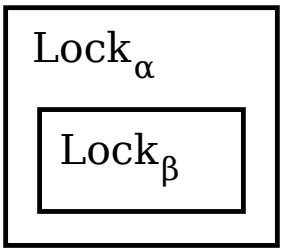


Figure 12. Lock composition.

6.2.2 Secret sharing with bipartite access structure

Bipartite sharing scheme similar to 6.2.1 can also be implemented by using a secret sharing scheme with this sharing property[22, 15]. After constructing bipartite scheme AVDF encryption has to be matched with each share.

7 Implementation

The code part of the project contains a proof of concept that demonstrates creation of time-lock chain. It was written in Go programming language [27] which is a systems programming language. The language offers execution speed such that when it's tuned correctly[21] it can be comparable to C code.

Application is a command-line tool that performs brute-force calculations and finds the keys if parameters do not overwhelm the machine that it's being ran on. The calculations are not offloaded onto a graphics card and rely solely on processor's capabilities. It uses all available resources for it's computations. If maximum block count parameter is provided or operation is in single-block mode the data file with chain data is produced.

Appendix C contains application's development timeline. Appendices D and E contain code-base size and dependency list.

7.1 Operation modes

The application doesn't implement all of the complex schemes defined in this thesis. The calculations can be performed in several ways:

- Single block mode works on a single pool of keys/hashes without a chain.
- Adaptive mode emulates chain behavior with asymmetric keys or hashes used for block validation.

7.1.1 Single-block mode

Single-block mode performs a brute-force calculation for a single targeted set of keys to measure brute-force time for a single block with different parameters. The data is then serialized into a file.

7.1.2 Adaptive chain-based mode

Adaptive chain-based mode picks a PoW target difficulty according to previous chain timing data or command-line parameter (genesis difficulty). This difficulty is then used for it's next brute-force operation. If the block count parameter is set the chain is stopped after the specific block and serialized into a file.

In a Bitcoin-like PoW difficulty is a number that filters hashes for validity. For time-lock case it has to use asymmetric public keys (AVDF puzzles). As asymmetric public keys can be regarded to as numbers they can be ordered as well. The application uses a threshold value for the keys to match them and doesn't generate them in advance. It correctly[22] implements AVDF Loop workflow (defined in chapter 2.2) for El Gamal's cryptosystem. SHA256 validation mechanism is also implemented and can be chosen using a command-line argument. Choice of SHA256 option doesn't change any other options and block production mechanism.

Blocks in the code are similar to block epochs in this document. The next block epoch is always present and is generated from previous ones.

The application doesn't implement complex peer-to-peer binary communication protocols. Application's chain environment doesn't emulate real-world PoW behavior. It can't cooperate

[21] Parallelism is easier to implement; Garbage collector is optional.

[22] Generates public keys without knowing private keys.

with other miners. It emulates only best-case execution without any wastefully targeted computation. Forking, network partitioning, network failures and other interferences/attacks are also not taken into account.

7.2 Difficulty retargeting mechanisms

Depending on the implementation of the underlying PoW validation primitive the system may need a different kind of difficulty adjustment curve. Hash-based validation and AVDFs use different mechanics to change the difficulty of underlying PoW problems.

SHA256 is a hash-based scheme and it allows 2^{256} of combinations. While AVDF-based cryptosystem has to be adjusted and it's size is changed every time a harder problem is required. The scheme similar to as in Hash-based adjustment could also be used for AVDFs but then the user would lose practicality of document encryption using generated AVDF public keys.

The application contains implementations of two difficulty adjustment mechanisms. AVDFs require difficulty that gets increased when harder problem is needed while Hash-based verification requires the difficulty to decrease.

7.2.1 Difficulty retarget jump limit

Implementations of Bitcoin nodes[20, 21] limit radical jumps of the difficulty. This is required to avoid dramatic changes in difficulty as sometimes chain can get overwhelmed by influx of mining power.

7.2.2 ElGamal's scheme tests

ElGamal's scheme has shown that it's not only possible to adjust it to match performance of SHA256 but also use it for validation PoW block. A benchmark that compared these two algorithms was performed on a single Raspberry Pi 4B that is equipped with Quad core 64-bit 1.5GHz ARM processor. It executed 20 runs of each blockchain where each run is similar to one depicted in figure 5.

Graphs from appendices A.1 and A.2 represent blockchain execution on a single CPU that was developed for this thesis. Enlarged versions of equilibrium that is visible in the last 60 blocks are included in appendices A.3 and A.4. The graph of SHA256 is meant to represent current state of Bitcoin and similar PoW networks. Whereas graph of ElGamal represents AVDF-based PoW network. Difficulty adjustment step is capped at 4 because sometimes the initial jump is too high. High jump can result in unrealistic difficulty targets. Bitcoin also uses this value and it's capped at 4[20, 21]. The long tail of SHA256's difficulty is produced because it starts from smaller value relatively to ElGamal. ElGamal's cryptosystem starts at difficulty $0.5 * 10^4$ and stops increasing at about $1.5 * 10^5$ whereas SHA256 is initialized with $2^{256} - 0.5 * 10^4$ and stops decreasing[23] at about $4.4 * 10^{74}$

The graphs show that ElGamal's scheme is more volatile than SHA256. There are many instances where block target time 10 seconds that is marked by black dotted line is surpassed and sometimes even values near 25 seconds are reached. It may be a problem of optimization or it could be a problem with the cryptosystem itself. It could be the consequence of cryptography type and may not be fixable without trying other cryptosystems.

[23] In SHA256 PoW difficulty gets harder for lower values[1].

Haar wavelet transformation was also performed before computing SMA of block difficulties. The wavelets were averaged at fourth level of the signal decomposition. It's presented in appendix A.5 but it doesn't show any changes between results of appendix A.1.

8 Performance

The presented version of PoW protocol can be assessed for performance in two ways:

- Miner performance
- Key brute-force performance
- User time-lock encryption performance
- Message size.

8.1 Miner performance

Mining performance completely depends on the PoW mechanism[1]. PoW difficulty is adjusted so that mining wouldn't continue without slowdown or very large front-running won't occur[21, 20].

8.2 Key brute-force performance

Miners can use any means of parallelization to find keys for given key epoch. They are not constrained with not using custom mining implementations on GPUs[24] or ASICs[25].

8.3 User time-lock encryption performance

The design of PoW described in chapter 5 provides means to publicly obtain public keys. Every public key appears publically available after miners mine it and is used for validation of each blockchain block. As a batch of unsolved public keys are available and freely obtainable the users can use them without any permissions or requests. Nothing is persisted in context of the presented design.

8.4 Message size

The only limiting factor performance-wise is encryption of each time-lock message. Chapter 6 defines a way to encrypt the messages but the final encryption size and structure is chosen individually for every message. Complex time targets can result in very long messages (see sample code result lengths of appendix B) and it's evident that time locking should be only used to encapsulate the keys that will be used for decryption. Once the time-lock is unlocked the in-advance chosen strong key would provide the data. This means that two parties can exchange any number of messages without any restrictions and the only limit could become the size of the time-lock encryption message.

[24] Graphical processing unit.
[25] Application-specific integrated circuit.

9 Security

System defined in chapter 5 is based on Bitcoin-like PoW. This means that many security concepts are inherited from Bitcoin:

- It's hard to find block validation hash[1]
- Some computation gets ignored[1] because forking preserves only the block that was produced by majority
- Others.

9.1 Block hashes as public keys

Difficulty-chosen SHA256 value that is used in Bitcoin's block validation is hard to find. This thesis presents this same concept which is transformed to be used with asymmetric cryptosystems. It's still hard to find the private key when previous data is the input into AVDF's random generator. Eventually difficulty parameter of the presented chain will reach a value that is close to equilibrium and that would result in security which is currently provided by Bitcoin.

It's a wishful thinking that some system can achieve hash levels of Bitcoin but it may be possible with a usecase that works and is useful.

Security depends on the token-reward model incentivizes mining of the coin instead of brute-forcing the keys without providing them back to the system.

9.2 External brute-force attack

One of Bitcoin's security assumptions is that no external actor can combine more computing power than the shared mining of all miners[1]İt's called 51% attack and the system defined in this thesis is vulnerable to it.

Bitcoin guards itself in several ways. One of them is snow-layer-based block production.

Bitcoin blocks are chained in such a way that every block verification depends on it's previous one. Current block may be a victim of a 51% fork but it's immutability gets increased with every consecutive block that depends on it and it's hard to cheat and front-mine all the time.

9.3 Quantum computing based attack

Introduction of a cryptosystem instead of hashing algorithm for block validation decreases security of block validation depending on the choice of cryptosystem.

ElGamal's cryptosystem (see chapter 1.3) was used to test feasibility of difficulty adjustment and on-the-spot AVDF derivation. It is malleable which means that it's ciphertexts can be tampered with. It's based on discrete logarithm problem which is known to be attackable by quantum computers. So although being convenient for practical tests of this thesis it's not secure to use in real-world case.

Second scheme that was considered was ECIES which is the current standard for secure communication. This scheme is based on Elliptic curve discrete logarithm problem so it's also attackable by quantum computer.

A different kind of unknown cryptosystem was presented to be used instead of ElGamal. LAC.CPA is a Lattice-based cryptosystem that could be secure enough to prevent quantum-computer

cracking. It's a fresh cryptosystem so it may not be extensively secure[26] in on itself but it's a good sign that lattices can be adjusted for this task. It's based on a hard lattice problem – learning with errors. LWE is not NP-hard – it's based on a modified version of problem that is comparable to NP-hard so it's good enough.

10 Conclusions

This thesis has described a trustless alternative to send messages into a near future based on PoW. It presented multiple changes that ensure that the chain is lively and users can access the keys for production of time-locks.

Conclusions:

- Chapter 5 described an approach for short-term[27] time-locking. It was done by complementing the design of the PoW with additional abstractions such as block epoch. This looks as a viable strategy to construct PoW-based time-lock scheme.
- Hardness of block validation cryptosystem can be adjusted on the fly depending on chain's difficulty. Tests using ElGamal's cryptosystem can adjust it's public key length depending on the needed difficulty (see chapter 5).
- Some currently widely used public key cryptosystems are vulnerable to quantum computers but some newly proposed ones can be used for this system instead. LWE-based cryptosystem LAC.CPA (see chapter 3.3) could theoretically be used to prevent quantum-based key cracking.
- Elliptic curve cryptography is not ideal for use in trustless key derivation. It doesn't ensure existence of a private key for all points of an elliptic curve which means that decryption is not guaranteed (more in chapter 3.2).

10.1 Future work

This thesis provides several abstractions and theories but it's not enough to build a well functioning and unbeatable system. To be sure that system is usable at least these points have to be taken into account:

- Explore ElGamal's difficulty retargeting mechanism by changing constraints on cryptosystem's prime number.
- Investigate LAC.CPA and other related lattice cryptosystems for security and suitability.
- Investigate the possibility to use elliptic curves. Lossful variant of a cryptosystem can still allow decryption if time-lock is produced with future keys.
- Investigate the volatility reduction of the mechanism. Implementation of ElGamal's scheme increases volatility (see section 7.2.2) compared to SHA256 which is currently used by many blockchains.

[26] Or the design of the trustless key creation may be unsecure.

[27] For instance with 2 weeks of block difficulty retarget interval (setting from Bitcoin) from 1 to 2 weeks of lock time can be achieved. See figure 7b.

10.1.1 Explore ElGamal's prime number constraints

Performance graphs of ElGamal's cryptosystem are more volatile than it's SHA256 graph (see chapter 7.2.2). It would be nice to explore how the constraints on prime number impact the volatility of the system and up to which extent this can be tweaked.

10.1.2 Explore other cryptosystems

El Gamal's cryptosystem works for non-production use, but it has vulnerabilities to emerging quantum technology and doesn't pass indistinguishibility tests.

11 Recommendations

With the presented locking mechanism every time lock will end up having a different size. Users may need to look into efficiency during message encryption. Some schemes may offer smaller message sizes for decryption so many different encryption schemes should be allowed to be chosen at any moment (see chapter 6).

11.0.1 Targeted key duplication prevention by storage

Locks that use previously known keys will be revealed instantly because someone will have them. The list of already used keys (or threshold) should be hosted by the algorithm.

Also cryptosystems shouldn't be used if they became exhausted in the past. Cryptosystems will eventually get exhausted because public keys will be persisted in the history. This is not a problem for PoW but it's a problem for time-locks as locking won't be useful if all of the keys will be available. It's inevitable that in the beginning the chain won't have enough hashing power to provide useful time locks.

Hash-based systems can't have this problem because hashes are not used anywhere else except for the system.

Duplicates don't need to be taken into account only if:

- Private keys somehow won't be used anywhere else – in this case there is no purpose to use asymmetric cryptography.
- History would be deleted – impossible, Internet doesn't forget anything.

References

[1] A.M. Antonopoulos. *Mastering Bitcoin: Unlocking Digital Cryptocurrencies*. O'Reilly Media, 2014.

[2] Christoforus Juan Benvenuto. Galois Field in Cryptography, 2019.

[3] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. Cryptology ePrint Archive, Report 2018/601, 2018. https://eprint.iacr.org/2018/601.

[4] Gwern Branwen. Time-lock encryption, 2019. https://www.gwern.net/Self-decrypting-files.

[5] Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Proceedings of CRYPTO 84 on Advances in Cryptology*, pages 10--18, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[6] Levent Ertaul and Weimin Lu. ECC Based Threshold Cryptography for Secure Data Forwarding and Secure Key Exchange in MANET (I). In Raouf Boutaba, Kevin Almeroth, Ramon Puigjaner, Sherman Shen, and James P. Black, editors, *NETWORKING 2005. Networking Technologies, Services, and Protocols; Performance of Computer and Communication Networks; Mobile and Wireless Communications Systems*, pages 102--113, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[7] Oriol Farràs, Jaume Martí-Farré, and Carles Padró. Ideal multipartite secret sharing schemes. In Moni Naor, editor, *Advances in Cryptology - EUROCRYPT 2007*, pages 448--465, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[8] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51--59, June 2002.

[9] Darrel Hankerson and Alfred Menezes. *Elliptic Curve Cryptography*, pages 397--397. Springer US, Boston, MA, 2011.

[10] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. Ntru: A ring-based public key cryptosystem. In *Proceedings of the Third International Symposium on Algorithmic Number Theory*, ANTS-III, pages 267--288, Berlin, Heidelberg, 1998. Springer-Verlag.

[11] https://github.com/btcsuite/btcd/graphs/contributors. BTCD – alternative full node bitcoin implementation written in Go, 2013. https://github.com/btcsuite/btcd.

[12] Markus Jakobsson and Ari Juels. *Proofs of Work and Bread Pudding Protocols(Extended Abstract)*, pages 258--272. Springer US, Boston, MA, 1999.

[13] Angshuman Karmakar, Jose Maria Bermudo Mera, Sujoy Sinha Roy, and Ingrid Verbauwhede. Saber on arm cca-secure module lattice-based key encapsulation on arm. Cryptology ePrint Archive, Report 2018/682, 2018. https://eprint.iacr.org/2018/682.

[14] Neal Koblitz. Elliptic curve cryptosystems. *Math. Comp. 48 (1987), 203-209*, 48(11):203--209, 1987.

[15] Bin Li. Bipartite threshold multi-secret sharing scheme based on hypersphere. *American Journal of Computational Mathematics*, 9:207--220, 2019.

[16] Jia Liu, Tibor Jager, Saqib A. Kakvi, and Bogdan Warinschi. How to build time-lock encryption. *Designs, Codes and Cryptography*, 86(11):2549--2586, Nov 2018.

[17] Xianhui Lu, Yamin Liu, Zhenfei Zhang, Dingding Jia, Haiyang Xue, Jingnan He, Bao Li, Kunpeng Wang, Zhe Liu, and Hao Yang. Lac: Practical ring-lwe based public-key encryption with byte-level modulus. Cryptology ePrint Archive, Report 2018/1009, 2018. https://eprint.iacr.org/2018/1009.

[18] A.J. Menezes, J. Katz, P.C. van Oorschot, and S.A. Vanstone. *Handbook of Applied Cryptography*. Discrete Mathematics and Its Applications. CRC Press, 1996.

[19] Daniele Micciancio. Inapproximability of the shortest vector problem: Toward a deterministic reduction. *Theory of Computing*, 8(22):487--512, 2012.

[20] Bitcoin node contributors. Bitcoin's difficulty adjustment jump limit, 2009. https://github.com/bitcoin/bitcoin/blob/78dae8caccd82cfbfd76557f1fb7d7557c7b5edb/src/pow.cpp#L56.

[21] BTCD Bitcoin node contributors. BTCD 's difficulty adjustment jump limit, 2013. https://github.com/btcsuite/btcd/blob/86fed781132ac890ee03e906e4ecd5d6fa180c64/blockchain/difficulty.go#L263.

[22] C. Padro and G. Saez. Secret sharing schemes with bipartite access structure. *IEEE Transactions on Information Theory*, 46(7):2596--2604, Nov 2000.

[23] Michael O. Rabin and Christopher Thorpe. Time-lapse cryptography. Technical report, Harvard University, School of Engineering and Applied Sciences, Cambridge, MA 02138, 2008.

[24] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300--304, 1960.

[25] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing*, STOC '05, pages 84--93, New York, NY, USA, 2005. ACM.

[26] R. L. Rivest, A. Shamir, and D. A. Wagner. Time-lock puzzles and timed-release crypto. Technical report, Cambridge, MA, USA, 1996.

[27] Robert Griesemer, Rob Pike, and Ken Thompson. Go programming language, 2009. https://golang.org.

[28] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612--613, November 1979.

[29] D. R. Stinson. An explication of secret sharing schemes. *Designs, Codes and Cryptography*, 2(4):357--390, Dec 1992.

A Graphs

A.1 ElGamal with SMA for difficulty adjustment

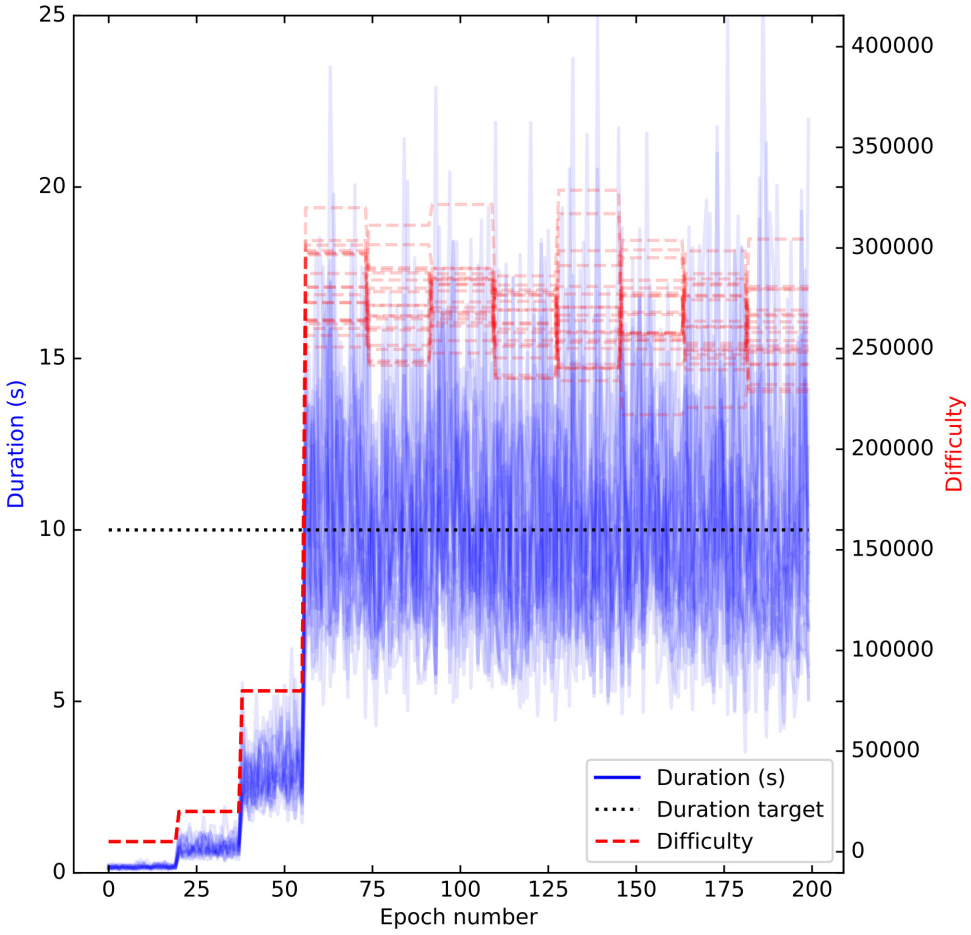


Figure 13. ElGamal as PoW validation mechanism with SMA for difficulty adjustment. 20 executions of 200 block epochs with 60 keys in each of them. Difficulty retargeting is performed every 18 epochs.

A.2 SHA256 with SMA for difficulty adjustment

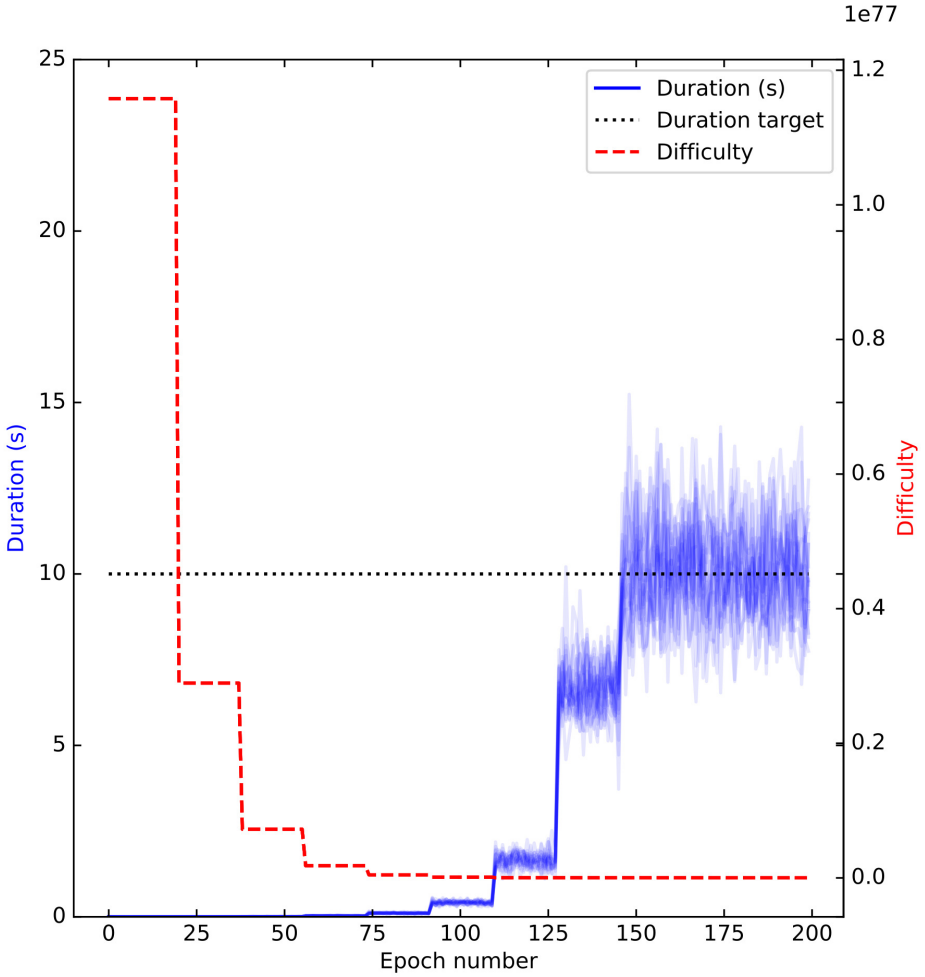


Figure 14. SHA256 as PoW validation mechanism with SMA for difficulty adjustment. 20 executions of 200 block epochs with 60 keys in each of them. Difficulty retargeting is performed every 18 epochs.

A.3 ElGamal with SMA for difficulty adjustment, last 60 blocks

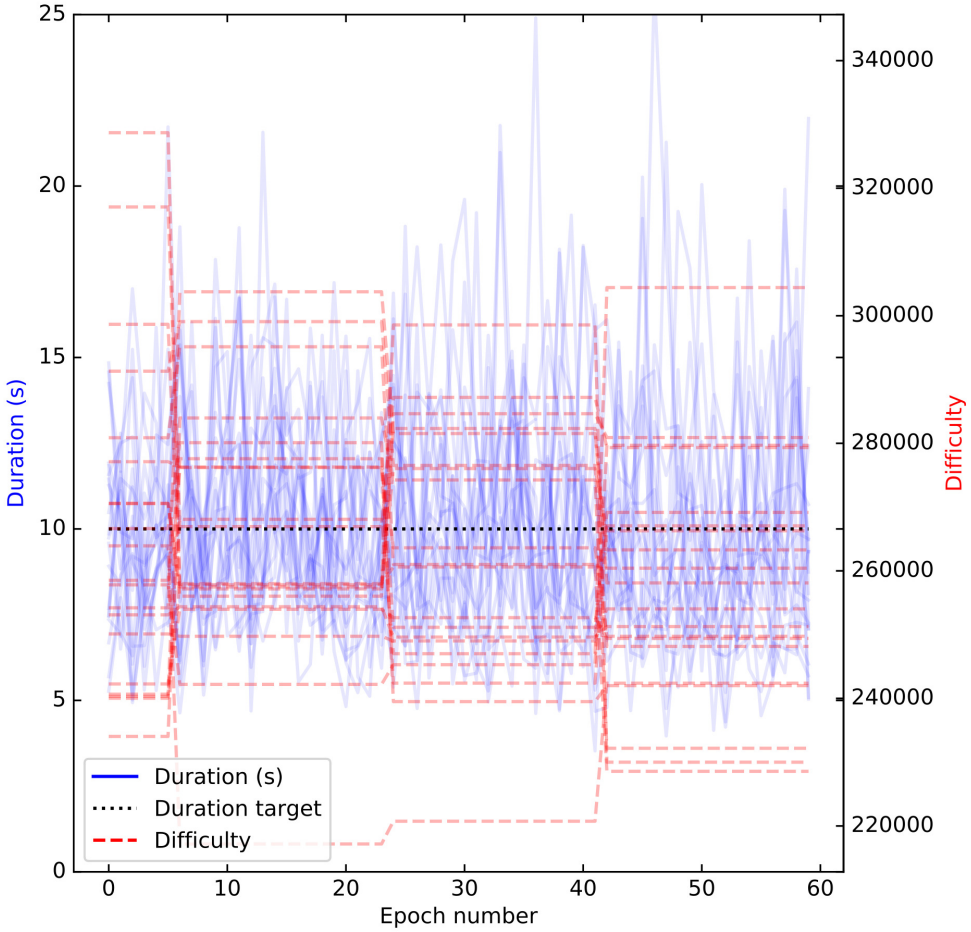


Figure 15. ElGamal as PoW validation mechanism with SMA for difficulty adjustment. 20 executions of 200 block epochs with blocks from 140 to 200 shown. Each epoch contains 60 keys. Difficulty retargeting is performed every 18 epochs.

A.4 SHA256 with SMA for difficulty adjustment, last 60 blocks

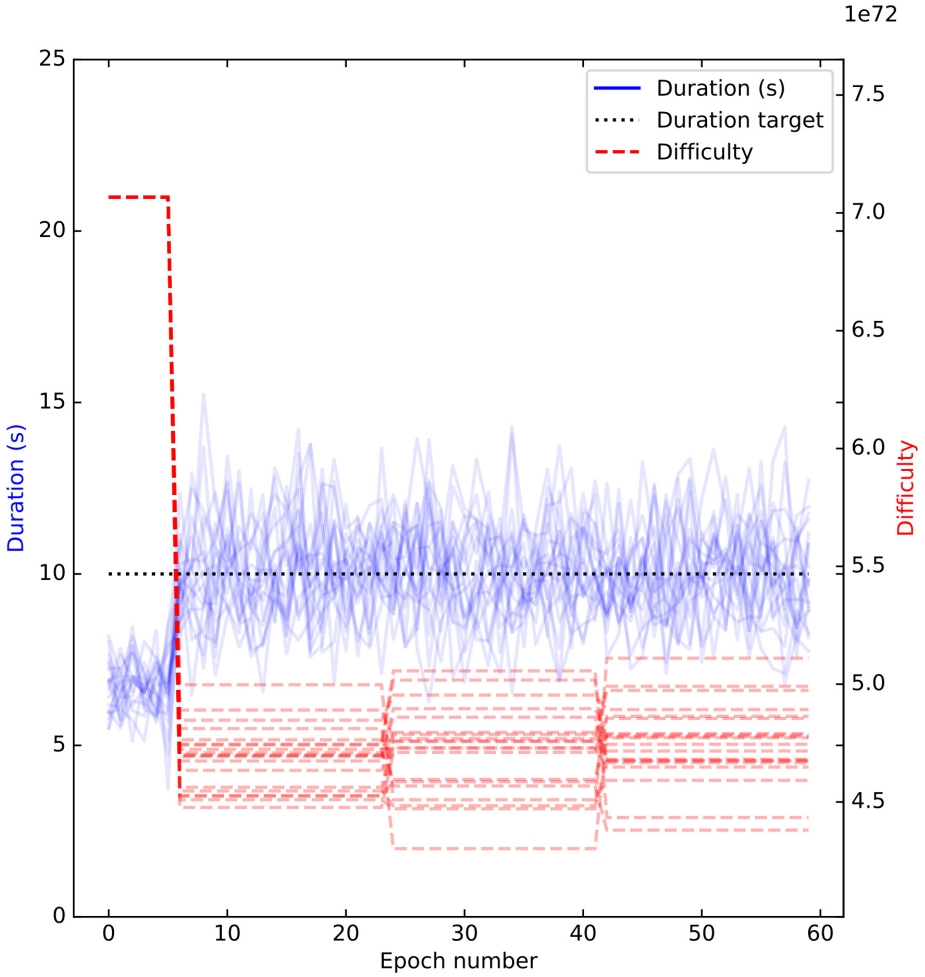


Figure 16. SHA256 as PoW validation mechanism with SMA for difficulty adjustment. 20 executions of 200 block epochs with blocks from 140 to 200 shown. Each epoch contains 60 keys. Difficulty retargeting is performed every 18 epochs.

A.5 ElGamal with Haar WT smoothing and SMA for difficulty adjustment

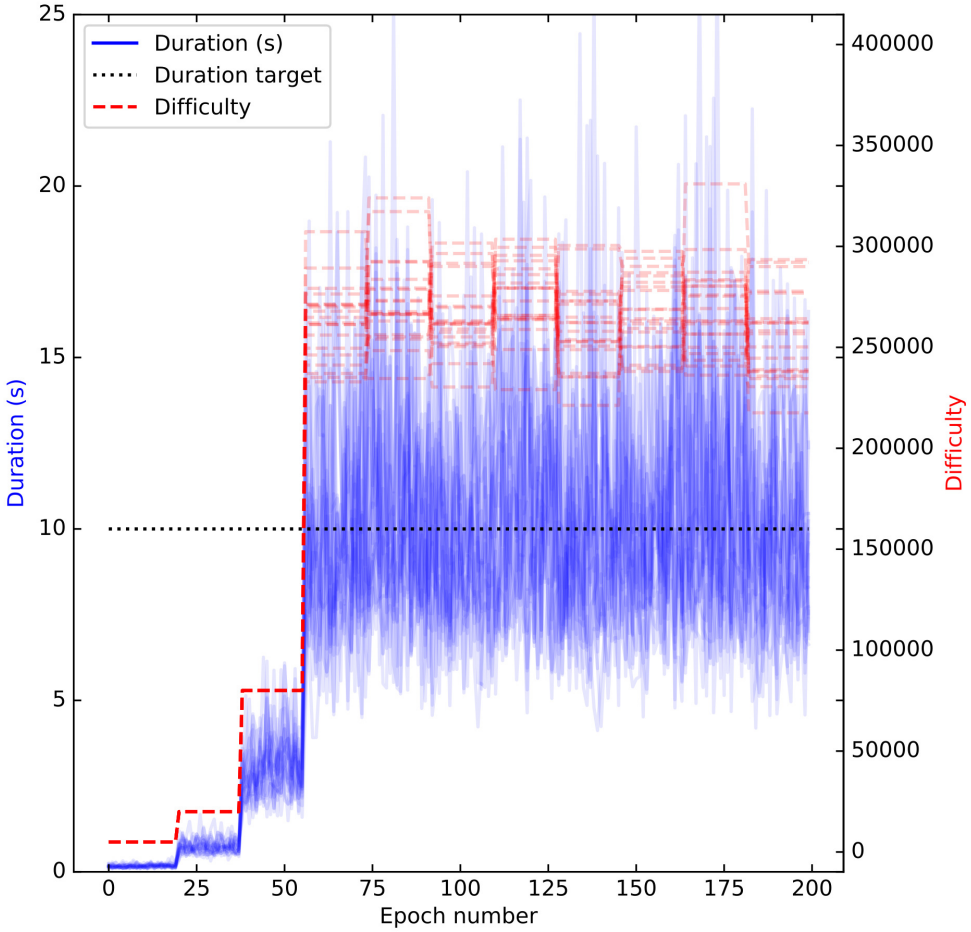


Figure 17. ElGamal as PoW validation mechanism with Haar WT for difficulty adjustment. Wavelet transform values are replaced with averages at the fourth layer of transformation. 20 executions of 200 block epochs with 60 keys in each of them. Difficulty retargeting is performed every 18 epochs.

B Elements of bipartite access structure

This code snippet returns refined version of results produced by equation 6.1. It returns a list of elements and doesn't include interleaving items. Code symbols are equivalent to equation 6.1: $w_d \equiv$ wd, $w_\alpha \equiv$ wa, $w_\beta \equiv$ wb, $l_\alpha \equiv$ la, $l_\beta \equiv$ lb, $i_\alpha \equiv$ ia, $i_\beta \equiv$ ib.

It's an example code that's written in Clojure programming language. It compiles and runs from browser: https://repl.it/languages/clojure.

Note: code copying breaks asterisks (*). They have to be manually replaced.

```
(defn find-thresholds [wd wa wb la lb]
  (for [ia (range 0 (inc la))
        ib (range 0 (inc lb))
        :when (and
                (>= (+ (* ia wa)
                       (* ib wb))
                    wd)
                (and (or (< (+ (* (max (dec ia) 0) wa)
                               (* ib wb))
                            wd)
                         (and (= 0 ia)
                              (* (max (dec ib) 0) wb)))
                     (or (< (+ (* ia wa)
                               (* (max (dec ib) 0) wb))
                            wd)
                         (and (= 0 ib)
                              (* (max (dec ia) 0) wa)))))]
    [ia ib]))
```

It's output is a list of index tuples ia and ib. Sample outputs:

```
> (find-thresholds 42 10 11 5 5)
([0 4] [1 3] [2 2] [4 1] [5 0])

> (find-thresholds 82 4 11 40 40)
([0 8] [2 7] [4 6] [7 5] [10 4] [13 3] [15 2] [18 1] [21 0])

> (find-thresholds 82 2 8 40 40)
([0 11] [1 10] [5 9] [9 8] [13 7] [17 6] [21 5] [25 4] [29 3] [33 2] [37 1])

> (find-thresholds 82 2 8 10 10)
([1 10] [5 9] [9 8])
```

C Application's development timeline

- 2017-12-15 – 2017-12-25 Initial version: Concurrent implementation of Bitcoin WIF format key cracking mechanism. Key serialization[28] is taken from BTCD Bitcoin node implementation [11]. WIF keys get loaded from a file into memory.
- 2018-11-24: Optimization and change of a random generator, 2x improvement in keypair production.

[28] Only WIF-based and base54 decoding functions.

- 2019-05-13: Restructuring for usage in alternative PoW-like algorithm tests. Worker part of the application is completely refactored into many files. First implementations of Elliptic curve cracking for any difficulty of a cryptosystem.
- 2019-06-19: Initial version of El Gamal's cryptosystem single key batch attack.
- 2019-09-04: Fixes of El Gamal key generation. Initial PoW mining implementation (with automatic difficulty adjustment) for El Gamal's cryptosystem.
- 2019-10-04 – 2019-10-08: Difficulty adjustment fix for El Gamal PoW implementation. Optimization – range-based public key matching method.
- 2019-10-11 – 2019-10-21: Implementation of SHA256 PoW mechanism[29]. Additional difficulty adjustment mechanism for SHA256 (See section 7.2).
- 2019-11-12 – 2019-12-15: Haar WT transformation for usage instead of SMA.
- 2019-12-16 – 2019-12-19: Difficulty-based Elliptic curve cryptosystem derivation and it's preparation for lossful ECIES PoW.

D Codebase size

Includes a copied library under directory "./ecies" which contains 302 lines of code. It had to be copied because the source was being changed. Directory has a README file.

Whole project:

```
1 $ gocloc --exclude-ext=txt,md,json,sh .
2 -------------------------------------------------------------------------------
3 Language     files      blank      comment    code
4 -------------------------------------------------------------------------------
5 Go              49        410          418    2607
6 BASH             5         10           29       5
7 -------------------------------------------------------------------------------
8 TOTAL           54        420          447    2612
9 -------------------------------------------------------------------------------
```

Tool: https://github.com/hhatto/gocloc/.

E Application's dependencies

E.1 Application 'bag' – Bitcoin address generator

Package './ecies' contains an implementation of ECIES scheme by Daniel Havir (https://github.com/danielhavir/go-ecies). Code in the package is a slightly changed implementation (4 lines of security checking logic removed from ECIES encryption code; simplification of hashing scheme; unit test added) that allows usage of very weak elliptic curve cryptosystems.

[29]The same is used in Bitcoin[1].

Cryptographic dependencies from Golang's standard library:

- ‘crypto/aes‘
- ‘crypto/cipher‘
- ‘crypto/ecdsa‘
- ‘crypto/elliptic‘
- ‘crypto/rand‘
- ‘crypto/sha256‘
- ‘crypto/sha512‘

Third-party dependencies (by relevance):

- ‘golang.org/x/crypto/openpgp/elgamal‘
- ‘github.com/oskanberg/gohaar‘
- ‘github.com/golang-collections/collections/stack‘
- ‘github.com/btcsuite/btcutil‘
- ‘github.com/btcsuite/btcd‘
- ‘gopkg.in/check.v1‘
- ‘github.com/davecgh/go-spew/spew‘
- ‘github.com/robert-zaremba/log15‘

Other dependencies from Golang's standard library:

- ‘bufio‘
- ‘bytes‘
- ‘encoding/base64‘
- ‘encoding/hex‘
- ‘encoding/json‘
- ‘flag‘
- ‘fmt‘
- ‘hash‘
- ‘io‘
- ‘io/ioutil‘

- ‘log‘
- ‘math‘
- ‘math/big‘
- ‘net/http‘
- ‘os‘
- ‘os/exec‘
- ‘reflect‘
- ‘runtime‘
- ‘runtime/pprof‘
- ‘strconv‘
- ‘strings‘
- ‘testing‘
- ‘time‘