

INTRODUCTION

The conventional digital signature scheme used in the Blockchain environment is Elliptical Curve Cryptography. However, this digital signature scheme can be easily cracked once quantum computers become a reality. It is therefore essential that we rely on quantum-resistant algorithms like lattice cryptography and leverage them into a blockchain environment. This article delves into the application and use of the Learning with Errors problem (LWE) as a potential candidate for a digital signature scheme.

LWE is a mathematical problem in lattice-based cryptography where, given noisy input-output pairs, the goal is to learn the underlying secret value hidden within the noise.

Let us try and formulate the concept of the LWE scheme-

1. The sender uses a pseudo-random number generator to generate the solution to a certain set of equations which we call as the private key.

Alice's **private key** for a certain transaction is $\begin{bmatrix} 8 \\ 7 \\ 2 \\ -1 \end{bmatrix}$ (suppose)

2. Every client has a public key associated with them. The public key is an N-dimensional linear equation that equates to 0. The equation is how a client is recognised in the Blockchain network. The equation is assigned as the public key once the client enters/joins the network. No two linear equations are the same i.e no two clients have the exact same coefficients for their variables on the LHS of the equation. There is no compulsion that the public key equation must always be generated in the same N-dimensions for all clients i.e Client A can have an equation $5x + 6y + 7z + 8a = 0$ whilst Client B can have an equation $6y + 15b + 20m + 8a = 0$. This equation is also called the cumulative equation (more on it will be elaborated in the protocol further) and never changes for a client in the network.

Alice's public key is $12x + 15y + 2z + 5a - 3b + 2c = 0$ (suppose)

Bob's public key is $12y + 15z + 2a + 5b - 3c + 2d = 0$ (suppose)

If the client wishes to introduce some sense of anonymity, then they can combine their equation coefficients and generate a single numeric string that signals their

identity. For now, this can solely be used locally i.e this numeric string cannot be used as a replacement for the broadcasting of this user's identity. Whilst the numeric string provides anonymity by masking the exact dimensions and the co-efficient values of the linear equation, it cannot be used as a unique marker because it is possible that although the equations of the clients are different, their numeric strings are same/alike.

To illustrate this, let us consider an example

Alice's public key is $12x + 15y + 2z + 5a - 3b + 2c = 0$ (suppose)

Bob's public key is $12y + 15z + 2a + 5b - 3c + 2d = 0$ (suppose)

Alice uses this anonymity technique to generate the following identity –

121525-3200 (suppose)

This equation is suitably anonymous because it does not reveal to any external entity what the dimensions of Alice's equations are and what her dimensional coefficients are.

Alice's x coefficient could be 1 or 12 or 121 or 1215 and so on...

Alice's kth coefficient (where k is the last coefficient) could be -3 or -32 or -320

I do not know just by looking at that numeric string the dimensions in which Alice's equations are in.

Notice that an additional zero has been added deliberately to increase the complexity of deciphering the last co-efficient. If this is not added, then the last 0 will directly indicate that it is the RHS of the equation and the number/numbers preceding it is the last dimension's co-efficient.

We will now generate Bob's anonymous numeric string. Bob's linear equation was as follows-

Bob's public key is $12y + 15z + 2a + 5b - 3c + 2d = 0$ (suppose)

Bob's Anonymous Numeric String (ANS) -- **121525-3200**

It is quite literally an exact copy of Alice's even though Bob's and Alice's equations/public keys are significantly different. One method that we could work with is that we randomly shuffle the dimensional values and generate new ANSs

that are dissimilar. Additionally appending any number of 0's in the RHS can significantly change the ANS and help us achieve the anonymity we wish for.

The only shortcoming of this concept/idea is that all entities in the network must keep a mapping of the linear equation (public key) and the corresponding ANS. Possession of the ANS cannot guarantee a specific client's identity the same way the possession of their public key does. Therefore, there is an inherent need to have a mapping of the ANS to the corresponding public key/ linear equation. If all clients anyway possess the linear equations/public key, why care to anonymise it? The whole purpose of anonymity is defeated if that happens.

One possible explanation we can provide for why a given client may wish to use this ANS is that the client may dislike viewing their accounts names (which is the public key) as equations. They may find it tedious to verify the identity of the receiver as they will have to check through an N-dimensional equation and verify the co-efficient values. While this task is no less difficult than verifying an ANS, clients usually prefer seeing numbers rather than mathematical equations. So this ANS can be useful in making it visually more appealing for the client to identify transaction entities and themselves on the network.

3. We use the concept of Learning with Errors to make the process of deciphering the private key from the public key difficult. To do so, every time a transaction is initiated, the client generates several N-dimensional linear equations, all of which are equated to 0 at the time of generation. The caveat that the client will have to follow is this – The sum of all the co-efficient values of a given dimensional variable must add up to the co-efficient of that variable that is enlisted as the client's public key.

To explain this concept, we will take an example of Alice's case

Alice's public key is $12x + 15y + 2z + 5a - 3b + 2c = 0$ (suppose)

For a given transaction, Alice generates the following equations

$$2x + 3y + 15z + 1a.... = 0 \quad \text{(Equation 1)}$$

$$6x + 10y - 20z + 1a.... = 0 \quad \text{(Equation 2)}$$

$$3x + 1y + 5z + 2a.... = 0 \quad \text{(Equation 3)}$$

$$1x + 1y + 2z + 1a.... = 0 \quad \text{(Equation 4)}$$

:

: (Equation n)

And, so on the equations can continue, but for understanding we will consider only these 4. If we sum up all these equations, we get

$$\begin{array}{rcl} 2x + 3y + 15z + 1a.... & = & 0 \\ 6x + 10y - 20z + 1a.... & = & 0 \\ 3x + 1y + 5z + 2a.... & = & 0 \\ 1x + 1y + 2z + 1a.... & = & 0 \\ \hline 12x + 15y + 2z + 5a.... & = & 0 \end{array}$$

These are precisely equal to the Alice's public key's x, y and z and a coefficients; therefore, these equations are all valid.

Care must be taken that all the coefficients must be integers. The public key will also contain only integer values and therefore, the equations must also be integer values. Special care must be taken to ensure that none of the coefficients in the equations are zero and none of the coefficients in the solution i.e the private key is zero.

4. Alice now plugs in the private key into all the equations generated and determines the true RHS of all those equations.

Alice's **private key** for a certain transaction is $\begin{bmatrix} 8 \\ 7 \\ 2 \\ -1 \end{bmatrix}$ (suppose)

Alice's respective equations are

$$2x + 3y + 15z + 1a = 0$$

$$6x + 10y - 20z + 1a = 0$$

$$3x + 1y + 5z + 2a = 0$$

$$1x + 1y + 2z + 1a = 0$$

(For the sake of simplicity we have considered only 4 equations and 4 dimensions)

The corresponding true RHS terms become

$$2x + 3y + 15z + 1a = 66$$

$$6x + 10y - 20z + 1a = 77$$

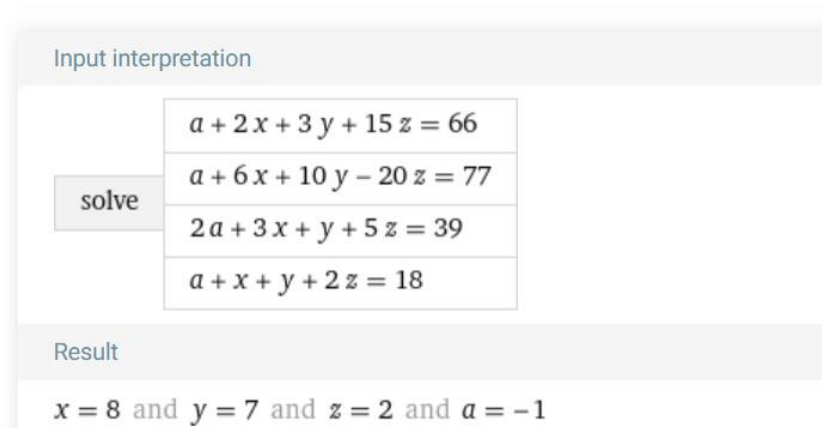
$$3x + 1y + 5z + 2a = 39$$

$$1x + 1y + 2z + 1a = 18$$

When we plug x, y, z and a accordingly

Example: $2x+3y+15z+1a \rightarrow 2(8)+3(7)+15(2)+1(-1)=16+21+30-1=66$ (67-1)

Now, if we were to send these equations as is, a classical computer can very easily decipher the private key. We will display that by passing these equations to a simple calculator.



The screenshot shows a web interface for solving a system of linear equations. It features a 'solve' button and a list of four equations: $a + 2x + 3y + 15z = 66$, $a + 6x + 10y - 20z = 77$, $2a + 3x + y + 5z = 39$, and $a + x + y + 2z = 18$. Below the equations, the result is displayed as $x = 8$ and $y = 7$ and $z = 2$ and $a = -1$.

Input interpretation	
solve	$a + 2x + 3y + 15z = 66$
	$a + 6x + 10y - 20z = 77$
	$2a + 3x + y + 5z = 39$
	$a + x + y + 2z = 18$

Result

$x = 8$ and $y = 7$ and $z = 2$ and $a = -1$

Diagram 1: Online Calculator depicting the solution of the four linear equations

Source: <https://www.wolframalpha.com/input?i=systems+of+equations+calculator>

Four equations are more than enough to solve and find the four variables. However, if there were 20 equations, it would make it slightly harder (must verify if the solution generated satisfies all the equations).

This validates our point that there is no significant use of just sending several linear equations with the true RHS.

5. Alice now uses another PRNG function to generate a certain boundary for error/noise distribution. The error will be introduced into every equation just to ensure that the RHS term is muddled/ obfuscated. The aim of doing so, is that now our classical or quantum computers will not be able to generate a definitive solution for the set of linear equations, thereby ensuring that the private key is preserved. Should they wish to generate the public key, they must first eliminate the noise/error factor from the RHS and then generate solutions. However, the equation they will see will be the muddled RHS (the true RHS + the error factor). Because, they are not able to fundamentally distinguish the true RHS from the obfuscated RHS, it is hard for even a quantum computer to brute force and fetch the solution i.e the private key.

Every equation cannot be plugged with the same error, this will defeat the purpose of plugging in errors. Alice therefore, generates a Gaussian distribution of the boundary generated by the PRNG (considered the mean of the Bell curve) and uses this distribution to plug in certain error/noise values into the respective equations.

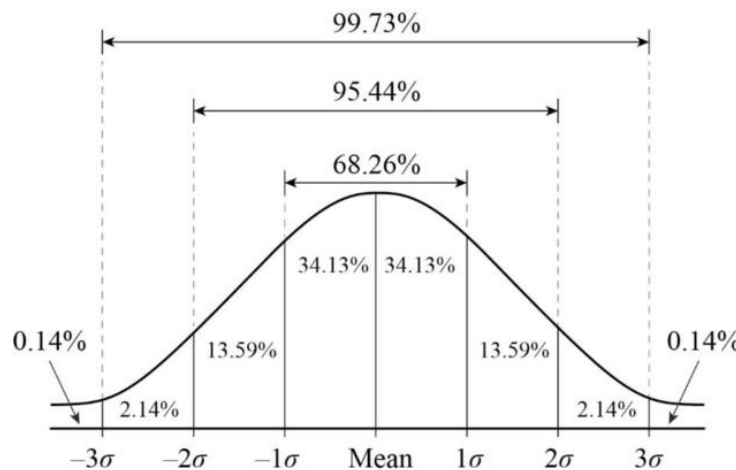


Diagram 2: A sample diagram that distributes a typical Gaussian Distribution

Source: <https://www.chegg.com/homework-help/definitions/normal-curve-31>

Alice must ensure that she does not append a uniform distribution of errors to her equations.

To explain this, let us consider the case where the mean is 0 and -3, 1 and 2 are valid error values generated from the curve.

These errors are now appended to every equation

$$2x + 3y + 15z + 1a = 66 \quad + 0 \text{ [Error factor]}$$

$$6x + 10y - 20z + 1a = 77 \quad -3$$

$$3x + 1y + 5z + 2a = 39 \quad +1$$

$$1x + 1y + 2z + 1a = 18 \quad +2$$

Effectively this provides us with the equations as

$$2x + 3y + 15z + 1a = 66$$

$$6x + 10y - 20z + 1a = 74$$

$$3x + 1y + 5z + 2a = 40$$

$$1x + 1y + 2z + 1a = 20$$

These equations will not generate a definite solution like before, and even if it does the solution will be a decimal, not an integer.

Now coming back to the point raised previously that Alice must not prepend uniform errors, let us take the two equation sets, the first set being the equations without any noise and the second set being the equations with the noise introduced and add them up together.

Set 1

$$2x + 3y + 15z + 1a = 66 \quad +$$

$$6x + 10y - 20z + 1a = 77 \quad +$$

$$3x + 1y + 5z + 2a = 39 \quad +$$

$$1x + 1y + 2z + 1a = 18 \quad =$$

Set 2

$$2x + 3y + 15z + 1a = 66 \quad +$$

$$6x + 10y - 20z + 1a = 74 \quad +$$

$$3x + 1y + 5z + 2a = 40 \quad +$$

$$1x + 1y + 2z + 1a = 20 \quad =$$

$$12x + 15y + 2z + 5a = 200$$

$$12x + 15y + 2z + 5a = 200$$

Effectively the two equations give us the same results. This happened because the error distribution (0, 1, 2 and -3) effectively cancelled each other out in the summation. Even though we embedded errors, they effectively cancelled each other out and revealed the true RHS in the summation equation.

Whilst it is true that an attacker who consolidates my equations will still not know if the consolidated equation is laden with an error summation (a non-zero error) or free from all errors (like the case discussed), it is always considered a safe practice to ensure that such faults do not occur.

Much of what we have elaborated so far on are the basics and essentials necessary to generate a digital signature which is primarily how this protocol will behave/function.

6. The primary aim of doing all these is to assist us with the generation of the digital signature. A digital signature helps us verify the integrity of the document and also the authenticity of the sender (that the sender is really the sender). We realise that conventional ECDSA and RSA employed digital signature algorithms will be very easily broken when Quantum Computers become a reality.

To thwart this, we discuss our protocol that uses concepts of LWE to generate a digital signature.

In a digital signature scheme, the message to be sent is first hashed. It is then encrypted with the private key of the sender and sent to the receiver. The primary aim of using a private key here is to mask the hash and not leave it as a plain entity. Once the receiver receives the same, the receiver will compute the hash of the message received and verify it with what the sender has sent. The public key is used to unlock the encrypted/mangled hash and it is then compared to check if the message is intact.

The presence of the key pair enforces the authenticity of the sender and the equivalence of the hashes verifies the integrity of the message.

The procedure employed in this protocol is very analogous to a traditional DS scheme but with some added computational overhead.

Black box hashing function $H()$

We will define a black box function $H()$, which basically hashes the contents passed into it. We are not defining it as a conventional hash function because the miner has another verifier black box function $H^{-1}()$ or $f()$ that extracts certain contents from this hashed message and verifies certain parameters (certain results (much of it will be discussed when we reach that section)). A traditional hash function is one-way and while we want to enforce the same for the message we generate, we still want to be capable of extracting individual details out of the hash function to a certain degree. Therefore, we define it as a black box hash function and not a regular hash function like SHA-256.

Identity Factor (IF)

Whilst the LHS of the public key equation can highlight that the message is originating from a specific client (all equations are unique), it still does not directly validate the fact that the client who possessed the public key was the entity that initiated the transaction/ sent the message. To allow for that, we decide on employing a novel approach that is very similar to how keys are generated in cryptography. Once a client joins the Blockchain network, an Identity Factor (IF) pair is generated for the client. An identity factor is defined as follows -

$$IF(\alpha) = \{ID_1, ID_2 \mid ID_2^a \neq ID_2^b \text{ for any } a, b\}$$

$$\text{And } ID_1 \oplus ID_2 = \alpha$$

Where α is a certain variable constant and \oplus is some mathematical function.

Certain characteristics of this IF function that must be upheld/followed –

- a) $\alpha \oplus ID_2^{-1} \neq ID_1$ (Inverse operand)
- b) $\alpha \oplus^{-1} ID_2 \neq ID_1$ (Inverse operator)
- c) $(ID_1 \parallel k_1 \parallel k_2 \parallel k_3 \parallel \dots \parallel k_n) \oplus ID_2 = (\alpha \parallel k_1 \parallel k_2 \parallel \dots \parallel k_n)$
(Commutativity)

Where k_1 to k_n are linear constants and \parallel is a certain joining mechanism

Depending on the \oplus function, either a) or b) will hold true

The characteristics will be discussed further when we go to decryption

What the math basically implies is that when a new entity joins the network, a certain constant is obtained and broken down into two identity factors ID_1 and ID_2 . The first ID factor originates at the client end and the second ID factor originates at the miner's end. It is true that there is no mechanism where keys/ values can originate on their own at two ends that are associated, but we will believe at this point that there is some secure mechanism through which both the entities receive the identity factors.

The client will include this identity factor in the hashed message that is generated to indicate that this is in truth the client.

The miner/s hold a distributed secure ledger that houses the details of every client and their corresponding ID_2 (no two IDs are same). When the hash is received, the miner extracts the ID factor from the message and then uses the stored mapping to verify if α is generated in truth. If it is, then it is a validation to the fact that this is the client itself.

If ID_1 of the client is ever compromised, it will risk impersonation but it will not result in ID_2 being deciphered because α is not known.

Likewise, if ID_2 and α is leaked from the miners' end, then computing ID_1 is not possible based on the nature of the criteria we have defined. Therefore, ensuring that ID_1 is not compromised is crucial.

The message

The most important aspect of any digital signature scheme is the message itself. We have generated a set of linear equations and appended some noise into them. The private key is used to generate the required RHS and correspondingly a PRNG (and Gaussian distribution) generates the error. The hashed message that is to be sent will be appended to this RHS term. The role of the private key is to mangle the hash/ obscure it. The same is being done by our private key, but differently. It is generating the true RHS, and with the addition of the noise and the message, we are technically obscuring what part of the equation on the RHS is the true RHS, the noise and the message. Therefore, the private key is in some capacity achieving the same tasks as a traditional DS scheme.

Consider an example-

Alice's combined equation $12x + 15y + 2z + 5a + \dots = 25134$

Alice knows what is the message, the noise and the true RHS because of the private key she possesses. To a stranger, there is no way to decipher/ identify what parts of the RHS are signifying the transaction at hand.

As previously suggested, we now want to append our message into the linear equations we generated. However, every message/transaction is not necessarily of the same size/length. How do we append these messages into the equations in such a way that no attacker/impersonator is aware that this equation contains my message embedding?

To solve this, we will follow the conventional approach followed in the DS scheme -- hashing. It must be understood that this hash function $h()$ is not the same as the black box function $H()$ defined previously.

We want a certain message/transaction of any length to be generated as an integer of fixed length so we can append them into our equations. Having a fixed length output will ensure that no 3rd party entity who views the equations is aware if the transaction initiated is small/big (I do not imply a small amount transaction here, I imply a small detail transaction, with lesser metadata).

To explain how this will work, let us consider an example

Transaction m

Sender: $12x + 15y + 2z + 5a - 3b + 2c = 0$

Receiver: $12y + 15z + 2a + 5b - 3c + 2d = 0$

Transaction amount: **0.05 BTC**

Transaction ID: **123789**

Nonce :

:
:

When this transaction m is passed into the corresponding hash function $h()$, we generate the following integer \rightarrow **1765421** (suppose)

So, $h(m) = 1765421$, which is the integer equivalent of the transaction suggested.

Now conventional $h()$ functions provide the hashed result as a binary number. However, since our aim is to generate its decimal form, we must take the binary result and appropriately convert it into decimal form.

The probability of two integers generated from their corresponding hashes being the exact same is very low. This indicates that $h(m)$ will always generate a unique solution.

This $h(m)$ is primarily what is verified at the miner's end in the conventional protocols, the same will be done in our case too.

However, $h(m)$ is not simply appended into the equations. As suggested previously, we have to account for the identity factor and another term, called the nonce. Once we have accounted for them both, we will have to pass these three arguments into the $H()$ black box and generate the final result that is going to be appended into the equations.

Nonce

In this section, we shall discuss different aspects pertaining to nonces. These will include the need for a nonce, the generation of nonces and counter-nonces and what role they play in the equation.

A nonce is a unique random number that is generated, usually by the server to uniquely identify a certain session/transaction with the client. It is used to introduce randomness and ensure that every transaction is unique and valid only for a certain duration.

The primary reason why nonces are introduced in protocols is to ensure that attackers do not carry out replay attacks.

A replay attack is an attack where the attacker stores certain sensitive exchanges between the client and the server, and later uses the same message/exchanges to interact with the server. With no nonce, the server will not be able to identify that a certain exchange/ transaction has already been completed before.

If an attacker were to fetch the hashed message and the identity factor embedded in the RHS of the equations (they do not need to know what part of it is the hash and what part of it is the message), then they could append it to any new equation they craft, such that the LHS is the same as the client, and cause a certain transaction to constantly happen (because the miners verified it). To protect against it, we use nonces.

Whilst nonces are unique, the main distinguishing factor that makes them truly usable in a cryptosystem is the fact that they have a certain time validity associated with them. This time validity basically indicates the time for which that nonce is considered active/usable. After its validity, it is rendered incapable. So, even if the attackers were to store the nonces and then append them into new messages, because those nonces had expired, those transactions/messages would be considered replayed transactions/messages and invalidated. A nonce's validity time is typically small, so the time window that the attacker has to replay an attack is very small.

However, one point we do not account for in nonce usage is this. When exchanging the nonce securely, the server will have to provide the time validity details to the client. If an attacker is able to identify this parameter, which is included like a JSON key value pair, then an attacker can forge the timestamp/validity to a time of the attacker's liking. If the server verifies this JSON parameter timestamp to determine the validity of the nonce, then clearly the nonce is compromised, and the whole point of thwarting replay attacks is lost. Traditionally to protect this, digital signatures are used on the nonce parameters. However, using a Digital Signature inside another digital signature will be highly impractical. Therefore, we devise another approach, where we generate something called counter-nonces.

Two nonces will be generated, one the actual nonce n possessed by the sender and the counter-nonce cn , possessed by the miner. The nonce and the counter-nonce are mathematically linked as pairs. For every nonce generated, there is an equivalent counter nonce generated.

We represent the same in mathematical form below

$$\text{Nonce generation } \tilde{n} = (n, cn) \mid n \Theta cn = \begin{cases} 1 & \text{if } n \text{ and } cn \text{ are not the right pair} \\ 0 & \text{if } n \text{ and } cn \text{ are the right pair} \end{cases}$$

Where Θ is some mathematical operation/function

Characteristics of the (n, cn) pair

$$a) (n \parallel k_1 \parallel k_2 + \dots \parallel k_n) \Theta cn = \begin{cases} (1 \parallel k_1 \parallel k_2 \parallel \dots \parallel k_n) & \text{for invalid pair} \\ (0 \parallel k_1 \parallel k_2 + \dots \parallel k_n) & \text{for valid pair} \end{cases}$$

Where \parallel is some joining mechanism

- b) Θ is a verification function that generates binary results, it cannot be used to extract cn or n, given one of it is known.

Based on this concept of nonces and counter-nonces, we will now understand how they are used in this protocol. When a sender wishes to initiate a transaction the nonce is generated at the sender's side and the counter-nonce is generated at the miner's end. The miner stores inside a secure ledger, a mapping of the latest counter-nonce associated with the public key of the sender. It is from this dictionary/ledger that the miner will fetch the counter-nonce and verify if the transaction received is valid.

There are two alternate approaches suggested which incorporates the concept of time validity for the nonces.

A) Validity time is associated with the counter-nonce

Every counter-nonce that the miner generates for a valid nonce, is associated with a certain expiry time at the miner's end. The miner's clock generates a certain validity time for every counter-nonce that is associated with a specific client. If the counter nonce is not used within its expiry time, then it indicates that there is some discrepancy with the transaction. It could either be the fact that the transaction never got approved (due to insufficient funds/ incorrect destination address etc.) or that someone is trying to replay the transaction at a later time. In both cases, there is no need to preserve/ keep the counter-nonce active and valid for that user. So once the time is expired, the counter-nonce is discarded. If that very transaction is received by the miner at a later stage, when it parses the ledger for the counter-nonce, it will identify that there are no valid counter-nonces catering to that specific transaction ID, and therefore will render that transaction invalid. More on this will be elaborated further when we discuss $H^{-1}()$ black box function.

B) The nonce is mangled after its validity is expired

In this approach, we associate the validity period with every nonce such that the time is an intrinsic property of the nonce (possibly like a class data structure where nonce and timestamp as attributes). A mangling function is embedded into this class and access to the nonce is made private i.e only functions defined in this class can display the nonce. Once the validity of the nonce expires, the mangling function is called, which permanently mangles/defaces the nonce and changes it into something incomprehensible by the miner. When the miner applies the Θ verification function, the result will be 1, which will indicate that the key pairs are not valid anymore.

Whilst it is true that an attacker can copy the old nonce and just use that instead of this data structure, we can make sure we enforce the fact that the miner will only accept this nonce class and not a single nonce variable. Since the attacker is not aware of the constitution of the nonce class, he cannot re-create this data structure and therefore, possession of the old valid nonce will still not help him/her/them with replay attacks.

The final choice on which approach to select out of the two depends on the requirements and constraints of the blockchain system itself. It depends on what the entities of the network want to prioritise. If they wish to transfer the burden of managing the validity of the nonces, then the first approach will be ideal. If they wish to reduce the burden of the miner (who already has to generate and manage the IF securely), then the second approach will be more feasible.

COMBINING ALL THE COMPONENTS

We now understand that there are three important components that need to be included in the final equation that is to be sent to the miner. The first entity is $h(m)$, the second is ID_1 (we shall represent it as i for simplicity) and the third is n . $h(m)$ is the hash of the transaction, i is the identity factor of the client and n is the nonce for that transaction.

All these three parameters are passed to the black box $H()$ function where they are first combined and then a corresponding result is generated.

So, the output of the $H()$ function can be depicted as $H(h(m) || i || n)$.

We will provide a short-hand notation for $H(h(m) || i || n)$ and call it the Final factor (FF).

What does the $H()$ function truly do? Do we really need it?

We understand that $h(m)$ generates a result that is unique – no two transactions will generate the same integer. Similarly, we understand that nonces are also generated by PRNGS and so therefore, they are unique too. The identity factor may not necessarily be unique, as our constraints suggest that ID_2 must always be unique. But, for understanding and simplicity, let us consider an ideal case, where I am even capable of making ID_1 unique. So now all the three parameters are unique.

Can I just append $h(m) || i || n$ to the final equation then? Each of those entities are unique which is true, however, there is no guarantee that on joining them using $||$ (which is some joining mechanism, either an OR function or a summation function), the corresponding result will be unique too.

To indicate this, we will consider a simple arbitrary example

Case 1

$$h(m) = 5 \quad i = 7 \quad n = 5$$

$$h(m) || i || n = 5+7+5 = \mathbf{17}$$

where $||$ is summation function

Case 2

$$h(m) = 3 \quad i = 4 \quad n = 10$$

$$h(m) || i || n = 3+4+10 = 17$$

We see a clear case where unique parameter values still yield the same result. This can support our case that we need the outer black box $H()$ to make this result unique too.

But another valid question comes up – so what if two situations yield the same final result?

The attacker may generate the same $h(m) || i || n$'s final result, but when they are individually extracted in the $H^{-1}()$ function, the attack will fail, because of the introduction the counter-nonce and the ID_2 and α factors for that client into the equation. So, just using $H()$ as the joining mechanism and not as a fresh black box hash function could still a valid and secure approach.

It is true that if the attacker has access to the ledger storing all the crucial parameters like ID_2 and α , then using $H()$ as a black box function will be more secure, but we will not be dealing with such extreme cases in this paper.

GENERATING THE FINAL SET OF EQUATIONS

We have accounted for all the various parameters that were essential to fully protect a transaction's integrity, authenticity and validity. Now, we must return back to the first discussion we had on the LWE problem and the set of linear equations that are generated by the client.

There are two approaches we can follow from here –

A) Generate a single equation

In this approach, we combine all the set of equations (summation) and generate a single equation with the true RHS + total error/noise. We add our final factor (FF) directly to the RHS of this final equation. Once complete, we have a single equation that resembles say this

Alice's equation: $12x + 15y + 2z + 5a - 3b + 2c = 37975641$

The LHS is the same as the public key of Alice. The RHS now contains 3 components, the true RHS, the noise/error and the final factor FF which is basically $H(h(m) || i || n)$.

This idea seems feasible because it does not involve sending several equations on the network and relying on the network to ensure correct, timely delivery of all equations. It also minimises the workload of the miner as the miner does not have to do the summation on receiving the equations.

However, the only flaw in this algorithm is the security of the private key. The greater the number of equations there are, the more difficult it is to generate an exact solution. If only a single equation is sent out, the attacker only needs to brute force different values in different dimensions until the final summation exceeds the RHS (the worst-case scenario). Then different keys can be tried to extract out the FF in some capacity. If the attacker is able to replicate/ take over a miner, then the $H^{-1}()$ function will stripe the FF off and then the attacker can brute force and obtain the private key.

Whilst it is true that possession of the private key will not compromise the victim entirely (only that transaction), one transaction is enough for the

attacker to transfer all the funds into their account. Similarly, if the IF is compromised, then the client is always vulnerable.

B) Generating multiple equations

In this approach, we send the miner all the equations that we generated. If we are sending all the equations, then how are we sending the FF?

To account for that, we decide to generate the FF first and then break it down into several disproportionate chunks. Once broken down, these disproportionate chunks are all added to the various equations in that set. Once that is completed, each equation in that set has a true RHS, a noise factor and a portion of the FF on the RHS of it. By breaking the FF into several disproportionate chunks, we ensure that not all equations have the same amount of the FF. An attacker does not potentially know what portion of the FF is part of which equation. Any manipulation they try to do to these equations will result in the FF changing and thereby rendering the transaction invalid.

To explain the same mathematically,

Let us consider the final factor term as M.

Alice's equations are as follows

$$2x + 3y + 15z + 1a = (66 + 0)$$

$$6x + 10y - 20z + 1a = (77 + 3)$$

$$3x + 1y + 5z + 2a = (39 + 1)$$

$$1x + 1y + 2z + 1a = (18 + 2)$$

(For simplicity we will consider that she will broadcast only these 4 equations)

The final factor M is broken down into n chunks where n is the number for equations there are such that

$$A) \quad M_1 \parallel M_2 \parallel \dots \parallel M_n = M$$

where M_i is the i^{th} chunk added to the i^{th} equation in the set and \parallel is some joining function

$$B) \quad \text{Len}(M_i) \neq (M // n) \text{ for all } M_i, \text{ but can be true for some } M_i$$

Where $\text{Len}(x)$ is the length of chunk x and $//$ is integer division

Each of these M_i s are now appended to their corresponding equations

$$2x + 3y + 15z + 1a = 66 + M_1$$

$$6x + 10y - 20z + 1a = 80 + M_2$$

$$3x + 1y + 5z + 2a = 40 + M_3$$

$$1x + 1y + 2z + 1a = 20 + M_4$$

Once these chunks are appended, these set of equations are sent across to the miner. The miner collects all these equations together and then sums the result up, so he/she/they can generate the final consolidated equation. The consolidated equation now satisfies characteristic A) of the message breaking mechanism, and therefore provides the FF i.e M.

Miner's summation function result

$$12x + 15y + 2z + 5a = 206 + (M1 || M2 || M3 || M4)$$

$$= 12x + 15y + 2z + 5a = 206 + M$$

Let us consider that M is 194 suppose,

The equation that the miner will see on summation will be

$$12x + 15y + 2z + 5a = 400$$

The miner is unaware of what part of 400 truly belong to the true RHS, the noise and the final factor. The miner uses this consolidated equation to verify the authenticity, integrity and validity of the transaction that was initiated.

This scheme is relatively better in terms of security than the previous one, because there are multiple equations that the attacker must brute force if he/she/they intend to obtain the private key. The vulnerability posed by a single equation is eliminated here.

An attacker is unaware of what portion of the message chunk is stored in what equation. Tampering with one equation can significantly change the final FF (irrespective of whether the message chunk in that equation is small or large). The reason we claim so, is that whatever the black box function generated, is but a unique string/stream. Any minute tampering of it will change the string, indirectly changing the underlying mapping details, thereby significantly changing one or all of the underlying parameters. Therefore, in this scheme an attacker may not be tempted to tamper with the equations, realising its consequences.

However, there are other challenges we must account for. It is possible that due to network delays, certain equations do not reach the miner in a timely fashion. If the miner does not receive even one equation, then the corresponding FF will not be accurately generated. It indicates that whilst

we broke down the message load amongst the equations, the equation importance still stayed the same. Every equation was just as equally important as any other, even though they contained smaller chunks/details of the final factor.

Whilst our scheme demotivates an attacker from tampering an existing equation, it is still vulnerable to an addition based tampering attack. An attacker may craft his own new equations such that after their addition too, the LHS stays the same. He can append his own message chunks and other details and send it across with the existing equations. Whilst this will definitely fail at the miner's end, it can still cause availability issues (the client is never able to initiate a transaction).

To protect from this, we can ensure that we fix n , where n is the number of equations that are to be sent. n does not need to be static, it can be dynamically decided and altered by the consensus rules of the network. A fixed n will ensure that no attacker is able to prepend their equations into the existing equation set and cause availability issues.

There is another availability issue at hand. An attacker may decide that whilst he does not want to add his own details to the existing equations, he/she/they want/s to make sure that the sender is unable to ever make a valid transaction. To do so, they deliberately modify a given equation's RHS, only so that the validity, integrity and authenticity checks fail, and the transaction is rendered invalid.

To protect against this attack, we need to enforce a certain mechanism where I can make the equations immutable. It is an ideal case scenario. An equivalent approach is the conventional approach followed today. Use public key cryptography to encrypt the exchanged messages, so that no one is able to snoop into the equations in our case and modify it.

At this juncture, we will assume that there is some quantum resistant encryption algorithm in place that securely exchanges the messages I need.

The trade-offs of opting for both the schemes have been discussed. We will now, move to the miner's end and understand verification next.

The sender/client has generated the equations, appended the message chunk and generated the final digital signature that is to be sent to the miner. The equation(s) are broadcasted in the network and whichever miner picks up this transaction from the memory pool, is in charge of verifying the digital signature of the sender.

Once the miner has received the equation(s), as previously suggested, he/she/they generate/s a consolidated equation that holds the FF in its true but latent form.

We will now discuss the verification of this DS scheme on the Miner's end and the additional responsibilities we will provide the miner.

The miner's black box function $H^{-1}()$

The miner's black box function is composed of several intermediate components that are all used to verify the authenticity, validity and the integrity of the transaction at hand.

To briefly list these entities out, they are

- A) The distributed ledger with the mappings (hash algorithms are used to speed up the search process)
- B) The A-box (Authenticity verification box)
- C) The Nonce-box (Validity verification box)
- D) Hash computer ($h(m)$)
- E) Two $h(m)$ verifier boxes (Integrity verification box)
- F) The ZKPV box (Zero-knowledge proof verification box)

The reason why they are called boxes are that each box has a certain activity it completes, that is more or less unique to that particular entity. Therefore, all the set of tasks associated with that particular activity are grouped into one box.

The boxes are written in the same order in which they will operate inside the $H^{-1}()$ function. The black box takes the transaction details (m), the digital signature or the final equation received (after summation), and details from the distributed ledger. If there are any additional inputs that this function takes, we will mention them as and when they appear.

We will go through them in order and understand how they work.

1. The secure distributed ledger for miners

As the name suggests, this is a log/ ledger that holds the following details in a key value pair format (like a dictionary) -

Public key equation: $\{ (ID_2, \alpha) , [\text{Counter-nonce} : \{ \text{Validity period, Timestamp} \}, \dots]$

The **public key equation** is used as the **key** to this dictionary to associate all the attributes of a specific client to that client

ID₂ and **α** are **immutable** entries, they are stored as a tuple.

Counter-nonces are unique to every new transaction and are therefore stored as an array. Whilst it is true that usually no client will initiate multiple transactions within a very short time span (within the validity period of the first counter-nonce generated), we still account for several counter-nonces being present, all valid at a given time and therefore store it as an array.

Each counter nonce is associated with certain attributes/metadata and therefore, the counter-nonce metadata is associated with the counter-nonce as a key value pair again.

The **validity period** is the time for which the given counter-nonce is available in the ledger. Once it reached 00:00:00:00, then the counter-nonce is popped out of the ledger and rendered invalid.

The **timestamp** is primarily included to account for multiple valid counter-nonces existing at the same time. Which counter-nonce to select can be verified via the counter-nonce timestamp which will synchronise with the nonce timestamp.

This ledger is present with every valid miner in the network and appropriate hash search algorithms are employed to speed up the search of the ledger. The process of pushing the counter-nonces into the ledger and popping it out is handled by the counter-nonce generator and will not be discussed here. The primary reason for including this entity first, is that one of the most crucial inputs to the $H^{-1}()$ function comes from the secure distributed ledger.

One must not confuse this with the distributed ledgers maintained by all the clients in the blockchain network. That is a different entity entirely. This ledger is solely possessed by valid miners and is securely sealed/locked. A miner is aware of the existence of the ledger, but a miner cannot modify the contents of it, the network rules and protocols and the generators do it automatically.

2. The Authentication Box (A-box)

The first and foremost part of the $H^{-1}()$ function is the A-box. The authentication box primarily verifies the authenticity of the client. Before the validity or the integrity of the transaction is checked, it is important to verify if the transaction truly originated from the sender. Public key cannot be a useful means of validating that and therefore, we decided to use Identification Factors for the same.

We will briefly recall the third characteristic property of the Identification Factors here –

$$\text{R3: } (ID_1 \parallel k_1 \parallel k_2 \parallel k_3 \parallel \dots \parallel k_n) \oplus ID_2 = (\alpha \parallel k_1 \parallel k_2 \parallel \dots \parallel k_n)$$

What this rule implies is that if we were to use the \oplus function to operate ID_2 on ID_1 that is joined with some other n constants, we will generate α , which is joined to those n constants.

What this implies in simple terms is that joining of additional constants to ID_1 does not affect the way it operates with ID_2 on applying the \oplus function.

We will leverage this rule on our Digital Signature. We will fetch the final equation generated. We will use the LHS of the equation to identify this specific client's attributes in the distributed ledger. We will then fetch the ID_2 and α entities from it. Once we have done so, the A-box performs the \oplus operation on the RHS of the final equation.

If ID_1 is valid, then it will generate α , on operating with ID_2 . If it is not, then it will generate some garbage result.

But the interesting thing to note is that, this box has indirectly fetched the identity factor of the client and neutralised it back to α , without

directly extracting ID_1 . To facilitate easier understanding of all the concepts from here onwards, we will consider the following equation

$$\text{LHS} = (\text{True RHS} + \text{noise}) \parallel h(m) \parallel ID_1 \parallel n$$

The reason we are not writing $H(h(m) \parallel ID_1 \parallel n)$ here is because we are currently inside the $H^{-1}()$ function, so whatever mangling $H()$ has carried out, is considered to be reversed by the $H^{-1}()$ function. Therefore, we have excluded mentioning it as $H()$.

On being operated by the A-box, the corresponding equation becomes

$$[(\text{True RHS} + \text{noise}) \parallel h(m) \parallel ID_1 \parallel n] \oplus ID_2 = (\text{True RHS} + \text{noise}) \parallel h(m) \parallel \alpha \parallel n$$

Provided ID_1 is correct.

Now if we were to unjoin the α term (or more like extract it out from the equation), our new equation would become

$$\text{LHS} = (\text{True RHS} + \text{noise}) \parallel h(m) \parallel n$$

A valid extraction would eliminate the Identity term from the very equation itself, thereby ensuring that the identity is validated.

It is true that at this box, we cannot confirm if the identity is truly valid, as we do not know in truth at this stage, if we have correctly removed alpha and generate the above equation or generated this –

$$\text{LHS} = (\text{True RHS} + \text{noise}) \parallel h(m) \parallel n \parallel \text{garbage-result}$$

To continue with the verification, we will now move to the next box, the validity/nonce box.

3. The Nonce box

Once the authentication of the transaction has been done (not yet verified), the modified equation now reaches the nonce box. The latest counter-nonce that is associated with this client is fetched from the distributed ledger to operate on this modified RHS.

If no counter-nonces exist then it indicates that this transaction is either a replay attack or a delayed one. In either of the cases, the

absence of the counter-nonce indicates that the equations cannot be operated upon.

In such a scenario, the transaction is considered invalid and rejected. The nonce box generates an **F1 error**, which is shown to the miner. More on these error messages will be elaborated once we delve into the additional roles and responsibilities provided to a miner. But for now, we can understand that an F1 error indicates a failure of the validity of a given transaction. Additional metadata is associated with the error message to highlight the condition that triggered the error.

If the counter-nonce is present, then it is fetched and operated on the existing nonce, using the Θ function. Again, as a reminder, we will recollect the characteristic of the nonce, counter-nonce pair –

$$(n \parallel k_1 \parallel k_2 + \dots \parallel k_n) \Theta cn = \begin{cases} (1 \parallel k_1 \parallel k_2 \parallel \dots \parallel k_n) \text{ for invalid pair} \\ (0 \parallel k_1 \parallel k_2 + \dots \parallel k_n) \text{ for valid pair} \end{cases}$$

It is similar to what we have seen previously. If the nonce and the counter-nonce are a valid pair, then they effectively cancel each other out. If they are not, then the result is the generation of 1 as the final result.

In an ideal situation, where the A-box has correctly eliminated the identification factors, the Nonce box will correctly eliminate the nonce factor in the RHS of the equation without accessing the nonce directly.

Unlike the A-box however, the Nonce-box generates two equations. We will understand why it does so, with examples and equations

The input to the Nonce-box can be one of the following two

a) **LHS = (True RHS + noise) \parallel h(m) \parallel n**

b) **LHS = (True RHS + noise) \parallel h(m) \parallel n \parallel garbage-result**

While the two equations are different, the Nonce box only caters to/operates on the nonce part of the RHS. It considers the garbage-result in b) as just another constant.

So, irrespective of the actions of the A-box, the Nonce-box only caters to resolution of the nonce portion of the equation, so it would not matter if the RHS was that of a) or that of b). For simplicity's sake, we will take equation a) as our standard equation.

Here, there could be two possibilities-

$$\text{a) RHS} = (\text{True RHS} + \text{noise}) \parallel h(m) \parallel n$$

$$\text{b) RHS} = (\text{True RHS} + \text{noise}) \parallel h(m) \parallel n''$$

a is the case when the nonce is valid i.e the nonce is the correct pair opposite of the counter nonce. b) is that case when the nonce is invalid i.e it does not align with the counter-nonce fetched from the ledger.

We will now consider these two equations as our objective equations and apply the Θ function on them respectively

$$\begin{aligned} \text{a) } &[(\text{True RHS} + \text{noise}) \parallel h(m) \parallel n] \Theta cn \\ &= (\text{True RHS} + \text{noise}) \parallel h(m) \end{aligned}$$

$$\begin{aligned} \text{b) } &[(\text{True RHS} + \text{noise}) \parallel h(m) \parallel n''] \Theta cn \\ &= (\text{True RHS} + \text{noise}) \parallel h(m) \parallel 1 \end{aligned}$$

So, we observe that if the nonce is valid, we will resolve the nonce factor and leave only the hash and the rest of the terms. Similarly, if the nonce is incorrect, we will have a 1 included in the equation, which is analogous to the inclusion of garbage in the previous A-box.

The Nonce-box is not aware if it has resolved the nonce or not. Therefore, it would have to generate two equations and then carry them forward.

The first equation generated as an output of the Nonce-box would be

$$\text{LHS} = (\text{True RHS} + \text{noise}) \parallel h(m)$$

This is the ideal equation, where the nonce is completely eliminated.

The second equation should be

$$\text{LHS} = (\text{True RHS} + \text{noise}) \parallel h(m) \parallel 1$$

However, generating this equation and carrying it forward will not provide any useful results. More on this will be elaborated upon, once we reach the final box. For now, we will understand that we cannot pass this equation with the 1. We will stripe it off and pass the rest of the equation along to the next box.

To distinguish between these two equations (they have become alike now), we will call the ideal equation as LHS0 and the incorrect nonce equation as LHS1 (because 1 was the stripped-out factor).

To re-iterate a point here, even if the equation contained a garbage result from the authentication box, it is still okay; the Nonce-box handled the nonce (either valid or invalid). So, the overall equation that is reaching the hash verification box, is

$$\text{LHS} = (\text{True RHS} + \text{noise}) \parallel h(m) \dots\dots\dots <1>$$

OR

$$\text{LHS} = (\text{True RHS} + \text{noise}) \parallel h(m) \parallel \text{garbage-from-Abox} \dots\dots\dots <2>$$

Where LHS is either LHS1 or LHS0

So, there are two hash verifier boxes that succeed the Nonce-box.

The valid modified equation (LHS0 which could either be <1> or <2>) goes to the first hash verification box. The invalid modified equation (LHS1 which could either be <1> or <2>) goes to the second hash verification box.

If the boxes on the valid side succeed, then it automatically proves that the authenticity, integrity and validity of the message are preserved. The invalid evaluation can be straight away discarded.

If the valid box side fails, then it indicates that something is wrong (in one of the parameters). We then check the invalid side of boxes. If the invalid side has succeeded, then it means that authentication and integrity of the message was preserved, but its validity was compromised (the result had a 1 which we deliberately striped out).

We flag the **F1** error to highlight that the integrity is compromised. So, there are two circumstances when the **F1** error is flagged.

If both the valid and the invalid box fails, then it can indicate that there was no issue with the validity of the message, one of authenticity or integrity must have failed. If integrity can also be verified, then it will straight up indicate that the two verification boxes received equation<2> which is why both sides failed, whilst integrity did not.

When that happens, we will generate an **F0** error, when both the hash verification boxes fail but we realise that the integrity is not the factor that is compromised. **F0** error indicates to the miner that this transaction did not originate from the owner of the public key, but instead by some other malicious entity.

4. Hash generator

This is similar to $h(m)$ that is calculated at the client end. It is assumed that once the A-box and Nonce box's actions are complete, some form of authenticity and validity check are completed, and so the hash can start being generated. Again, the hash generated here must also be converted into an integer so it can be extracted out later.

5. Hash verification function / Hash box

As the name suggests, this function takes two arguments as input.

The first is the hash generated by the hash generator box. The second is the modified equation received from the Nonce-box. This box works in tandem with the next box, the ZKPV box, but we will briefly highlight the roles of this box here.

The hash box stripes the hash from the final equation, thereby retaining only the true RHS and the noise (in an ideal condition). What do we mean by striping?

We fetch the calculated hash from the hash generator and strip/eliminate/subtract that portion from the RHS of the equation, thereby eliminated the $h(m)$ factor in the equations.

Once that is completed, the final modified equation is passed to the next box for verification.

At this stage, there are three possibilities.

- a) The first possibility is where the hash is correctly striped out from the equation (considering that the equation is also correct)

Final equation: $LHS = (True\ RHS + noise)$

- b) The second possibility is where the hash is again striped out correctly from the equation, but the equation consists of the garbage that was appended in the A-box (indicative of authentication failure).

Final equation: $LHS = (True\ RHS + noise) || garbage-A-Box$

- c) The third possibility is the case where the transaction was modified, because of which $h(m)$ at the miner's end is not the same as $h(m)$ sent by the sender. During the striping, the process may stripe some parts of the $(True\ RHS + noise)$ or leave some parts of the original $h(m)$ in the equation itself (for an ideal equation). In case of an equation like $\langle 2 \rangle$, incorrect striping will still lead to some garbage remaining in the RHS or no valid $(True\ RHS + noise)$ remaining in the RHS. All of this we will consider as one case .

Final equation: $LHS = (True\ RHS + noise) || garbage$

OR

Final equation: $LHS = garbage$

6. Zero Knowledge Proof Verification Box

The last and the most crucial aspect of the $H^{-1}()$ function is the ZKPV black box. For now, it is claimed to be a black box because we do not exactly know the ZKP that will be used in this black box as yet. This is the determining entity of the entire $H^{-1}()$ function, and because this is a black box now, we imply that $H^{-1}()$ is also a black box to some capacity.

This box receives the final stripped equation as one of the mandatory parameters. It then uses certain zero-knowledge proofs to verify if the equation received is the same as the initial equation generated by the client (the equation with the true RHS +noise).

If it is, then it automatically validates all the boxes in the chain, thereby assuring the miner that this DS is valid and the transaction has not been tampered with.

If the equation does not match with the initial equation generated by the client, then the ZPKV box uses certain other mechanisms to identify which of the three possibilities, discussed one page before, has occurred. Based on the results it generates, it appropriately generates an **F0** or an **F2** error.

For now, our ZPKV only takes the final striped equation. There could be several other latent variables/parameters we may need to pass when we develop the ZPKV fully. They will have to be accounted for, when implementing the ZPKV box.

To mathematically suggest,

ZPKV(LHS= some-RHS) = True , if equation is correct
= False, if equation is invalid

Before, we dive into all the possible results we may obtain from the two hash verification boxes and the ZPKV boxes, we will take the three possibilities suggested previously, and highlight what error my ZPKV box will provide in that scenario-

CASE 1: Final equation: LHS = (True RHS + noise)

In this case, the ZPKV function/box generates a True result, which validates the transaction. There will be no error flagged in this case.

CASE 2: Final equation: LHS = (True RHS + noise) || garbage-A-Box

In this case, the ZPKV function/box will clearly generate a False result. Upon further inspections on what is causing the false generation, the indication of the garbage by the A-box will directly imply that the authenticity of the message has failed. It will flag to the miner that this transaction constitutes an **F0 (authentication)** error.

CASE 3: **Final equation: LHS = garbage**

This case will also generate a False result. Upon inspection, if it is revealed that a larger/smaller chunk of the RHS was scraped by the new hash, then it will obviously indicate that the new hash is incorrect. This will result in the ZPKV box flagging an **F2 (integrity)** error.

The first equation of case 3 will also behave identically as the one stated, as this mechanism checks for increased/decreased scraping by the new computed hash.

Before, we jump into the entire functioning of the two Hash verification functions, let us quickly revisit what these error flags are

1. **F0 error** – The F0 error is an error flag that indicates that there is a violation in the authentication of the message. It indicates that the current transaction could be the result of impersonation, MITM etc.
2. **F1 error** -- The F1 error is an error flag that indicates that there is a violation in the validity of the message. It indicates that the current transaction could be the result of a replay attack etc.
3. **F2 error** – The F2 error is an error flag that indicates that there is a violation in the integrity of the message. We shall believe here that the hash generator on the miner's end is perfect and without flaw. Signalling of this flag indicates that the transaction details have been tampered (possibly MITM).

All these errors are mutually exclusive on purpose. That is, they are triggered only by flaws in the property they are trying to protect. There is no contention that the miner needs to receive only one of the F_i errors. Whatever is the violation, that error is generated.

If all three entities $h(m)$, i and n are tampered with, then all three flags F2, F0 and F1 are triggered.

The use of these flags helps alert the miner of potential malicious entities lurking in the network. Before we dive deeper into the new roles assigned to miners based off these error flags, we will understand the full -working of the Hash verification boxes (two) and the ZPKV box.

The equation absolved of the nonce on the valid end is propagated to the Hash box, where the hash is striped. That is then forwarded to the ZPKV box where the validity is checked. If it is true, the transaction is validated and the next transaction is taken upon for checking.

If the valid box chain fails, then the invalid box chain is checked. If the invalid box chain succeeds, then it indicates that validity is the sole case of failure, so **F1** error is generated and the transaction is not processed any further.

If both the valid and the invalid chain boxes fail, then it clearly indicates that the error is not with the validity of the message (F1 error is never generated in this case). The valid equation chain is considered and the ZPKV verifies which of the two cases, CASE 2 or CASE 3 is violated. Depending on the case, the appropriate error flag is flagged (**F0 or F1**) and the transaction is considered invalid.

Whilst it is true that the use of flags increases the computational overhead of the miner, it still helps identify what are the potential flaws in the network, so that malicious entities/attacks can be thwarted.

NEW MINER RESPONSIBILITIES

Until previously, the role of miner was solely to validate transactions and mine new blocks. In this protocol/approach, we provide additional responsibilities to the miner with appropriate punishment and rewards.

As explained previously, the miners are now aware of why a certain transaction failed in validation. They can use the metadata generated to comprehensively understand if there are certain malicious entities in the blockchain network.

The miners can then use consensus algorithms and other networking protocols to identify potential miscreants, based on all the error data collected, and eliminate them from the blockchain network thereby sanitizing the network. The lesser the malicious entities in the network, the lesser the time a miner has to spend on validating an invalid transaction, and thereby higher is the probability of the miner generating a block that will join the main network.

Miners will be rewarded if they correctly remove malicious entities from the network. Miners will be severely penalized, if they act out of malice and eliminate legitimate entities from the network. Their reward will again be defined via BTC (supposed). Their penalization can include losing the BTC earned, or being deposed from the role of a miner.

The higher penalties will ensure that miners do not unnecessarily remove legitimate entities from the network. The added computational overhead will be worth the burden if they can remove even one malicious entity from the network.

It is true that there must be some form of accountability that must be established so a particular client knows why they have been eliminated from the network. Similarly, there must be a resolution made on what happens to the Bitcoin (BTC) present in the wallet of the eliminated entity.

These ideas can constitute a paper of its own. It will not be discussed in great lengths here; this is only a proposed idea of protection.

CONCLUSION

In this paper, we discussed a new novel scheme that uses the LWE quantum-resistant algorithm as a scheme to develop a Digital Signature Protocol. It uses the concepts of appending noise, to append crucial parameters essential for a Digital Signature scheme.

The paper discusses these parameters in great length and highlights how the Digital Signature is constructed at the client end.

It then discusses how it is deciphered and validated at the miner's end. It further goes on to propose an idea of using miners as the network sanitizers.

Overall, this paper discusses the complete Digital signature scheme, from the time the private key is generated to the time the DS is considered valid/invalid.