Yobicash – technical paper

Christian Nyumbayire

Description

Yobicash is a peer-to-peer cryptocurrency for sending and receiving encrypted data. Users create tokens by consuming memory, and the conversion rate between the memory consumed and the tokens created is 1 on 1. Encrypted data is sent by converting tokens into data, so if a user has 10 tokens, he can send 10 bytes of data to the other user. The size of the data sent does not count of the increase given by the encryption process.

To remove scalability issues, a DAG (Directed Acyclic Graph) is used, making of Yobicash a "blockchain-less" cryptocurrency [1], while Iota is a working example of a DAG-based cryptocurrency [2].

To disincentivize doublespending, it is necessary to ensure that nodes have no incentives to omit or censor transactions. Different measures are taken to ensure it. Nodes are paid by users to receive new transactions and compete in order to have the user fees, and when users spend their outputs, the data present in them (if any) is pruned. This is meant to incentivize liveness and ensure the economic sustainability of the network.

To remove the incentives for censorship and other non-cooperative behaviors as selfish omissions (e.g.: selfish mining in Bitcoin), transactions are anonymous and untraceable and data is encrypted.

Coinbases are just a type of inputs in the transactions. To ensure there is no inflation apart from the one introduced by coinbases, every transaction has a height, which has to be the longest one among the input transactions heights. Transactions with height 0 are those building coinbases but having no inputs, so the longest height has to bring to the oldest coinbases (implicitly) inputted by a transaction. Transaction DAGs are checked starting from the inputs and going one height backward. Doing this for every incoming or requested transaction ensures an implicit consensus on which transactions are valid and which are not.

Cryptography

Yobicash makes use of standard cryptographic primitives and constructions in order to reach its ends. The pros are that they are well understood and simpler on respect of new constructions, the

cons are that they are less powerful, and in certain cases efficient, than some new constructions and could not resist quantum attacks.

Yobicash is aimed to be a simple, dumb system meant to just be used to exchange value and data. In the future, the primitives used could be improved, but it will depend on the incentives and the governance structure how and why.

Collision-Resistant Hashing Algorithm

A collision-resistant hashing algorithm is a map H from a binary string of varied length msg to a binary string of fixed length digest where the probability to extract msg from digest is negligible and the probability of finding two strings msg and msg' where H(msg) == H(msg') is negligible.

```
msg = binary string
length = uint32
digest = binary string of length length

H(msg) = digest
```

The hashing algorithm used is sha512. This hashing algorithms is tought to be quantum-resistant, which means resistant to quantum attacks.

Memory Hard Function

A memory hard function is an algorithm which maps a binary string seed and a set of parameters params to a binary string digest of fixed length. The function uses a fixed amount of memory which depends on the input parameters and, in certain constructions, also on the input seed. For completeness, the notation indicates also the used memory mem.

```
seed = binary string
params = [ binary string | uint32 | biguint ]
length = uint32
digest = binary string of length length
mem = uint32

MHF(seed, params) = (digest, mem)
```

The memory hard function algorithm used is Balloon [3]. The internal hashing algorithm used is sha512.

Discrete Logarithm Problem

The discrete logarithm problem states that given a group G of order q (not per se prime), a generator g of G and an element x of G and y = g^x is computationally hard to find x given only y and g. x is called the discrete logarithm of y to the base g.

Different cryptographic constructions as RSA and elliptic curve cryptography are based on the discrete logarithm problem.

Diffie-Hellman Problem

The Diffie-Hellman Problem is an extension of the Discrete Logarithm Problem stating that given a group G of order q, and two elements x and y of G, it is computationally hard to find y = g^(x*y) given only y and g.

The Diffie-Hellman Problem is at the foundation of different cryptographic algorithms as the Elliptic Curve Diffie-Hellman (ECDH) key exchange algorithm, used in encrypted communication protocols as HTTPS to build anonymously shared encryption keys in unsafe communication channels.

Symmetric/Secret Key Encryption Algorithm

A symmetric (or secret key) encryption algorithm is a cryptographic construction encrypt that map a plaintext plain and a binary string key to a cyphertext cyph. The decryption of the string is obtained by applying the encryption algorithm encrypt to the cyphertext cyph and the key key. The algorithm has to ensure that, given the cyphertext only, it is computationally hard to retrieve the plaintext or the key.

key = gen(1^{λ})
cyph = encrypt(plain, key)
plain = encrypt(cyph, key)

The symmetric key encryption algorithm used is AES-256. AES-256 is tought to be quantum-resistant, which means resistant to quantum attacks.

Message Authentication Code (MAC) Algorithm

A message authentication code (MAC) algorithm is a cryptographic construction MAC that maps a binary string of varied length msg to a fixed length string tag. The mapping is univocal with a high probability. The tag produced is used to authenticate the string msg.

Generation:

tag = MAC(msg)

Message authentication:

given (msg, tag)
tag' = MAC(msg)
fail if tag' != tag

The message authentication code used is HMAC, which consists of the use of a collision-resistant hashing algorithm H as MAC. The hashing algorithm used is sha512.

Key Derivation Function (KDF)

A key derivation function is a map KDF from some public parameters params to a secret key key used in a symmetric encryption algorithm.

```
KDF(params) = key
```

The function is used to allow all the end points of an encrypted communication channel to derive the symmetric key without having to share it.

The key derivation function used is KDF2.

Elliptic Curve Integrated Encryption (ECIES) Algorithm

An elliptic curve integrated encryption algorithm is an algorithm for authenticated encryption. The sender and the receiver exchange their public keys g^x and g^y, which are used to generate a symmetric encryption key key via a key derivation function KDF. The encrypted message is authenticated with a message authentication code MAC.

Receiver Setup:

```
sk is a natural number in [0, q−1]
pk = sk*G
```

Sender Encryption:

```
given receiver pk

r is a natural number in [0,q−1]
R = r*G
P = r*pk (with P not the point on infinity)

fail if P is the point on infinity (P + q == q)

S = p_x
key = KDF(S)
cyph = encrypt(msg, key)
tag = MAC(key, cyph)
```

Receiver Decryption:

```
given sender (R, cyph, tag)
```

```
(p_x, p_y) = P = R*sk
S = p_x
key = KDF(S)
tag' = MAC(key, cyph)

fail if tag' != tag

msg = decrypt(m, k)
```

The ECIES algorithm used has ed25519 as its elliptic curve, KDF2 as key derivation function, AES-256 as the symmetric encryption algorithm and HMAC as message authentication algorithm.

Interactive or Non-Interactive Proof Scheme (IP)

An interactive proof (IP) scheme on an relation R (x, w) of a language L (X, W) of sets X and W is a pair of interactive (non-interactive) algorithms (Prove, Verify) where Prove takes as input a pair (x, w) of (X, W), namely the instance and the witness, and Verify takes as input the instance x and outputs {0, 1}, where 0 means that Verify(x) does not accept x as the instance of a relation R of the language L, while 1 means that Verify(x) accepts x as an instance of a relation R of the language L.

Interactive or Non-Interactive Zero Knowledge Proof Scheme (ZKP)

A zero knowledge proof (ZKP) is an interactive (non-interactive) proof system (Prove, Verify) for a relation R (x, w) of a language L (X, W) where is computationally unfeasible to extract the witness w of the relation R only knowing the instance x or running Verify.

Sigma (Σ) Protocol

A sigma (Σ) protocol is an interactive, 3-moves, zero knowledge proof (Prove, Verify) of an NP language. In a sigma protocol, in the first move Prove outputs an instance x, in the second Verify outputs a challenge c, in the last move Prove reply with a response r that may make Verify accept or reject.

```
x = Prove(x, w)
c = Verify(x)
r = Prove(c, w)
{0, 1} = Verify(x, c, r)
```

Sigma protocols can be composed, obtaining Or Sigma protocols, where the composed protocol accepts only if at least one of the protocols accept and And Sigma protocols, where the composed protocol accepts only if all the protocols accept.

Schnorr Protocol

The Shnorr protocol is a Σ protocol (Prove, Verify) of a relation R (x, w) of the language L (X, W) where, given a group G of order q, a generator g of G, a natural number w in [0, q-1] and an element x of the group G where x = g^w, X is the set of all the possible elements x, W is the set of all the possible elements w and R is a pair (x, w) where x = g^w. The protocol is zero-knowledge assuming the hardness of the Discrete Logarithm Problem.

Setup:

```
q is prime
G is a group of order q
g is a generator of G
```

1st move, Prover:

```
R = (x, w) where x = g^w
u = random integer in [0, q-1]
a = g^u
```

2nd move, Verifier:

```
receives (x, a)

c = random integer in [0, q-1]
```

3rd move, Prover:

```
given c

r = u + w*c
```

final, Verify:

```
g^r == a*x^c -> g^(u + w*c) == g^u * (g^w)^c = g^(u + w*c)
```

The protocol is the first historical example of a sigma protocol, and of a zero-knowledge system for anonymous credential [4].

Algorithms and Data Structures

Yobicash main data structure is the transaction. The lack of blocks and blockchains has the benefit of making the whole construction simpler, but it opens to new difficulties, the first that it is still not clear what is the best way to build an incentive-compatible graph of transactions without using a blockchain. Yobicash try to achieve incentive-compatibility by enforcing censorship-resistance via encryption and the use of anonymous credentials, by

removing a fixed supply curve but still requiring that the resources used are provably the product of a required work (which we could not call a full-fletched proof-of-work, in absence of a changing difficulty), by letting supply and demand define money creation incentives and difficulty, and by letting nodes for user transaction fees.

Coinbase

A coinbase is the data structure used for conveying money creation. It is just the run of a memory hard function. The seed is the msg input in MHF. The memory spended by running the funcion is the spendable amount, while the digest produced by the function is the “coins”.

```
params = mhf_params
seed = mhf_seed
coins = mhf_digest
amount = mhf_memory

coinbase = (params, seed, coins, amount)
```

Syntactic Invariants

```
(digest, mem) = MHF(coinbase.params, coinbase.seed)
digest == coinbase.coins
mem == coinbase.amount
```

Input

An input is a link to the output coins of one transaction sent to the creator of the transaction where they are included. An input has to authenticate anonymously the creator as the receiver of the output coins by retrieving the 3rd move of the Schnorr protocol instantiated in the linked output coins.
The linked output is located by the id of the transaction where it has been included and its index in the transaction outputs.
To ease the validation of the transaction, the linked output coins amount is repeated in the input.

```
tx_id = hash_digest
tx_height = uint32
output_idx = uint32
output_amount = uint32
c = schnorr_c
r = schnorr_r

input = (tx_id, tx_height, output_idx, output_amount, c, r)
```

Output

```
amount = uint32
pk = ecies_pk
checksum = hash_digest
```

```
r = ecies_R
cyph = ecies_cyph
tag = ecies_tag
g = schnorr_g
x = schnorr_x
a = schnorr_a

output = (amount, pk[, checksum, r, cyph, tag], g, x, a)
```

Syntactic Invariants

```
if output.r and output.cyph and output.tag and output.checkum:
     R = output.r
     cyph = output.cyph
     tag = output.tag

     size(cyph) == output.amount

     if ecies_sk is known:
          ECIES_decrypt(R, cyph, tag) does not fail
          H(ECIES_decrypt(R, cyph, tag)) == output.checksum
```

Transaction

```
id = hash_digest
height = uint32
coinbases_len = uint32
coinbases = [coinbase]
inputs_len = uint32
outputs_len = uint32

tx = (id, height, coinbases_len, coinbases, inputs_len, inputs,
outputs_len, outputs)
```

I/O Methods

```
// given a tx_id, return 0 if it has not been found, 1 otherwise
io_lookup_tx(tx_id) = {0, 1}

// given a tx_id, return the tx if it has been found, error
otherwise
io_get_tx(tx_id) = tx | error

// given a tx, return (), the bottom type or null/void in some
// languages, if it has been put, error otherwise
io_put_tx(tx) = () | error

// given a tx_id, return (), the bottom type or null/void in some
// languages, if it has been pruned, error otherwise
// pruning consist of removing the encrypted data
// data metadata as its checksum, tag and pk are left in case
// the encrypted data is still retrievable and need to be
// verifiable off-dag
io_prune_tx(tx) = () | error
```

Syntactic Invariants

```
id == H(tx.height||tx.coinbases_len||tx.coinbases||tx.inputs_len||
tx.inputs||tx.outputs_len||tx.outputs)

coinbases = tx.coinbases
len(coinbases) == tx.coinbases_len
coinbases are valid

inputs = tx.inputs
len(inputs) == tx.inputs_len
inputs are valid

outputs = tx.outputs
len(outputs) == tx.outputs_len
outputs are valid

sum(coinbases => coinbase.amount) + sum(inputs => input.amount) ==
sum(outputs => output.amount)
```

Semantic Invariants

q = order of the group G used in the Schnorr protocol

```
max_height = 0

inputs_checksum   =   H(tx.height||tx.coinbases_len||tx.coinbases||
tx.inputs_len||tx.outputs_len||tx.outputs)

for idx in 0..tx.inputs_len:
     input = tx.inputs[idx]

     io_lookup_tx(input.tx_id) == 1

     inputs = tx.inputs

     input_tx = io_get_tx(inputs.tx_id)
     input.idx <= input_tx.outputs_len - 1

     input_output = input_tx.outputs[input.tx_id]
     input_output.amount == input.amount

     input_checksum = H(inputs_checksum||idx)
     // c binds the input to all the rest of the transaction
     // this measure ensure there is no way to steal the inputs
     // without knowing the witness w of the Schnorr protocol
     // As far the hashing function H is pre-image resistant and
     // collision resistant and the rest of the transaction is
     // not deterministic (thanks to the witness w and the fields
     // in outputs and the ECIES secret key sk), it can be used as
a valid
     // source of entropy
     input.c == biguint(input_checksum) mod q
     output.g^input.r == output.a*output.x^input.c

     max_height = max(input_tx.height, max_height)

max_height == tx.height
```

Security

The security of the transaction is ensured as far as the Schnorr protocol variables u and w and the ECIES secret key sk are chosen randomly. The only way to ensure it is to compel the user to use good sources of entropy. This is not always so easy in all the scenarios (e.g.: virtual machines). This problem is common to all the cryptographical constructions using standard random generators as a source of randomness. This will be one of the points to inspect.

Protocols

pull_tx

```
[user]: (“pull_tx”, pk, tx_id)

[node]: (R, cyph, tag) | internal_error
     cyph = ecies_encrypt(plaintext, key)
     plaintext = tx | error
```

push_tx

```
[user]: (“push_tx_info”, pk, size, fees)

[node]: (R, cyph, tag) | internal_error
     cyph = ecies_encrypt(plaintext, key)
     plaintext = (ok, pk) | no | error

[user]: (R, cyph, tag) | internal_error
     cyph = ecies_encrypt(plaintext, key)
     plaintext = (tx, pk) | no | error

[node]: (R, cyph, tag) | internal_error
     cyph = ecies_encrypt(plaintext, key)
     plaintext = ok | error
```

Bid

A bid is the set of prices per byte offered by a node to the network users for introducing their new transactions in the network. This has to be known in advance in order to reduce the risk of unintended double-spendings by the user, who, in the attempt to introduce her transactions in the network, may end up sending different versions of the transaction but spending the same outputs.

There can be different prices for different ranges of data. This allows having non-linear pricings and overpriced (underpriced) transactions with excessive (marginal) space requirements.

Nodes include their output Schnorr protocol anonymous credentials in the bid, in the case the user will accept their offer.

```
bid = (g, x, a, [(from_size, to_size, price)])
```

Syntactic Invariants

Size ranges have no intersections.

Protocols

get_bid

```
[user]: (“bid”, pk)

[node]: (R, cyph, tag) | internal_error
     cyph = ecies_encrypt(plaintext, key)
     plaintext = bid | error
```

Wallet

Anonymous credentials make impossible to have more than a wallet (there are no public keys) but at the same time to have a proper wallet (there are no public keys). So a wallet is just the set of spent outputs (inputs and their outputs with their Schnorr protocol witness and location) and unspent outputs (outputs with their Schnorr protocol witness and location) by the user. The total balance is the sum of the input coins which are not been spent.

w = schnorr_w
```
unspent_outputs = [(w, tx_id, idx, output)]
spent_outputs = [(w, tx_id, idx, output, input)]

wallet = (unspent_outputs_len, unspent_outputs, spent_outputs_len,
spent_outputs, balance)
```

Syntactic Invariants

len(unspent_outputs) == unspent_outputs_len
len(spent_outputs) == spent_outputs_len

sum(unspent_outputs => unspent_output.amount) – sum(spent_outputs => spent_output.amount) == balance

for unspent_output in unspent_outputs:
    unspent_output is syntactically valid

for spent_output in spent_outputs:
    spent_output is syntactically valid
    spent_output.data is absent // it has been pruned when it was spent

Semantic Invariants

g = generator of the group G of order q used in the Schnorr protocol

for unspent_output in unspent_outputs:
    tx_id = unspent_output.tx_id

```
        io_lookup_tx(tx_id) == 1

        tx = io_get_tx(tx_id)
        tx is syntactically valid

        idx = unspent_output.idx
        idx <= tx.outputs_len-1

        output = unspent_output.output
        w == g^output.x


for spent_output in spent_outputs:
        tx_id = spent_output.tx_id

        io_lookup_tx(tx_id) == 1

        tx = io_get_tx(tx_id)
        tx is syntactically valid
        tx is semantically valid // max regression level = 1

        idx = spent_output.idx
        idx <= tx.outputs_len-1

        output = spent_output.output
        w == g^output.x
```

Bibliography

[1]: X. Boyen, C. Carr, and T. Haines, *Blockchain-Free Cryptocurrencies: A Framework for Truly Decentralised Fast Transactions*, 2016
[2]: Serguei Popov, *The tangle*, 2016
[3]: D. Boneh, H. Corrigan-Gibbs, and S. Schechter, *Balloon Hashing: A Memory-Hard Function Providing Provable Protection Against Sequential Attacks*, 2017
[4]: C. P. Schnorr, *Efficient identification and signatures for smart cards*, in G Brassard, ed. Advances in Cryptology, 1990