Re-implementation of the Picnic signaturescheme in Python

Thorsten Knoll

info@thorstenknoll.de

March, 2019

I. Introduction

Post-quantum cryptography, NIST and Picnic

Quantum computers are experiencing fast development and seem to be available within a timeframe of the next few decades. One of their properties will be to break huge parts of modern cryptography. Especially the discrete logarithm and prime-factorisation loose their trapdoor funcionality in regards to the efficient quantum algorithms from Grover and Shor. Therefore the need for new cryptographic algorithms arises, beeing save in regards to the availability of quantum computers. This field of research goes with the name "Post-quantum cryptography" (PQC). The american National Institute of Standards and Technology (NIST) called out a challenge to find the next PQC standards. This challenge is in round two of three at the time of writing this document. 69 submissions from round one were reduced to 26 candidates in round two of the challenge. These 26 candidates got announced by NIST not long ago at the end of january 2019. One of the submissions surviving the first round is the Picnic signaturescheme.

NIST PQC page: `https://csrc.nist.gov/Projects/Post-Quantum-Cryptography`

Evaluation and reviews

69 submissions is quite a huge amount in terms of reviewing and evaluating them. Additionaly PQC is a new field of research with a history not much longer than a decade. Cryptographers (and -analysts) are working intense on creating, evaluating, reviewing and breaking PQC algorithms. The state of PQC is that every litte step helps towards the goal of having secure PCQ algorithms. At RheinMain University of Applied Sciences (HSRM) the Master Students in Computer Science found together in a research course to participate in this process. This re-implementation is part of this efford.

Goals: Learning Picnic (and LowMC)

The sole purpose of this re-implementation is learning and understanding the Picnic algorithm and it's underlying zero-knowledge proof system. The goals are to provide an "easy to read and understand" codebase in a high-level language (Python) and make it a little easier to follow and learn the designprinciples of Picnic. The execution of Python-Picnic is awefull slow compared to the reference implementations in C. It is very unlikely that this will ever reach a usable state for production. Additionaly the code is not reviewed by anyone but the author and is surely not secure for productive use. But it may help to understand Picnic.

On the way to understand Picnic, another algorithm must be understood too. That is the LowMC blockcipher. There are two main parts in Picnic, where LowMC plays a major role. Firstly the derivation of the public key is a LowMC encryption of the Picnic private key. Secondly each Picnic zero-knowledge round is a modified LowMC encryption, fitted into a Multi-Party-Computation (MPC) scheme. In the following description of the implementations we start with LowMC for that reason.

Reference documents and implementations by the Picnic Team

The original publications and reference implementations (in C) are available at:

Microsoft Picnic projectpage: `https://www.microsoft.com/en-us/research/project/picnic/`
Microsoft Picnic Github: `https://github.com/Microsoft/Picnic`

Re-implementations in Python

These re-implementations in Python follow the original MIT Licenses and are public available at:

LowMC in Python: `https://github.com/ThorKn/Python-LowMC`
Picnic in Python: `https://github.com/ThorKn/Python-Picnic`

Installation and usage instructions are given inside the projects `README` files. This `PDF` can be found inside the `Python-Picnic` project in the `docs` folder.

II. LowMC

Idea and overview

Picnic makes intense use of the LowMC algorithm. Therefore we start with LowMC before hopping over to Picnic. A standalone LowMC Python-implementation got build as a starting point to understand Picnic. That is why there is a own Github project under the name `"Python-LowMC"`.

LowMC is a blockcipher in a roundbased construction scheme like many other blockciphers. The name is an abbreviation for "Low multiplicative complexity". XOR in GF2 is a linear operation (ADD), while the multiplication (AND) in GF2 is a non-linear operation (Figure 1). LowMC tries to keep the count of AND operations as low as possible while still maintaining a given security level (L1, L3, L5). Additionaly LowMC is also designed to keep the AND-depth low. That means most AND operations could be done in parallel.

A	B	XOR(A,B)	AND(A,B)
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Figure 1: Linear and non-linear operations

Figure 2 shows the roundbased scheme of LowMC. Only the sbox part of the algorithm uses multiplications (ANDs). The other parts strictly contain only linear operations. From a mathematical point of view the XOR is a bijective function and the multiplications (ANDs) in GF2 are not bijective. Therefore the sboxes are the only not bijective part in LowMC. That is the reason for choosing LowMC in Picnic and it keeps the signature lengths smaller than with other blockciphers. We'll see how this is achieved with the Picnic implementation later on.

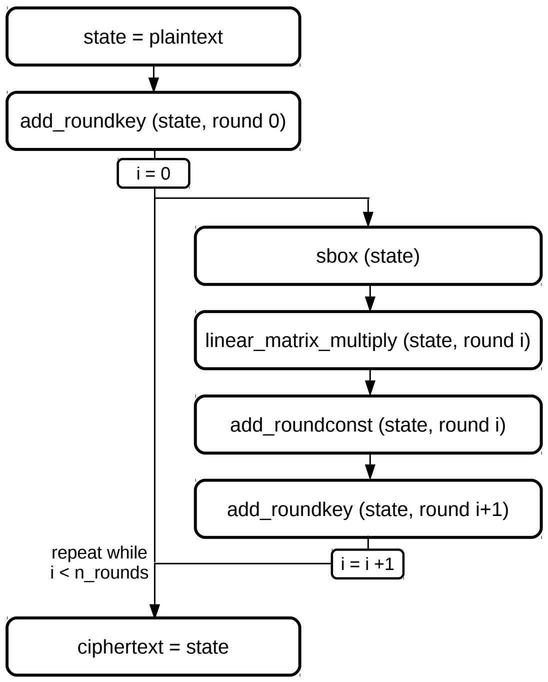
state = plaintext
add_roundkey (state, round 0)
i = 0
sbox (state)
linear_matrix_multiply (state, round i)
add_roundconst (state, round i)
add_roundkey (state, round i+1)
i = i +1
repeat while
i < n_rounds
ciphertext = state

Figure 2: LowMC scheme

Pre-calculated constants

The python program `generator.py` generates the files for all security levels with the following pre-calculated constants:

- Linear layer matrices
- Round constants
- Roundkey matrices

The generation of the constants-files is not mandatory for usage as the project contains them "ready to use".

Private functions

`__apply_sbox()`

Sboxes are the only parts in LowMC, that contain multiplications in GF2 (ANDs). A fixed number of sboxes is applied to the state. Each sbox substitutes 3 Bit of the original state through a fixed substitution scheme. Let a, b and c be three bits in the state. Then a', b' and c' get computed by:

$$a' = a \oplus b \cdot c$$
$$b' = a \oplus b \oplus a \cdot c$$
$$c' = a \oplus b \oplus c \oplus a \cdot b$$

Where $\oplus$ is XOR and $\cdot$ is AND. So each 3-Bit sbox contains exactly 3 multiplications (ANDs). The security level (L1, L3, L5) defines the number of 3-Bit sboxes (n) per LowMC-round as shown in figure 3. The surplus Bits in the state get no substitution, shown on the right of the figure. The total number of multiplications for a complete encryption in LowMC sums up to $3 * n * rounds$. In example for security level L1 this calculates to $3 * 10 * 20 = 600$ ANDs.

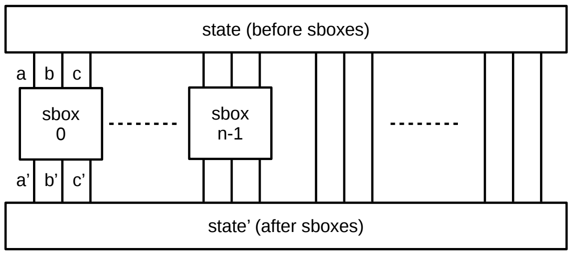

Figure 3: Sboxes per LowMC round

The function applies the sboxes to a state in memory and has no parameters and no returns. The actual state is stored in the private variable `self.__state`. The sboxes are stored as a list (you can imagine it as a lookup-table) in the private variable `self.__sbox`.

`__multiply_with_lin_mat(round)`

The state gets matrix multiplied with the constant and pre-calculated linear matrix. This contains only linear operations.

`Add roundconstant`

This needs no seperate function. It is a one-line operation with XOR on the state and therefore contains only linear operations.

`__key_addition(round)`

For XOR'ing the roundkey to the state the roundkey must be derived first. This is done by a matrix multiplication of the private key and the constant, pre-calculated roundkey-matrix. Again this only contains linear operations.

`Decryption functions`

The decryption works pretty much the whole encryption way backwards. For the matrix multiplications their inverse matrices are needed. They get calculated within the constructor of the LowMC class and stored in seperate, private variables. The neccessary functions are named like the ones for encryption but with a `_inv` appended to their names. The same rule applies for the names of their private variables.

Public functions (API)

`LowMC(Security level) - Constructor`

Constrcuts an object of LowMC with the parameters regarding to the security level. The following security levels are available and shall be given to the constructor as strings: "picnic-L1", "picnic-L3" and "picnic-L5". The fitting file with the constants must be in the project directory and gets read (see Pre-calculated constants). The constants from the file get stored in the private variables `self.__lin_layer`, `self.__round_consts` and `self.__round_key_mats`.

`generate_priv_key()`

Generates a private key of the length specified within the security level. The private key is stored in the private variable `self.__priv_key`. The CSPRNG from the python package `os` is used (`os.urandom(bytelength)`).

`set_priv_key(priv_key)`

Instead of generating the private key, it can also be set by giving a bytearry to this function. The bytearray must match the specified keylength from the security level.

`encrypt(plaintext)`

Encrypts a plaintext and returns the ciphertext. The plaintext length must match the specified blocksize length (security level) and must be given as a bytearray to the function. The ciphertext is returned as a bytearray of the same length. Before using this function a private key must be set (or generated).

`decrypt(ciphertext)`

Decrypts a ciphtertext and returns the plaintext. The ciphertext length must match the specified blocksize length (security level) and must be given as a bytearray to the function. The plaintext is returned as a bytearray of the same length. Before using this function a private key must be set (or generated).

Testvectors

The repository contains the python file `test_lowmc.py`. One can simply run this and nine different testvectors get executed on the implementation. Three vectors for each security level. This testfile is also a good starting point to see how the implementation can be used.

Prerequisites

The code is tested with Ubuntu 16.04 LTS and Python3.6. The package "BitVector" for python is required. It is recommended to use a virtual environment for Python, like `virtualenv`. In (very) short lines:

```
virtualenv -p /usr/bin/python3.6 myvenv
source /path_to_myvenv/bin/activate
<myvenv>pip install BitVector
<myvenv>python test_lowmc.py
```

III. Picnic

Components

Picnic is a Post-quantum signature scheme that does not rely on hard number theoretical problems like discrete logarithm or prime-factorization. Instead Picnice embedds symmetric cryptographic primitves into a zero-knowledge proof system. The components of Picnic are:

- A blockcipher (LowMC)
- A Hashfunktion (SHA3-SHAKE)
- A Zero-knowledge proof system (ZKB++)

LowMC is discussed earlier in this document. It is a parametric blockcipher algorithm that simulates a gatebased (XOR, AND in GF2) electrical circuit from input to output with low AND-gate counts. The Python-LowMC implementation will be used for Picnic. SHA3-SHAKE will not be discussed in detail in this document. It is a NIST-standardized Hashalgorithm, based on the Spongeconstruction with an arbitrary output length. In Picnic SHA3-SHAKE is used as a Hashfunction as well as a Key-Derivation-Function (KDF). Python-Picnic uses the Python library `hashlib` whereever SHA3-SHAKE is needed

That leaves the zero-knowledge proof system ZKB++ to explain. The following descriptions are on a very abstract level and will not explain every detail. Instead we'll focus on getting a broad overview of the functionality to learn the key points of the implementation. Most of the following examples are based on the documentations and presentations of the Picnic research team [1].

Proof of knowledge: The Σ - Protocol

Zero-knowledge proofs are based on the Σ-protocol, but with the twist to transmitt no parts of the underlying secret anywhere in the communication. The basic Σ-Protocol is a communication scheme between two parties as in figure 4. A prover wants to convince a verifier about the knowledge of a secret.

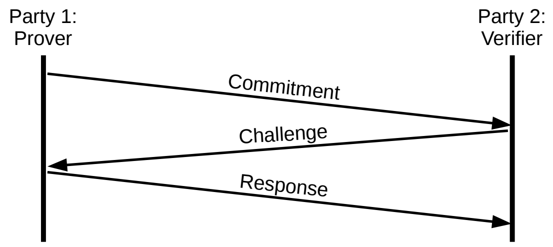

Figure 4: Σ-Protocol scheme

To prove this knowledge, a 3-way communication is held. A classic example for the Σ-Protocol is the Schnorr protocol. It is based on the discrete logarithm. The prover knows a secret x so that $y = g^x$ with g beeing a generator in a cyclic group G_p with the order p. The messages in the Schnorr protocol are:

[1] `https://asiacrypt.iacr.org/2018/files/SLIDES/TUESDAY/Z411/post%20quantum%20signatures%20-%20asiacrypt18v2.pdf`

- Commitment: Prover chooses a random r and commits $t = g^r$ and y.
- Challenge: Verifier chooses a random c and sends it to the prover.
- Response: Prover sends $s = r + cx$.

The verifier accepts, if $ty^c = g^s$. Because:

$$ty^c = g^r y^c = g^r g^{cx} = g^{r+cx} = g^s$$

Inserted into the sequence diagramm above, this communication would look like figure 5.

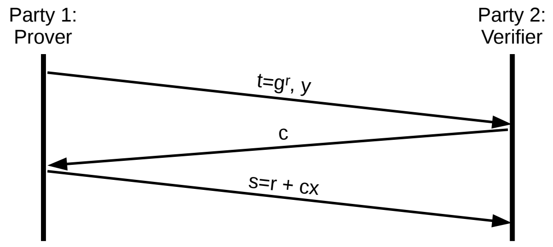

Figure 5: Schnorr protocol

LowMC in ZKB++

In the next step towards ZKB++ we'll look into how LowMC works as the One-way-function inside a Σ-protocol.

Imagine LowMC as a electronic circuit of XOR and AND gates as in the example in figure 6. There are inputs ($x_{1..8}$) and outputs ($y_{1..6}$) from the circuit. The function of this LowMC circuit could be $f_{LowMC}(x) = y$. For given inputs the outputs can be calculated efficiently. For given outputs there is no efficient algorithm to determine the inputs. That is a hard to reverse function.

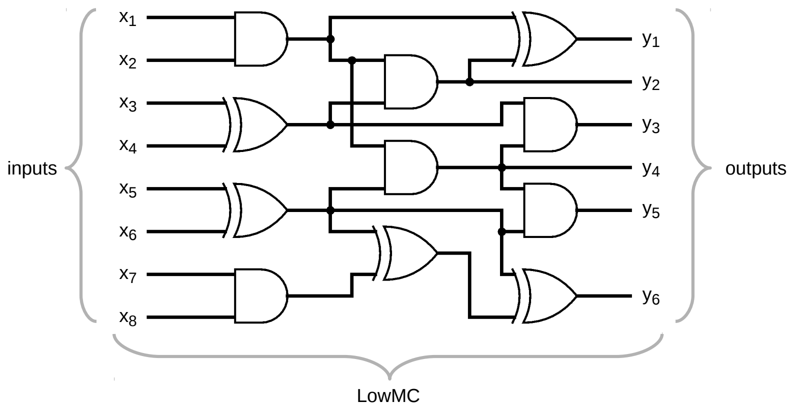

Figure 6: A LowMC circuit as a One-way-function

Thinking in terms of the Σ-protocol we can define the following sentence:

A prover has knowledge of a secret key (inputs x) that computes with the LowMC cicruit (One-way-function) to a public key (outputs y).

MPC and the zero-knowledge proof

The zero-knowledge proof system ZKB++ is based on Multi-Party-Computation (MPC). MPC means that the prover from the Σ-protocol is splitted into different players. For ZKB++ the number of players is fixed to three. To get a grip of how MPC and the zero-knowledge property works, we'll look into an example from Melissa Chase from the Picnic research team (figure 7). She presented this example at RealWorldCrypto 2018 and the talk is public available as a video[2].

A prover knows the secrets a and b. The function f shall be the very simple circuit $c = a \oplus c$, where $\oplus$ is an XOR. Let H be a cryptographic secure Hashfunction (i.e. SHA3-SHAKE). This example is designed for two players, instead of the three players in ZKB++.

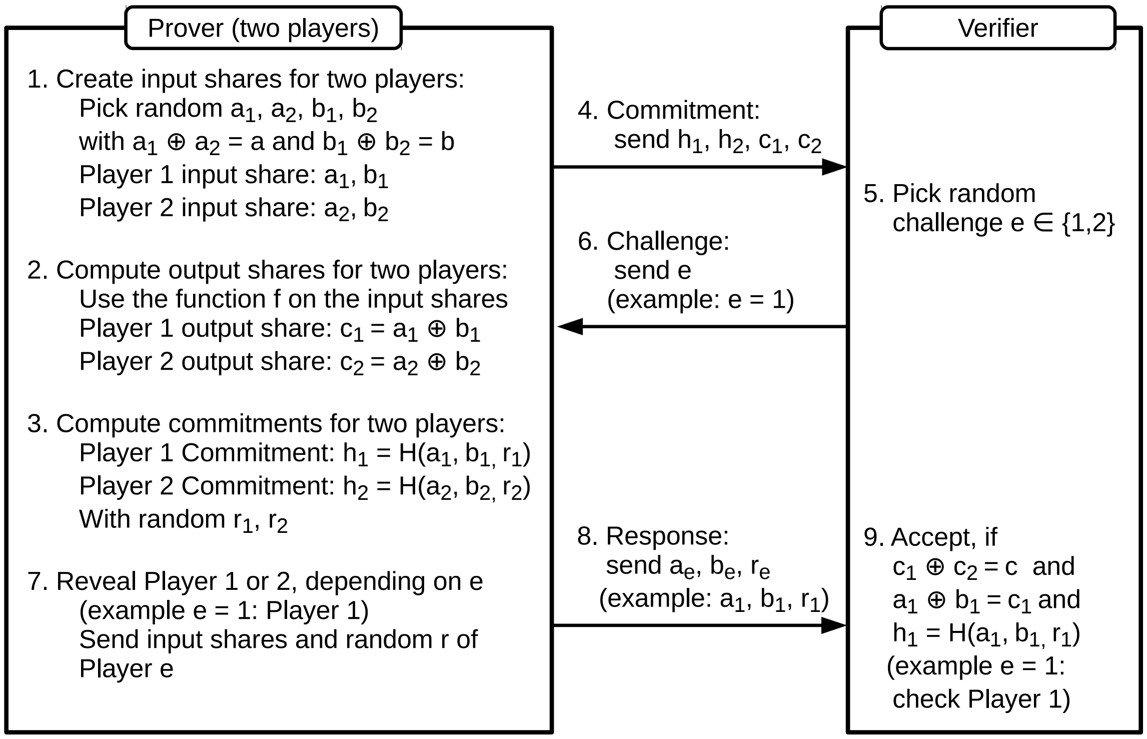
Prover (two players)
Verifier
1. Create input shares for two players: Pick random a1, a2, b1, b2 with a1 ⊕ a2 = a and b1 ⊕ b2 = b
Player 1 input share: a1, b1
Player 2 input share: a2, b2
2. Compute output shares for two players: Use the function f on the input shares
Player 1 output share: c1 = a1 ⊕ b1
Player 2 output share: c2 = a2 ⊕ b2
3. Compute commitments for two players:
Player 1 Commitment: h1 = H(a1, b1, r1)
Player 2 Commitment: h2 = H(a2, b2, r2)
With random r1, r2
7. Reveal Player 1 or 2, depending on e (example e = 1: Player 1) Send input shares and random r of Player e
4. Commitment: send h1, h2, c1, c2
5. Pick random challenge e ∈ {1,2}
6. Challenge: send e (example: e = 1)
8. Response: send ae, be, re (example: a1, b1, r1)
9. Accept, if c1 ⊕ c2 = c and a1 ⊕ b1 = c1 and h1 = H(a1, b1, r1) (example e = 1: check Player 1)

Figure 7: MPC with two players

It is easy to recognize the Σ-protcol in this MPC by the message scheme of "commitment", "challenge" and "response". The differences are that now two players are involved on the prover side. Both have their input- and output shares and the challenge determines which players input shares get revealed. The verifier can then check if the revealed player had a valid input share of the secrets a and b and therefore is convinced about the provers knowledge of the secrets by a probability of 50% (one of two players). This can be repeated n times till the wanted probability (defined by the security level) is reached. The probability calculates to $p = (1/2)^n$. As the function f is assumed to be "hard to reverse", nothing about the secrets a and b got learned. That is the zero-knowledge part of the proof.

[2] `https://www.youtube.com/watch?v=_J9ESIy8D2o`

Some remarks about the differences between this example and the MPC in Picnic need to be noted:

- This works because of $(a_1 \oplus a_2) \oplus (b_1 \oplus b_2) = (a_1 \oplus b_1) \oplus (a_2 \oplus b_2) = c_1 \oplus c_2 = c$
- LowMC has AND gates. This makes dividing the shares more complicated.
- ZKB++ has three players instead of two. The secrets have to be shared by three.
- The cheating probability increases to 2/3. Confidence probability is $p = (1/3)^n$.
- Challenges request one of three players to be revealed with $e \in \{0, 1, 2\}$.
- The lowest security level in Picnic has $n = 219$ MPC rounds.

MPC in the head and Random oracle models

The zero-knowledge proof we've discussed so far is interactive between the prover and the verifier. The communication in the Σ-protocol defines the verifier as a secure source of randomness for the challenges. The prover must not be able to guess in advance what the challenge will be.

For a signaturescheme it is wanted to calculate the proof without the need for an "external" source of randomness. The signatures shall be created and verified on a single machine without the counterpart of the verifier in the communication. This is called "MPC in the head", reduces the proof to a non-interactive version and is used in ZKB++. It wouldn't be a secure idea to let the prover choose the challenges on his own. The prover then could simply pick the challenge that favours him. A proveable secure solution to this is to use a "Random Oracle Model" (ROM) on the provers side to generate the challenges. In ZKB++ this is done with the cryptographic secure hashfunction SHA3-SHAKE. The output of SHA3-SAHKE is the source of randomness under the premise of the ROM and fulfills all the needed properties for secure random numbers.

The concrete implementation looks like this:
The round-commitments contain randomness (input shares) and are calculated and commited before the start of the protocol. To get the challenges, the commitments from each MPC round gets hashed. In the example above (figure 7) this would be the hash of h_1, h_2, c_1, c_2. The challenges then are extracted from this hash bitwise (figure 8). Each two bits from the hash are transformed to a single challenge $e \in \{0, 1, 2\}$ for a three party MPC. And there we see the reason for using a hashfunction with arbitrary output lengths (SHA3-SHAKE). The needed length of this hash depends on the number of MPC rounds n (security level L1: $n = 219$) as every MPC round has it's own challenge.

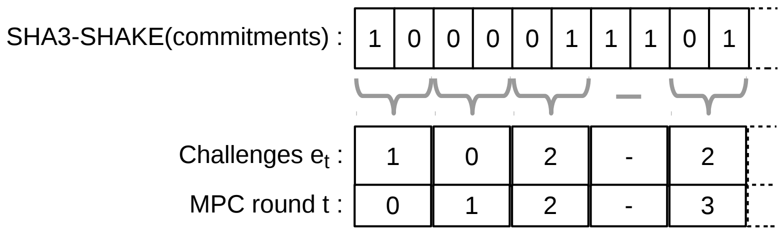

Figure 8: Hashes to challenges

Picnic has two different versions of MPC in the head defined. The first one is the Fiat-Shamir (FS) transformation, which is based on the described ROM but might not be quantum save. The second option is the Unruh (UR) transformation, which is based on a Quantum Random Oracle Model (QROM). The re-implementation Python-Picnic can only handle the FS transformation so far.

Signing a message: The Picnic loops

All the described parts taken together assemble to the Picnic signaturescheme. Picnic consists mainly of two big loops, shown in figure 9 for creating a signature (signing a message).

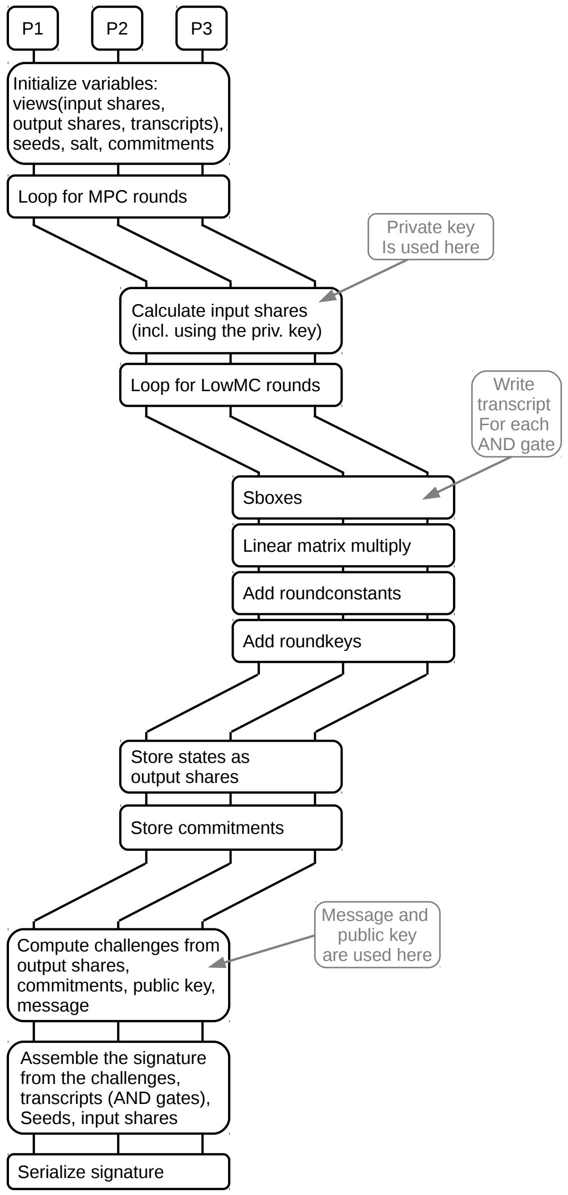
P1
P2
P3
Initialize variables: views(input shares, output shares, transcripts), seeds, salt, commitments
Loop for MPC rounds
Private key Is used here
Calculate input shares (incl. using the priv. key)
Loop for LowMC rounds
Write transcript For each AND gate
Sboxes
Linear matrix multiply
Add roundconstants
Add roundkeys
Store states as output shares
Store commitments
Message and public key are used here
Compute challenges from output shares, commitments, public key, message
Assemble the signature from the challenges, transcripts (AND gates), Seeds, input shares
Serialize signature

Figure 9: Signing a message: Picnic loops

Keys, Message and Signature

Picnic works with keypairs for signing and verifying as every other public key cryptographic algorithm. In figure 9 the parts are marked, where the keys and the message to sign is used. The generation of the keypair is done with the LowMC encryption. A random plaintext message is needed to do this. The lengths of the keys and the plaintext are defined through the security level. Therefore the **key generation** is:

- Pick a random private key sk (secret key).
- Pick a random plaintext message p (Not to confound with the message to sign).
- Compute the encryption of p with sk: $C = LowMC(sk, p)$.
- The public key is the tuple $pk = (C, p) = (LowMC(sk, p), p)$.

The **message** M to be signed must be an arbitrary bytearray of the length $1 \leq |M| \leq 2^{55}$. The messsage takes account into the creation of the challenges. In the description of "MPC in the head" above, this was not explained to keep things as simple as possible. But in the Picnic signaturescheme the challenges depend on the commitments, the output shares **and** the message M.

The Picnic signing function is defined as:

$$signature\ \delta = picnic\_sign(sk, pk, M)$$

The resulting **signature** δ can be understood as a transcript of the complete communication in the MPC protocol. The signature in Picnic includes:

- The challenges for all MPC rounds.
- The player revelations according to the challenges (transcripts, seeds and input shares). Remark: Only one player gets revealed per MPC round.
- The commitments of the player according to the challenges. Remark: Only one players commitment gets revealed per MPC round.

LowMc and the signature sizes:
At the beginning of the LowMC description it was said, that LowMC was chosen because it reduces the signature sizes in Picnic, compared to other blockcipher algorithms. The argumentation was, that the low count of AND gates does this. In figure 9 can be seen that the transcripts of the LowMC rounds are done in the sbox-part of the algorithm. And only the calculations from the AND gates get recorded as a transcript. That is the point where Picnic takes advantage of LowMC to minimize the size of the signatures.

Types and variables in the implementation

With the given explanations about the Picnic algorithm it should be not that hard to understand the Python sourcecode. The functions mostly map directly to the explanations. Instead of describing all functions in detail, an affiliation of the code-variables to the examples above will be given. We'll start with the public variables in table 1 defined through the security level of Picnic. If Python-Picnic would support more security levels (only L1 at the moment), the constructor would set these variables according to them. Supporting more security levels is implemented in Python-LowMC and it can be seen there, as an example how to implement this feature.

Variable	Description
self.blocksize	Internal blocksize of LowMC in bits (also statesize)
self.blocksize_bytes	Internal blocksize of LowMC in bytes
self.keysize	Lenght of private and public key in bits
self.rounds	Number of LowMC rounds
self.sboxes	Number of 3-Bit sboxes in LowMC
self.mpc_rounds	Number of MPC rounds
self.hash_length	Lenght of the SHAKE ouput as hashfunction in bits
self.lowmc	Name of the security level as string

Table 1: Public variables

Before the private variables, we'll declare the implementation specific types (datastructures) and describe them in table 2. They are defined in an own file, named `picnic_types.py`. All the types are classes with only a constructor and some variables. Imagine them like C structs.

Class	Variables	Description
Publickey	self.public_key self.p	Public key as BitVector, length = self.keysize Plaintext message for public key generation as BitVector, length = self.keysize
View	self.i_share self.transcript self.o_share	Single input share as a BitVector, length = self.blocksize Single transcript of a LowMC round as a BitVector, length = AND gates in one LowMC round Single output share as BitVector, length = self.blocksize
Commitment	self.hash self.n_commitments	Hash of a single commitment as a bytearray, length = self.hash_length Not used by now
Proof	self.seed_1 self.seed_2 self.i_share self.transcript self.view_3_commit	Seed of the first not revealed player in a round Seed of the second not revealed player in a round Input share of the revealed player in a round Transcript of the AND gates of the revealed player A merged commit of the three players (to save space)
Signature	self.proofs self.challenges self.salt	List of Proofs (class Proof), length = self.mpc_rounds List of all challenges, length = self.mpc_rounds Salt as bytearray, length = self.blocksize

Table 2: Picnic types (classes)

Now, with the knowledge about the types (classes) we can define the private variables. They mostly store the informations from the MPC rounds for every player (three of them). Therefore the views, commitments and seeds are two-dimensional arrays with the first dimension beeing the MPC rounds and the second dimension beeing the three players. Python does not have datatypes for arrays. They are nested lists, but can be accessed in the same way as arrays in other languages by bracket-notations ([dim 1][dim 2]). The private variables are described in table 3.

Variable	Description
self.__priv_key	Private key as BitVector, length = self.keysize
self.__pub_key	Public key as type Publickey (class)
self.__views	Two-dimensional array of Views (class) dims: [self.mpc_rounds][3 (players)]
self.__commitments	Two-dimensional array of Commitments (class) dims: [self.mpc_rounds][3 (players)]
self.__seeds	Two-dimensional array of seeds as BitVectors. Each BitVector has length = self.blocksize dims: [self.mpc_rounds][3 (players)]
self.__salt	Salt as a BitVector, length = self.blocksize
self.__tapes_pos	Position counter on the tapes. Type = Integer
self.__challenges	List of the challenges $e \in \{0, 1, 2\}$, length = self.mpc_rounds
self.__prove	List of Proofs (class), length = self.mpc_rounds
self.__signature	Stores the complete Signature (class), including proofs, challenges and salt
self.__signature_ser	Stores a serialized signature as bytearray

Table 3: Private variables

Verify a Picnic signature

To verify a given signature, nearly the same steps as to create a signature are executed. The challenges decided which one of the three players in each round got revealed. The transcripts, seeds and shares of the one revealed player got stored into the signature. In the verification the other two players must be recalculated. After recalculating the "missing" players data, the challenges are re-computed.

In figure 9 can be seen, that the computation of the challenges is the secondlast step in the creation of the signature. The data from all three players are merged into a single hash and the challenges are derived from this hash. Only after the challenges are fixed, the proofs for the signature are assembled, based on the challenges. In other words the challenges depend on the calculated data of all three players.

The signature is accepted as valid, if the re-calculation of the "missing" (not revealed) players and the data from the revealed player computes to the same challenges as in the signature.

The main algorithmic differences between sign and verify are that each MPC round and each LowMC round only calculates two players in the verification. And which two players are calculated depends on the round challenge in the signature. Therefore all internal functions are re-written to accept the round challenge as an additional parameter and calculate only the two "missing" players. This seems a little redundant in the sourcecode but keeps good review and learning capabilities. The verify functions can be recognized by the postfix `_verify` in their names.

The API to Python-Picnic

In comparison to `Python-LowMC` there are no private functions in `Python-Picnic`. All functions follow the Python scheme for public functions (no leading `__`). That could be changed in future reworks. Therefore the functions to use `Python-Picnic` are described here. Additionaly there ist a testfile (`tests.py`) in which the usage can be seen by function-call examples.

`Picnic-LowMC` got expanded from the standalone version (in the LowMC Github) to a version for the Picnic needs. For example the sbox-function in LowMC needs to store the outcomes of the AND-gates for the signature. That was not a part in the standalone LowMC. A new, expanded version of LowMC is included within `Python-Picnic` as the file `lowmc.py`.

The Picnic specifications inculde a scheme for serializing signatures. The serialization and file-storage of a signature are implemented in `Python-Picnic`. That enables a way to compare signatures to the ones from the reference implementation. The file-storage writes textfiles with the serialized signature as a HEX-string. This is not the most compact way, but enables human readability and comparison.

The serialization of the keys is also a part of the Picnic specification but is not implemented in `Python-Picnic` so far. This is a priority issue for future works. That means, that the keys are gone, if the Picnic-object is destroyed (the execution of the sourcecode ends). A function for setting the private key and the plaintext message is included. This function then derives the public key.

`generate_keys(p, priv_key)`

Sets or generates the plaintext message p (for LowMC key generation) and the private key pk. If the parameters are given, they are setted. If left emtpy, they are generated with `os.urandom()`. The public key then gets derived via LowMC encrpytion. The parameters must be bytearrays with correct length according to the security parameter. So far only "Picnic-L1" is implemented. Therefore the sizes must be 16 bytes. The data is stored in the matching private variables from table 3.

`sign(message)`

Signs a message M. The keys must be set or generated before. The message must be passed as a bytearray of arbitrary length in the bounds $1 \leq |M| \leq 2^{55}$. The signature gets stored into the private variable `self.__signature` (see table 3).

`verify(message)`

Verifies a signature regarding to the message. The keys must be set or generated before. The message must be passed as a bytearray of arbitrary length in the bounds $1 \leq |M| \leq 2^{55}$. The signature must be available in the private variable `self.__signature` before calling `verify`. This can be done by signing a message or by reading one from a file before.

`serialize_signature()`

Serializes a signature from the private variable `self.__signature` and stores it into the private variable `self.__signature_ser`.

`deserialize_signature()`

Deserializes a signature from the private variable `self.__signature_ser` and stores it into the private variable `self.__signature`.

`write_ser_sig_to_file(filename)`

Writes a serialized signature from the private variable `self.__signature_ser` to a textfile. The filename must be provided as a string. The textfile then contains the serialized signature as a HEX-string.

`read_ser_sig_to_file(filename)`

Reads a textfile with the given filename (string). The textfile must contain a serialized signature as HEX-string. The serialized signature then gets stored into the private variable `self.__signature_ser`.

`print_signature()`

Does a printout into the console of the signature in the private variable `self.__signature`. The variable must contain a signature.

`print_signature_ser()`

Does a printout into the console of the serialized signature in the private variable `self.__signature_ser` as a HEX dump. The variable must contain a serialized signature.

Prerequisites

The code is tested with Ubuntu 16.04 LTS and Python3.6. The package "BitVector" for python is required. The package "hashlib" is used for the SHA3-SHAKE parts. SHAKE is available in "hashlib" since Python 3.6. Therefore `Python-Picnic` will not run with Python versions lower than 3.6. It is recommended to use a virtual environment for Python, like `virtualenv`. In (very) short lines:

```
virtualenv -p /usr/bin/python3.6 myvenv
source /path_to_myvenv/bin/activate
<myvenv>pip install BitVector
<myvenv>python tests.py
```