Biscuit: Shorter MPC-based Signature from PoSSo

Principal submitter

This submission is from the following team, listed in alphabetical order:

- Luk Bettale, IDEMIA, France
- Delaram Kahrobaei, Queens College, City University of New York, USA
- Ludovic Perret, Sorbonne University, France
- Javier Verbel, Technology Innovation Institute, UAE

Point of Contact:
Ludovic Perret
Sorbonne University
CNRS/LIP6/PolSys
Boite courier 169
4, Place Jussieu
P: +33-6-44-27-88-35
email : `ludovic.perret@lip6.fr`
F-75252 Paris cedex 5, France

Contents

1 Introduction

This document presents Biscuit, a new multivariate Digital Signature Scheme (DSS) based on the hardness of solving a set of *hard structured algebraic equations*. Biscuit is in the lineage of the MQDSS [24, 23] and Picnic [21, 68] signature schemes that were submitted to the previous NIST post-quantum cryptography (PQC) standardization process [22, 62]. The high-level framework of Biscuit is similar to MQDSS and Picnic. It is derived from a Zero-Knowledge Proof-of-Knowledge (ZKPoK) using the Fiat-Shamir transform [38].

The central building block of MQDSS is a ZKPoK for the problem of solving a system of non-linear quadratic equations over a finite field $\mathbb{F}_q$ ($\mathtt{MQ}_q$ problem). Biscuit differs from MQDSS as it relies on special structured polynomials. The crucial hardness assumption underlying the design of Biscuit is that structured equations considered in Biscuit, which are roughly power of random affine forms (PowAff2 problem), are *generics* and as difficult to solve as random instances of $\mathtt{MQ}_q$ for Gröbner bases algorithms [19, 18].

From Picnic, we borrow the uses of Multi-Party Computation (MPC) techniques, in particular the so-called MPC-in-the-Head (MPCitH) approach [50, 46, 54], that was used to derive a ZKPoK for a block-cipher key-recovery problem. Biscuit relies on a different hard problem, but borrows from Picnic the uses of MPCitH-based proof system to derive ZKPoK. In particular, Biscuit is based on MPCitH-based proof systems, BN [9] and some optimizations from BN++ [53], that allow generating a ZK proof for the pre-image of an arbitrary arithmetic circuit defined over $\mathbb{F}$.

The signature size of a DSS derived from a MPCitH-based proof system is usually related to the number of multiplications required to evaluate the circuit. This motivates the use of systems of algebraic equations generated by the power of affine forms. Such systems can be evaluated using a much smaller number of multiplications than random algebraic equations while not being easier to solve for generic algorithms. This leads to a scheme with the following practical features :

Name	Size (bytes)			Performance (cycles)		
	sk	pk	sig	keygen	sign	verify
biscuit128s	115	50	4 758	82 632	80 555 671	78 899 797
biscuit192s	158	69	11 349	210 159	724 466 241	714 947 231
biscuit256s	212	93	20 192	332 165	1 291 111 734	1 274 568 395

Table 1: Time performance of Biscuit-short.

Name	Size (bytes)			Performance (cycles)		
	sk	pk	sig	keygen	sign	verify
biscuit128f	115	50	6 726	82 505	9 653 412	8 734 302
biscuit192f	158	69	15 129	210 150	81 492 308	75 826 788
biscuit256f	212	93	27 348	353 223	147 099 575	137 359 832

Table 2: Time perfomance of Biscuit-fast.

After this introduction, the document is divided as follows.

- Section 2 presents notation and basic concepts.
- Section 3 introduces the hard problem considered in Biscuit, the so-called `PowAff2` problem, and provides a high-level description of the main components of Biscuit, including a new `MPC` protocol for `PowAff2`.
- Section 4 is a detailed specification of Biscuit and also performance estimations.
- Section 5 considers various attacks against Biscuit and algorithms for solving `PowAff2`. In particular, it leads to an improved method to compute the number of iterations required for our `MPC` protocol.
- We conclude the document with a discussion on the advantages and limitations of Biscuit.

Acknowledgements. The Biscuit team would like to thank Daniel Escudero for spending a significant amount of time in the initial preparation of Biscuit and discussions on the use of `MPC` in multivariate cryptography.

The third author would like to thank Google which partially supported this work thanks to a gift for supporting post-quantum research and Mohab Safey El Din for insightful discussions on the hardness and genericity of `PowAff2`.

2 Notation and Basic Concepts

Notation

All over this document, we repetitively use the following notation:

- λ: the security parameter.
- $\mathbb{F}_q$: a finite field of q elements.
- $\mathbb{F}_q^m$: denotes the vector space of dimension m over $\mathbb{F}_q$.
- $\mathbb{F}_q[x_1, \ldots, x_n]$: denotes the ring of polynomials in the variables $x_1, \ldots, x_n$ over the field $\mathbb{F}_q$.
- $[n]$: the set $\{1, \ldots, n\}$ for an integer n.
- $[\![a]\!]$: an additive sharing of an element a in a ring.
- $[\![a]\!]_i$: the i-th coordinate of a sharing $[\![a]\!]$.
- $a \leftarrow \mathcal{A}(x)$: indicates that a is the output of an algorithm $\mathcal{A}$ on input x.
- $a \overset{\$}{\leftarrow} \mathcal{S}$: means that a is sampled uniformly at random from a set $\mathcal{S}$.
- Bold lower-case letters denote vectors. Given $\boldsymbol{t} \in \mathbb{F}_q^m$, then $(t_1, \ldots, t_m) \in \mathbb{F}_q^m$ denotes the vector of coordinates of $\boldsymbol{t}$ in the canonical basis.
- $\boldsymbol{x} + \boldsymbol{y}$: denotes the element-wise addition.

- $\boldsymbol{x} \odot \boldsymbol{y}$: denotes the element-wise product.
- $\|$ the concatenation of two vectors or two byte strings.

In the whole document, we use the following concepts.

Multi-party computation

- A *multi-party computation*(`MPC`) protocol is a protocol executed by a set of N parties knowing a public function f. Its goal is to compute an image $z = f(x_1, \ldots, x_N)$, where the value x_i is only known by the i-th party. An `MPC` protocol is considered secure and correct if, at the end of the protocol, every party i knows z, and no information about its secret input value x_i is revealed to the other parties.
- An *additive sharing* of an element a in a ring is a tuple $[\![a]\!] := ([\![a]\!]_1, \ldots, [\![a]\!]_N)$ such that $a = \sum_{i=1}^{N} [\![a]\!]_i$. Each $[\![a]\!]_i$ is called a *share* of a. Throughout this document, we use the word sharing to refer to additive sharing.
- A set of N parties shares an element a: Every party i knows a value $[\![a]\!]_i$, and the tuple $[\![a]\!] = ([\![a]\!]_1, \ldots, [\![a]\!]_N)$ is a sharing of a.
- We say that a set of parties open a shared value α to indicate that every party i broadcasts its own share $[\![\alpha]\!]_i$ to the other parties. Hence, all the parties obtain α.
- Let a, b, and c be elements in a ring. Suppose a and b are shared between a set of N parties, and all the parties know c. Then, the parties can compute sharings of $a + b$ and $c \cdot a$ by the following procedure: Every party $i \in [N]$ computes $[\![a]\!]_i + [\![b]\!]_i$ and $c \cdot [\![a]\!]_i$, respectively. However, to compute a sharing of $a + c$, the party $i \neq 1$ holds its share $[\![a]\!]_i$, and only the party $i = 1$ computes $[\![a]\!]_1 + c$.
- We say that the triple $([\![z]\!], [\![x]\!], [\![y]\!]) \subset \mathbb{F}_q$ is a *multiplicative triple* if it holds that $z = x \cdot y$.

MPC-in-the-Head

The MPC-in-the-Head (`MPCitH`) technique is a widely used method to build signature schemes from `MPC` protocols. It was introduced in 2007 by Ishai, Kushilevitz, Ostrovsky, and Sahai in [49]. They showed how to build robust Zero-Knowledge Proof-of-Knowledge (`ZKPoK`) for various computational problems originating from `MPC` protocols. Then, the `ZKPoK` is turned into a Digital Signature Scheme (`DSS`) by using the classical Fiat-Shamir transformation [38]. Biscuit follows this design strategy.

3 High Level Description

This section provides a general view of the design of Biscuit.

3.1 The `PowAff2` Problem

The primary hard problem considered in Biscuit is the problem of solving multivariate equations defined as the product of two affine forms. This problem is parameterized by a tuple of positive integers (n, m, q), denoted as `PowAff2`.

Definition 1 (The `PowAff2` problem)**.** *Given* $\boldsymbol{t} = (t_1, \ldots, t_m) \in \mathbb{F}_q^m$ *and affine forms* $A_{k,j}(x_1, \ldots, x_n) \in \mathbb{F}_q[x_1, \ldots, x_n]$ *with* $k \in [m]$ *and* $0 \leq j \leq 2$*, i.e.,*

$$A_{k,j} = a_0^{(k,j)} + \sum_{i=1}^{n} a_i^{(k,j)} x_i, \ \text{with } a_0^{(k,j)}, \ldots, a_n^{(k,j)} \in \mathbb{F}_q. \tag{1}$$

The `PowAff2` *problem asks to find – if any – a vector* $(s_1, \ldots, s_n) \in \mathbb{F}_q^n$ *such that:*

$$f_1(s_1, \ldots, s_n) = t_1, \ldots, f_m(s_1, \ldots, s_n) = t_m,$$

where $f_k(x_1, \ldots, x_n) = A_{k,0}(x_1, \ldots, x_n) + \prod_{j=1}^{2} A_{k,j}(x_1, \ldots, x_n)$*, for all* $k \in [m]$

`PowAff2` problem can be seen as a structured variant of the problem of solving m quadratic equations in $\mathbb{F}_q[x_1, \ldots, x_n]$, i.e. the Multivariate Quadratic ($\texttt{MQ}_q$) problem. It is known to be `NP`-Hard [44], and its hardness, in the average case, has been widely studied in the literature [10]. Hence, solving random instances of $\texttt{MQ}_q$ is widely considered as a hard computational problem.

When $m = n$ and for a sufficiently big field, we prove that polynomials as in the `PowAff2` problem behave as generic quadratic polynomials (see Theorem 3). This is also expected to be the case for $m > n$ [61], which is the setting considered in Biscuit. However, a formal proof of that is equivalent to solving the Fröberg conjecture [39, 4, 1], which is a long-standing problem in commutative algebra. Thus, solving random instances of the `PowAff2` problem would be as hard as random instances of the $\texttt{MQ}_q$ problem against generic algorithms.

Being secure against generic algorithms for the $\texttt{MQ}_q$ problem does not exclude the possibility of having potentially new faster algorithms that take into account the structure of the equations or the knowledge of the affine maps $A_{i,j}$. For instance, we can do a faster exhaustive search (see Section 5.2.2). However, polynomial systems with the structure of the `PowAff2` problem have been extensively studied before in post-quantum cryptography. They appear on algebraic attacks assumption against the Learning With Errors (`LWE`) problem with binary errors [1, 2] (which reduces to standard `LWE` [60], one of the main problems in lattice-based cryptography). Therefore, we are confident in the hardness of the `PowAff2` problem.

3.2 Multi-Party Protocol for `PowAff2`

In this section, we describe our `MPC` protocol to verify a solution of the `PowAff2` problem given quadratic polynomials $\boldsymbol{f} = (f_1, \ldots, f_m) \in \mathbb{F}_q[x_1, \ldots, x_n]^m$ and a target vector $\boldsymbol{t} \in \mathbb{F}_q^m$ as defined in Definition 1.

The protocol is described in Figure 1, and it is run by N parties sharing a vector $\boldsymbol{s} \in \mathbb{F}_q^n$. It is assumed that every party knows the target vector $\boldsymbol{t}$ and the affine forms $A_{k,0}, A_{k,1}$ and $A_{k,2}$ such that $f_k = A_{k,0} + A_{k,1} \cdot A_{k,2}$, for each $k \in [m]$. At the end of the protocol, the parties output **accept**

```
Inputs : Each party knows $\boldsymbol{t} \in \mathbb{F}_q^m$ and the linear polynomials $A_{1,0}, \ldots, A_{m,2} \in \mathbb{F}_q[x_1, \ldots, x_n]$
 such that $\boldsymbol{f} = (f_1, \ldots, f_m)$, and $f_k = A_{k,0} + \prod_{j=1}^{2} A_{k,j}$. The $i$-th party knows $[\![\boldsymbol{s}]\!]_i \in \mathbb{F}_q^n$,
 $[\![\boldsymbol{a}]\!]_i \in \mathbb{F}_q^m$, where $\boldsymbol{a} \xleftarrow{\$} \mathbb{F}_q^m$, and $[\![\boldsymbol{c}]\!]_i \in \mathbb{F}_q^m$, where $c_k = A_{k,1}(\boldsymbol{s}) \cdot a_k$.
MPC Protocol:
    for $k \in [m]$
 1 :   The parties locally compute $[\![z_k]\!] \leftarrow t_k - A_{k,0}([\![\boldsymbol{s}]\!])$, $[\![x_k]\!] \leftarrow A_{k,1}([\![\boldsymbol{s}]\!])$, and $[\![y_k]\!] \leftarrow A_{k,2}([\![\boldsymbol{s}]\!])$ .
 2 :   The parties get a random matrix $\varepsilon_k \xleftarrow{\$} \mathbb{F}_q$.
 3 :   The parties locally set $[\![\alpha_k]\!] \leftarrow ([\![x_k]\!] \cdot \varepsilon_k + [\![a_k]\!])$.
 4 :   The parties open $[\![\alpha_k]\!]$ so that they all obtain $\alpha_k$.
 5 :   The party locally compute $[\![v_k]\!] = [\![y_k]\!] \cdot \alpha_k - [\![z_k]\!] \cdot \varepsilon_k - [\![c_k]\!]$.
 6 :   The parties open $[\![v_k]\!]$ to obtain $v_k$.
   The parties output accept if $v_k = 0$ for all $k = 1, \ldots, m$ and reject otherwise.
```

Figure 1: MPC protocol to check that $\boldsymbol{t} = \boldsymbol{f}(\boldsymbol{s})$.

indicating they are convinced that the shared vector $\boldsymbol{s}$ satisfies $\boldsymbol{t} = \boldsymbol{f}(\boldsymbol{s})$. Otherwise, they output **reject**.

Our MPC protocol consists of m iterations of the somewhat standard MPC protocol introduced in [9] (along with the optimization given in [53, Section 2.5]) to check multiplicative triples of sharing.

The following is exactly Lemma 2 from [53] in the special case $C = 1$.

Lemma 1. *Let x_k, y_k, z_k, a_k and c_k be in $\mathbb{F}_q$. Suppose that a set of N, with input $([\![x_k]\!], [\![y_k]\!], [\![z_k]\!], [\![a_k]\!], [\![c_k]\!])$ from step 2 to step 6. Thus, if $z_k \neq x_k y_k$ or $c_k \neq x_k a_k$, then $v_k = 0$ with $1/q$.*

Proposition 1. *Suppose that a set of N parties genuinely follow the MPC protocol given in Figure 1 with inputs $\boldsymbol{t} \in \mathbb{F}_q^m$, $\boldsymbol{f} = (f_1, \ldots, f_m) \in \mathbb{F}_q[x_1, \ldots, x_n]^m$, and $[\![\boldsymbol{s}]\!] \in \left(\mathbb{F}_q^n\right)^N$. Let u be the number of indexes for which $t_k \neq f_k(\boldsymbol{s})$. If $u = 0$, i.e., $\boldsymbol{t} = \boldsymbol{f}(\boldsymbol{s})$, then the parties **accept**. Otherwise, if $\boldsymbol{t} \neq \boldsymbol{f}(\boldsymbol{s})$, then the parties **accept** with probability $1/q^u$.*

Proof. The proof follows directly from Lemma 1. □

3.3 The Biscuit Digital Signature Scheme

The Biscuit Digital Signature Scheme (DSS) results from applying the MPCitH technique to the MPC protocol given in Figure 1.

In Algorithm 1, we provide a high-level description of the Biscuit signing process of a message msg. A more detailed pseudocode of the signature generation is given in Algorithm 12. Roughly speaking, the signing process consists of a proof of τ honest message-dependent executions of the protocol given in Figure 1 all with **accept** as output.

Overall, the signing of a message msg is divided into 5 phases:

Algorithm 1 Signature (high-level description)

```
Require: $\boldsymbol{t} = \boldsymbol{f}(\boldsymbol{s}) \in \mathbb{F}_q^n$ such that $f_k = A_{k,0} + A_{k,1} \cdot A_{k,2}$ for all $k \in [m]$ and $A_{1,0}, \ldots, A_{m,2} \in \mathbb{F}_q[x_1, \ldots, x_n]$ are affine forms.
 1: procedure $\texttt{Sign}(\mathsf{msg}, \boldsymbol{s}, (\boldsymbol{t}, \boldsymbol{f}))$
    Phase 1: Setting parties inputs for MPC protocols.
 2:     $\mathsf{salt}, \mathsf{root}^{(1)}, \ldots, \mathsf{root}^{(\tau)} \xleftarrow{\$} \{0,1\}^{2\lambda}$
 3:     for $e \in [\tau]$ do
 4:         $\mathsf{seed}^{(e,1)}, \ldots, \mathsf{seed}^{(e,N)} \leftarrow \texttt{GetSeeds}(\mathsf{root}^{(e)}, \mathsf{salt})$
 5:         for $i \in [N]$ do
 6:             $[\![\boldsymbol{s}]\!]_i^{(e)}, [\![\boldsymbol{c}]\!]_i^{(e)}, [\![\boldsymbol{a}]\!]_i^{(e)} \leftarrow \texttt{PRF}(\mathsf{seed}^{(e,i)})$ and $\mathsf{com}^{(e,i)} \leftarrow \texttt{Commit}(\mathsf{salt}, e, i, \mathsf{seed}^{(e,i)})$
 7:         end for
 8:         $\boldsymbol{a}^{(e)} \leftarrow \sum_{i=1}^N [\![\boldsymbol{a}]\!]_i^{(e)}$,   $\boldsymbol{c}^{(e)} \leftarrow (A_{k,1}(\boldsymbol{s}) \cdot a_k)_{k\in[m]}$
 9:         $\boldsymbol{\Delta s}^{(e)} \leftarrow \boldsymbol{s} - \sum_{i=1}^N [\![\boldsymbol{s}]\!]_i^{(e)}$ and $\boldsymbol{\Delta c}^{(e)} \leftarrow \boldsymbol{c}^{(e)} - \sum_{i=1}^N [\![\boldsymbol{c}]\!]_i^{(e)}$ .
10:         $[\![\boldsymbol{s}]\!]_1^{(e)} \leftarrow [\![\boldsymbol{s}]\!]_1^{(e)} + \boldsymbol{\Delta s}^{(e)}$ and $[\![\boldsymbol{c}]\!]_1^{(e)} \leftarrow [\![\boldsymbol{c}]\!]_1^{(e)} + \boldsymbol{\Delta c}^{(e)}$
11:     end for
12:     $\sigma_1 \leftarrow \big(\mathsf{com}^{(e,i)} : (e,i) \in [\tau] \times [N]\big)$.
    Phase 2: Challenges for the protocols
13:     $h_1 \leftarrow \texttt{H1}(\mathsf{salt}, \mathsf{msg}, \sigma_1)$ and $\Big(\varepsilon_k^{(e)} : (e,k) \in [\tau] \times [m]\Big) \leftarrow \texttt{PRF}(h_1)$
    Phase 3: Simulation of checking protocols
14:     $\forall (e,i) \in [\tau] \times [N]$, use $[\![\boldsymbol{s}]\!]_i^{(e)}$, $[\![\boldsymbol{c}]\!]_i^{(e)}$, $[\![\boldsymbol{a}]\!]_i^{(e)}$ and $\varepsilon_k^{(e)}$ to simulate the $i$-th party in the MPC protocol in Figure 1.
15:     $\sigma_2 \leftarrow \Big([\![\alpha_k]\!]_i^{(e)} \,\|\, [\![v_k]\!]_i^{(e)} : (e,i,k) \in [\tau] \times [N] \times [m]\Big)$
    Phase 4: Challenging the views of the MPC protocol
16:     $h_2 \leftarrow \texttt{H2}(\mathsf{salt}, h_1, \sigma_2)$ and $\bar{i}_1, \ldots, \bar{i}_\tau \leftarrow \texttt{PRF}(h_2)$
    Phase 5: Opening the views of the checking protocol
17:     $\mathsf{path}^{(e)} \leftarrow \Big(\mathsf{seed}^{(e,i)}\Big)_{i \neq \bar{i}_e}$, $\forall e \in [\tau]$
18:     $\sigma \leftarrow \mathsf{salt} \| h_1 \| h_2 \| \Big(\mathsf{path}^{(e)} \| \mathsf{com}^{(e,\bar{i}_e)} \| \boldsymbol{\Delta s}^{(e)} \| \boldsymbol{\Delta c}^{(e)} \| [\![\boldsymbol{\alpha}]\!]_{\bar{i}_e}^{(e)}\Big)$        ▷ $[\![\boldsymbol{\alpha}]\!]_{\bar{i}_e}^{(e)} = ([\![\alpha_k]\!]_{\bar{i}_e}^{(e)})_{k\in[m]}$
19:     return $\sigma$
20: end procedure
```

- **Phase** 1. The signer samples a random salt, generates the inputs of all the N parties to run τ executions of the MPC protocol, and commits to these values. The inputs of the i-th party in the e-th execution of the protocol are generated from $\mathsf{seed}^{(e,i)}$, the commit $\mathsf{com}^{(e,i)}$ is the output of the commitment scheme Commit() on input $(\mathsf{salt}, e, i, \mathsf{seed}^{(e,i)})$.

- **Phase** 2. The signer uses a hash function H2 and a pseudo-random function PRF to generate the τm challenges for the τ executions of the MPC protocol.

- **Phase** 3. The signer performs the computations of every party i on every round e with the corresponding inputs to compute $[\![\alpha_k]\!]_i^{(e)}$ and $[\![v_k]\!]_i^{(e)}$. Notice these are the only values open in the protocol.

- **Phase** 4. The signer first uses a hash function H2 to hash the values computed in phase 3. Then, it uses the PRF to obtain the party inputs $\bar{i}_e$ that it won't open to the verifier of the e-th execution.

- **Phase** 5. Finally, the signature is assembled. It contains the salt, the hashes h_1, h_2, and all the necessary information to verify the computations of any party $i \neq \bar{i}_e$ all the executions of the MPC protocol.

Algorithm 2 Verification (high-level description)

```
Require: $\boldsymbol{t} = \boldsymbol{f}(\boldsymbol{s}) \in \mathbb{F}_q^n$ such that $f_k = A_{k,0} + A_{k,1} \cdot A_{k,2}$ for all $k \in [m]$ and $A_{1,0}, \dots, A_{m,2} \in \mathbb{F}_q[x_1, \dots, x_n]$ are affine forms and $\sigma = h_1 \| h_2 \| \left(\mathsf{path}^{(e)} \| \mathsf{com}^{(e,\bar{i}_e)} \| \boldsymbol{\Delta s}^{(e)} \| \boldsymbol{\Delta c}^{(e)} \| \, [\![\boldsymbol{\alpha}]\!]_{\bar{i}_e}^{(e)}\right)_{e \in [\tau]}$.
1:  procedure Verify($\mathsf{msg}, \sigma, (\boldsymbol{t}, \boldsymbol{f})$)
2:      $\left(\varepsilon_k^{(e)} : (e,k) \in [\tau] \times [m]\right) \leftarrow \mathtt{PRF}(h_1)$
3:      $\bar{i}_1, \dots, \bar{i}_\tau \leftarrow \mathtt{PRF}(h_2)$
4:      for $e \in [\tau]$ do
5:          $\forall i \in [N] \setminus \{\bar{i}_e\}$ compute $[\![\boldsymbol{s}]\!]_i^{(e)}$, $[\![\boldsymbol{c}]\!]_i^{(e)}$, $[\![\boldsymbol{a}]\!]_i^{(e)}$, and $\mathsf{com}^{(e,i)}$ as in Algorithm 1.
6:      end for
7:      $\sigma_1 \leftarrow \left(\mathsf{com}^{(e,i)} : (e,i) \in [\tau] \times [N]\right)$
8:      $h_1' \leftarrow \mathtt{H1}(\mathsf{salt}, \mathsf{msg}, \sigma_1)$
9:      $\forall (e,i,k) \in [\tau] \times ([N] \setminus \{\bar{i}_e\}) \times [m])$ compute $([\![\alpha_k]\!]_i^{(e)} \,\|\, [\![v_k]\!]_i^{(e)})$ as in Algorithm 1.
10:     for $e \in [\tau]$ do
11:         $([\![\alpha_k]\!]_{\bar{i}_e}^{(e)})_{k \in [m]} \leftarrow [\![\boldsymbol{\alpha}]\!]_{\bar{i}_e}^{(e)}$ and $[\![v_k]\!]_{\bar{i}_e}^{(e)} = -\sum_{i \in [N] \setminus \{\bar{i}_e\}} [\![v_k]\!]_i^{(e)}$
12:     end for
13:     $\sigma_2 \leftarrow \left([\![\alpha_k]\!]_i^{(e)} \,\|\, [\![v_k]\!]_i^{(e)} : (e,i,k) \in [\tau] \times [N] \times [m]\right)$
14:     $h_2' \leftarrow \mathtt{H2}(\mathsf{salt}, h_1', \sigma_2)$
15:     if $(h_1 \overset{?}{=} h_1')$ and $(h_2 \overset{?}{=} h_2')$ then
16:         return valid
17:     end if
18:     return invalid
19: end procedure
```

A high-level description of the verification process is given in Algorithm 2. First, the verifier extracts the challenges used in all the executions of the MPC protocols. Second, it computes $\mathsf{com}^{(e,i)}$ for every $e \in [\tau]$ and $i \neq \bar{i}_e$, and it computes h_1', as in step 8, using the $\mathsf{com}^{(e,\bar{i}_e)}$ given in the signature. Then, for every execution $e \in [\tau]$, it extracts the inputs of each party $i \neq \bar{i}_e$ and follows the MPC to obtain $[\![\alpha_k]\!]_i^{(e)}$ and $[\![v_k]\!]_i^{(e)}$. Finally, it uses the $[\![\alpha_k]\!]_{\bar{i}_e}^{(e)})_{k \in [m]}$ given in the signature and the missing $[\![v_k]\!]_{\bar{i}_e}^{(e)}$ to compute h_2' as in step 15. The verifier outputs **valid** if and only if $h_1 = h_1'$ and $h_2 = h_2'$. Otherwise, it outputs **invalid**.

4 Biscuit Specification

4.1 Parameter Space

The main parameters involved in Biscuit are:

- λ: security parameter of Biscuit,

- q: size of the finite field,
- n: number of variables in the public equations,
- m: number of public equations,
- τ: number of executions of the `MPC` protocol,
- N: number of parties in the `MPC` protocol.

4.2 Chosen Parameters

We specify two sets of parameters in this document:

- One achieving small signature while being slower to execute
- One achieving faster execution but with slightly larger signature.

Both sets use $q = 16$. By taking the field to be of characteristic 2, we take advantage of the fact that the addition of an element is an XOR that can be efficiently vectorized in the implementation. The parameters are given in Table 3 and Table 4.

In Section 5.4, we specify precisely how these parameters achieve a security level $\lambda \in \{128, 192, 256\}$.

Name	Parameters					Size (bytes)		
	q	n	m	τ	N	sk	pk	sig
`biscuit128s`	16	64	67	18	256	115	50	4 758
`biscuit192s`	16	87	90	30	256	158	69	11 349
`biscuit256s`	16	118	121	40	256	212	93	20 192

Table 3: Parameters of Biscuit-short.

Name	Parameters					Size (bytes)		
	q	n	m	τ	N	sk	pk	sig
`biscuit128f`	16	64	67	34	16	115	50	6 726
`biscuit192f`	16	87	90	54	16	158	69	15 129
`biscuit256f`	16	118	121	73	16	212	93	27 348

Table 4: Parameters of Biscuit-fast.

4.3 Data Representation

As described in Section 3.2, one iteration of the `MPC` protocol consist in m parallel execution of the multiplication checking protocol. These m can be done in parallel by representing elements of $\mathbb{F}_q$ as vectors. In this specification, we exclusively such vectors.

The Biscuit algorithms mainly use two types of data:

- **byte strings** for hashes, commitments, seeds, ...
- **vectors of elements in $\mathbb{F}_q$** for circuit evaluation and multiplication checking protocol.

We represent a vector of n elements in $\mathbb{F}_q$ as string of $n \lceil \log_2(q) \rceil$ bits, where binary representation of an element in $\mathbb{F}_q$ is represented with the least significant bit first.

Sometimes, one or several vectors have to be represented as byte strings (for output keys/signatures or input of hash functions for instance). When necessary, we represent a vector of n elements as a string of $k = n \lceil \log_2(q) \rceil$ bits $(b_1, \ldots, b_k)$, each group of $\lceil \log_2(q) \rceil$ representing one element in $\mathbb{F}_q$. From this, we obtain $K = \lceil \frac{k}{8} \rceil$ bytes $(B_1, \ldots, B_K)$ such that $B_i = \sum_{j=1}^{8} b_{(i-1)\cdot 8+j} \cdot 2^{(j-1)}$ for all $i \in [k]$ with the convention that $b_{(i-1)\cdot 8+j} = 0$ if $(i-1)\cdot 8 + j > k$.

This procedure is executed by the function $\texttt{pack}(\boldsymbol{v})$. It will allow to to pack together the concatenation of several vectors into one byte string (e.g. $\texttt{pack}(\boldsymbol{v}^{(1)} \| \boldsymbol{v}^{(2)} \| \ldots)$).

4.4 Auxiliary Functions

We use the SHAKE256 [31] extendable-output function (XOF) as a basic block to build the hash functions, the key-derivation and the pseudo-random functions. It takes as input a pair (input_bytes, output_len), where input_bytes is a string of bytes, and it outputs a string of output_bitlen bits which shall be a multiple of 8. We use the notation output_bitlen $= \infty$ to indicate that the function returns a tape object (typically, a finalized SHAKE256 state) that can be used to generate an arbitrary number of bits later on.

Let enc16() be the encoding of an integer on 2 bytes with the least significant byte first. We define:

- $\texttt{Commit}(\mathsf{salt}, e, i, \mathsf{seed})$: $\texttt{SHAKE256}(\texttt{0x00} \| \mathsf{salt} \| \texttt{enc16}(e) \| \texttt{enc16}(i) \| \mathsf{seed}, 2\lambda)$
- $\texttt{H1}(\mathsf{salt}, \mathsf{msg}, \sigma_1)$: $\texttt{SHAKE256}(\texttt{0x01} \| \mathsf{salt} \| \mathsf{msg} \| \sigma_1, 2\lambda)$
- $\texttt{H2}(\mathsf{salt}, h_1, \sigma_2)$: $\texttt{SHAKE256}(\texttt{0x02} \| \mathsf{salt} \| h_1 \| \sigma_2, 2\lambda)$
- $\texttt{ChildSeeds}(\mathsf{salt}, e, k, j, \mathsf{seed})$: $\texttt{SHAKE256}(\texttt{0x03} \| \mathsf{salt} \| \texttt{enc16}(e) \| \texttt{enc16}(k) \| \texttt{enc16}(j) \| \mathsf{seed}, 2\lambda)$
- $\texttt{ExpandTape}(\mathsf{salt}, e, i, \mathsf{seed})$: $\texttt{SHAKE256}(\texttt{0x04} \| \mathsf{salt} \| \texttt{enc16}(e) \| \texttt{enc16}(i) \| \mathsf{seed}, \infty)$
- $\texttt{PRF}(\mathsf{input}, k)$: $\texttt{SHAKE256}(\mathsf{input}, k)$

4.4.1 Seed generation using trees

During signature, the signer must generate a set of N seeds and reveal $N-1$ of them to the verifier for each iteration. The verifier then uses these seeds to check that the MPC protocol was correctly simulated. A binary tree structure allows generating the seeds using one root seed from a binary tree. Instead of sending $N-1$ seeds in the signature, this allows sending only $\lceil \log_2 N \rceil$ seeds that will be used to reconstruct all $N-1$ seeds required.

We describe in Algorithm 3 how to build all N seeds from a single root seed. In Algorithm 4, how to build a path of $\log_2 N$ values that will allow to recover all N seeds except one. Basically, it consists in giving the root of all the subtrees that the excluded seed does not belong to. Finally Algorithm 5 describe how to effectively recover the $N-1$ seeds from a path.

Algorithm 3 Get N seeds from a root seed

```
1:  procedure GetSeeds(root, salt, $e$, $N$)
2:      seeds $\leftarrow$ [root]
3:      for $k$ in $\{1, \dots, \lceil \log_2 N \rceil\}$ do
4:          newseeds $\leftarrow \emptyset$
5:          $j \leftarrow 0$
6:          for $s$ in seeds do
7:              seed$_1$, seed$_2$ $\leftarrow$ ChildSeeds(salt, $e$, $k-1$, $j$, $s$)
8:              newseeds $\leftarrow$ newseeds$||$[seed$_1$, seed$_2$]
9:              $j \leftarrow j+1$
10:         end for
11:         seeds $\leftarrow$ newseeds
12:     end for
13:     return seeds
14: end procedure
```

Algorithm 4 Get the path to recover all the seeds except the $\bar{i}$-th one from a root seed

```
1:  procedure GetPath(root, salt, $e$, $\bar{i}$, $N$)
2:      path $\leftarrow \emptyset$
3:      $s \leftarrow$ root
4:      for $k$ in $\{1, \dots, \lceil \log_2 N \rceil\}$ do
5:          $\bar{j} \leftarrow \left\lfloor \frac{\bar{i}-1}{2^{\lceil \log_2(N) \rceil - k}} \right\rfloor$          $\triangleright$ the $k$ most significant bits of $(\bar{i}-1)$
6:          seed$_1$, seed$_2$ $\leftarrow$ ChildSeeds(salt, $e$, $k-1$, $\lfloor \bar{j}/2 \rfloor$, $s$)
7:          if $\bar{j} \bmod 2 = 1$ then
8:              path $\leftarrow$ path$||$seed$_1$
9:              $s \leftarrow$ seed$_2$
10:         else
11:             $s \leftarrow$ seed$_1$
12:             path $\leftarrow$ path$||$seed$_2$
13:         end if
14:     end for
15:     return path
16: end procedure
```

Algorithm 5 Get N seeds using a path for $\bar{i}$

```
1:  procedure GetPathSeeds($\mathsf{path}, \mathsf{salt}, e, \bar{i}, N$)
2:      Extract $\mathsf{path}_1, \ldots, \mathsf{path}_{\lceil \log_2 N \rceil}$ form $\mathsf{path}$
3:      $\mathsf{seeds} \leftarrow [?]$
4:      for $k$ in $\{1, \ldots, \lceil \log_2 N \rceil\}$ do
5:          $\mathsf{newseeds} \leftarrow \emptyset$
6:          $j \leftarrow 0$
7:          $\bar{j} \leftarrow \left\lfloor \frac{\bar{i}-1}{2^{\lceil \log_2(N) \rceil - k}} \right\rfloor$          ▷ the $k$ most significant bits of $(\bar{i} - 1)$
8:          for $s$ in $\mathsf{seeds}$ do
9:              if $s \neq ?$ then          ▷ equivalent to $j = \lfloor \bar{j}/2 \rfloor$
10:                 $\mathsf{seed}_1, \mathsf{seed}_2 \leftarrow$ ChildSeeds($\mathsf{salt}, e, k-1, j, s$)
11:                 $\mathsf{newseeds} \leftarrow \mathsf{newseeds} || [\mathsf{seed}_1, \mathsf{seed}_2]$
12:             else
13:                 if $\bar{j}$ mod $2 = 1$ then
14:                     $\mathsf{newseeds} \leftarrow \mathsf{newseeds} || [\mathsf{path}_k, ?]$
15:                 else
16:                     $\mathsf{newseeds} \leftarrow \mathsf{newseeds} || [?, \mathsf{path}_k]$
17:                 end if
18:             end if
19:             $j \leftarrow j + 1$
20:         end for
21:         $\mathsf{seeds} \leftarrow \mathsf{newseeds}$
22:     end for
23:     return $\mathsf{seeds}$          ▷ seeds has '?' at the $\bar{i}$-th position
24: end procedure
```

4.4.2 Generate values in specific sets

In Biscuit, it is required to generate many values belonging to mathematical structures like $\mathbb{F}_q$ or $\mathbb{F}_q[x_1, \ldots, x_n]$. These values have to be sampled uniformly from bit strings. This is done with rejection sampling (for odd characteristics). We describe below the required algorithms. It has to be noticed that when q is a power of 2 as in the chosen parameters from Section 4.2, then the rejection is not required. It is still described here for completeness.

Algorithm 6 describes how to extract one vector of n values in a set $\mathcal{E}$ from a tape (e.g. provided by ExpandTape). When several separate vectors are required, we use the Expand function described in Algorithm 7. A special case of this is when we need to generate the coefficients of a whole polynomial system from a seed. This is described in Algorithm 8.

Algorithm 6 Sample n values from a set $\mathcal{E}$ from an infinite tape

Require: `map` is any bijective mapping from $\{0, \ldots, |\mathcal{E}| - 1\}$ to $\mathcal{E}$.

Ensure: A multiple of 8 bits are extracted from `tape` to sample one vector.

```
 1: procedure Sample(tape, $\mathcal{E}$, $n$)
 2:     $\ell \leftarrow \lceil \log_2(|\mathcal{E}|) \rceil$
 3:     if $n \overset{?}{=} 1$ then
    We are sampling an index in $\mathcal{E} = \{1, \ldots, N\}$
 4:         Let $\ell_8 = 8 \cdot \lceil \ell/8 \rceil$
 5:         repeat
 6:             $(b_0, \ldots, b_{\ell_8 - 1}) \leftarrow$ next $\ell_8$ bits of tape and update tape
 7:             $v_1 \leftarrow \sum_{k=1}^{\ell-1} b_k \, 2^k$                ▷ Use only the $\ell$ first bits
 8:         until $v_1 < |\mathcal{E}|$
 9:         $u_1 \leftarrow$ map($v_1, \mathcal{E}$)
10:     else
    We are sampling a vector of elements in $\mathcal{E} = \mathbb{F}_q$
11:         $\gamma \leftarrow 0$                ▷ to count the total number of consumed bits
12:         for $j$ in $\{1, \ldots, n\}$ do
13:             repeat
14:                 $(b_0, \ldots, b_{\ell-1}) \leftarrow$ next $\ell$ bits of tape and update tape
15:                 $\gamma \leftarrow \gamma + \ell$
16:                 $v_j \leftarrow \sum_{k=1}^{\ell-1} b_k \, 2^k$
17:             until $v_j < |\mathcal{E}|$                ▷ Always true if $q$ is a power of 2
18:             $u_j \leftarrow$ map($v_j, \mathcal{E}$)
19:         end for
20:         if $\gamma \bmod 8 \neq 0$ then
21:             Consume $8 - (\gamma \bmod 8)$ next bits of tape and update tape
22:         end if
23:     end if
24:     return $\boldsymbol{u} = (u_1, \ldots, u_n)$
25: end procedure
```

Algorithm 7 Expand k values using `Sample` from a seed

```
1: procedure Expand(seed, $k$, $\mathcal{E}$, $n$)
2:     tape $\leftarrow$ SHAKE256(seed, $\infty$)
3:     for $i$ in $\{1, \ldots, k\}$ do
4:         $\boldsymbol{v}^{(i)} \leftarrow$ Sample(tape, $\mathcal{E}$, $n$)
5:     end for
6:     return $\boldsymbol{v}^{(1)}, \ldots, \boldsymbol{v}^{(k)}$
7: end procedure
```

Algorithm 8 Expansion of a system from a seed

```
Ensure: $\boldsymbol{f} = (f_1, \dots, f_m) \in \mathbb{F}_q[x_1, \dots, x_n]^m$ with $f_k = A_{k,0} + A_{k,1} \cdot A_{k,2}$ for all $k \in [m]$ and
    $A_{1,0}, \dots, A_{m,2} \in \mathbb{F}_q[x_1, \dots, x_n]$ are affine forms as in Equation (1).
 1: procedure ExpandCircuit($\mathsf{seedF}, \mathbb{F}_q, n, m$)
 2:     $\mathsf{tape} \leftarrow \mathtt{SHAKE256}(\mathsf{seedF}, \infty)$
 3:     for $j$ in $\{1, 2, 0\}$ do                                    ▷ $A_{i,0}$'s are sampled sampled last
 4:         $\boldsymbol{b}^{(j)} \leftarrow \mathtt{Sample}(\mathsf{tape}, \mathbb{F}_q, m)$          ▷ Gather the constant terms for efficient evaluation
 5:         for $k$ in $\{1, \dots, m\}$ do
 6:             $\boldsymbol{a}^{(k,j)} \leftarrow \mathtt{Sample}(\mathsf{tape}, \mathbb{F}_q, n)$
 7:             $a_0^{(k,j)} \leftarrow b_k^{(j)}$
 8:         end for
 9:     end for
10:     $f_k \leftarrow \left(a_0^{(k,0)} + \sum_{i=1}^n a_i^{(k,0)} x_i\right) + \prod_{j=1}^2 \left(a_0^{(k,j)} + \sum_{i=1}^n a_i^{(k,j)} x_i\right)$, for $k \in [m]$
11:     return $\boldsymbol{f} = (f_1, \dots, f_m)$
12: end procedure
```

4.4.3 Circuit evaluations

Biscuit is based on the evaluation of a circuit, more precisely the evaluation of structured quadratic polynomials as in Definition 1. In our case, each party has to evaluate the linear parts of the circuit on their shares. In Algorithm 9, we describe the algorithm to evaluate the linear parts of the circuit. The output of the algorithm is the input shares of the occuring multiplications in $\boldsymbol{x}$ and $\boldsymbol{y}$, as well as the expected output shares in $\boldsymbol{z}$. As the circuit also has an affine part, the index of the party performing the computation is required too so that the affine part is taken into account only for one of the parties.

In addition, we also describe in Algorithm 10 the algorithm to evaluate the circuit on the unmasked secret and output the ciphertext $\boldsymbol{t}$ as well as on the multiplication input vector that will be usefull in the protocol.

Algorithm 9 Evaluation of the linear part of the circuit

```
Require: $\boldsymbol{f} = (f_1, \dots, f_m) \in \mathbb{F}_q[x_1, \dots, x_n]^m$ with $f_k = A_{k,0} + A_{k,1} \cdot A_{k,2}$ for all $k \in [m]$ and
    $A_{1,0}, \dots, A_{m,2} \in \mathbb{F}_q[x_1, \dots, x_n]$
Ensure: $(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$ are the correct multiplication triples
 1: procedure LinearCircuit($\boldsymbol{s}, \boldsymbol{t}$, idx, $\boldsymbol{f}$)
 2:     if idx $\overset{?}{=} 1$ then                    ▷ Keep the constant part only for the first party
 3:         $A'_{k,j} \leftarrow A_{k,j}$            for $k \in [m]$, $j \in \{0, \dots, 2\}$
 4:     else
 5:         $A'_{k,j} \leftarrow A_{k,j} - a_0^{(k,j)}$ for $k \in [m]$, $j \in \{0, \dots, 2\}$
 6:     end if
 7:     $\boldsymbol{x} \leftarrow \big(A'_{1,1}(\boldsymbol{s}), \dots, A'_{m,1}(\boldsymbol{s})\big)$
 8:     $\boldsymbol{y} \leftarrow \big(A'_{1,2}(\boldsymbol{s}), \dots, A'_{m,2}(\boldsymbol{s})\big)$
 9:     if idx $\overset{?}{=} 1$ then
10:         $\boldsymbol{z} \leftarrow -\big(A'_{1,0}(\boldsymbol{s}), \dots, A'_{m,0}(\boldsymbol{s})\big)$
11:     else
12:         $\boldsymbol{z} \leftarrow \boldsymbol{t} - \big(A'_{1,0}(\boldsymbol{s}), \dots, A'_{m,0}(\boldsymbol{s})\big)$
13:     end if
14:     return $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$
15: end procedure
```

Algorithm 10 Evaluation of the circuit with multiplications results

```
Require: $\boldsymbol{f} = (f_1, \dots, f_m) \in \mathbb{F}_q[x_1, \dots, x_n]^m$ with $f_k = A_{k,0} + A_{k,1} \cdot A_{k,2}$ for all $k \in [m]$ and
    $A_{1,0}, \dots, A_{m,2} \in \mathbb{F}_q[x_1, \dots, x_n]$
Ensure: $z_\ell = x_\ell \cdot y_\ell$, for $\ell \in [m]$ are the multiplications occurring during an evaluation
 1: procedure EvalCircuit($\boldsymbol{s}, \boldsymbol{f}$)
 2:     $\boldsymbol{x} \leftarrow (A_{1,1}(\boldsymbol{s}), \dots, A_{m,1}(\boldsymbol{s}))$
 3:     $\boldsymbol{y} \leftarrow (A_{1,2}(\boldsymbol{s}), \dots, A_{m,2}(\boldsymbol{s}))$
 4:     $\boldsymbol{z} \leftarrow \boldsymbol{x} \odot \boldsymbol{y}$                                  ▷ Element-wise product
 5:     $t \leftarrow \boldsymbol{z} + (A_{1,0}(\boldsymbol{s}), \dots, A_{m,0}(\boldsymbol{s}))$
 6:     return $\boldsymbol{y}, \boldsymbol{t}$
 7: end procedure
```

4.5 Biscuit Key-Generation

The secret-key is a random vector $\boldsymbol{s} \in \mathbb{F}_q^n$ and the public-key is a pair $\big(\boldsymbol{f} = (f_1, \dots, f_m), \boldsymbol{t} = \boldsymbol{f}(\boldsymbol{s})\big) \in \mathbb{F}_q[x_1, \dots, x_n]^m \times \mathbb{F}_q^m$ such that for all $k \in [m]$:

$$f_k(x_1, \dots, x_n) = A_{k,0}(x_1, \dots, x_n) + A_{k,1}(x_1, \dots, x_n) \cdot A_{k,2}(x_1, \dots, x_n),$$

where $A_{1,0}, \dots, A_{m,2} \in \mathbb{F}_q[x_1, \dots, x_n]$ are random affine forms as Equation (1). This implies that the set of polynomials $\boldsymbol{f}$ can be generated from a seed. We summarize the public-key/secret-key generation in Algorithm 11. As the secret key is not very long, we include during key generation the computation of the value $\boldsymbol{y}$ that will be used during signature, and we include it to the private key, along with the value of the public value $\boldsymbol{t}$. If shorter key is mandatory, note that we could also store only seedS, and derive the values $\boldsymbol{s}, \boldsymbol{t}$ and $\boldsymbol{y}$ in the signature procedure.

Algorithm 11 Key Generation

```
Require: $\lambda, q, n, m$ the Biscuit parameters
1: procedure Biscuit.KeyGen( )
2:     $\mathsf{seedF} \xleftarrow{\$} \{0,1\}^\lambda$
3:     $\boldsymbol{f} \leftarrow \texttt{ExpandCircuit}(\mathsf{seedF}, \mathbb{F}_q, n, m)$
4:     $\mathsf{seedS} \xleftarrow{\$} \{0,1\}^\lambda$
5:     $\boldsymbol{s} \leftarrow \texttt{Expand}(\mathsf{seedS}, 1, \mathbb{F}_q, n)$
6:     $\boldsymbol{y}, \boldsymbol{t} \leftarrow \texttt{EvalCircuit}(\boldsymbol{s}, \boldsymbol{f})$
7:     $\mathsf{sk} \leftarrow \mathsf{seedF}, \texttt{pack}(\boldsymbol{s}\|\boldsymbol{t}\|\boldsymbol{y})$
8:     $\mathsf{pk} \leftarrow \mathsf{seedF}, \texttt{pack}(\boldsymbol{t})$
9:     return sk, pk
10: end procedure
```

4.6 Biscuit **Signing Process**

Given a message msg, and a secret-key pk, the detailed signing procedure Biscuit.Sign that follows from the high-level description of Section 3.3 is given in Algorithm 12. It takes as input a message $\mathsf{msg} \in \{0,1\}^*$ and a secret-key sk.

Algorithm 12 Signature

```
Require: $\lambda, \tau, N, q, n, m$ the Biscuit parameters, $C = m$, the number of multiplications occurring in the circuit.
1:  procedure $\mathsf{Biscuit.Sign}(\mathsf{msg}, \mathsf{sk})$
2:      Extract $\mathsf{seedF}, \boldsymbol{s}, \boldsymbol{t}, \boldsymbol{y}$ from $\mathsf{sk}$
3:      $\boldsymbol{f} \leftarrow \mathtt{ExpandCircuit}(\mathsf{seedF}, \mathbb{F}_q, n, m)$
    Phase 1: Committing to the seeds and views of the parties.
4:      $\mathsf{rnd} \xleftarrow{\$} \{0,1\}^{2\lambda}$                ▷ Use $\mathsf{rnd} \leftarrow \emptyset$ for deterministic signature
5:      $\mathsf{salt}, \mathsf{root}^{(1)}, \ldots, \mathsf{root}^{(\tau)} \leftarrow \mathtt{PRF}(\mathsf{rnd} \| \mathsf{sk} \| \mathsf{msg}, 2\lambda + \tau \cdot \lambda)$
6:      for $e \in [\tau]$ do
7:          $\mathsf{seed}^{(e,1)}, \ldots, \mathsf{seed}^{(e,N)} \leftarrow \mathtt{GetSeeds}(\mathsf{root}^{(e)}, \mathsf{salt}, e, N)$
8:          for $i \in [N]$ do
9:              $\mathsf{com}^{(e,i)} \leftarrow \mathtt{Commit}(\mathsf{salt}, e, i, \mathsf{seed}^{(e,i)})$    $\mathsf{tape}^{(e,i)} \leftarrow \mathtt{ExpandTape}(\mathsf{salt}, e, i, \mathsf{seed}^{(e,i)})$
10:             $[\![\boldsymbol{s}]\!]_i^{(e)} \leftarrow \mathtt{Sample}(\mathsf{tape}^{(e,i)}, \mathbb{F}_q, n)$
11:             $[\![\boldsymbol{a}]\!]_i^{(e)} \leftarrow \mathtt{Sample}(\mathsf{tape}^{(e,i)}, \mathbb{F}_q, C)$
12:             $[\![\boldsymbol{c}]\!]_i^{(e)} \leftarrow \mathtt{Sample}(\mathsf{tape}^{(e,i)}, \mathbb{F}_q, C)$
13:         end for
14:         $\boldsymbol{\Delta s}^{(e)} \leftarrow \boldsymbol{s} - \sum_{i=1}^{N} [\![\boldsymbol{s}]\!]_i^{(e)}$
15:         $\boldsymbol{\Delta c}^{(e)} \leftarrow \boldsymbol{y} \odot \sum_{i=1}^{N} [\![\boldsymbol{a}]\!]_i^{(e)} - \sum_{i=1}^{N} [\![\boldsymbol{c}]\!]_i^{(e)}$
16:         $[\![\boldsymbol{s}]\!]_1^{(e)} \leftarrow [\![\boldsymbol{s}]\!]_1^{(e)} + \boldsymbol{\Delta s}^{(e)}$
17:         $[\![\boldsymbol{c}]\!]_1^{(e)} \leftarrow [\![\boldsymbol{c}]\!]_1^{(e)} + \boldsymbol{\Delta c}^{(e)}$
18:         for $i \in [N]$ do
19:             $[\![\boldsymbol{x}]\!]_i^{(e)}, [\![\boldsymbol{y}]\!]_i^{(e)}, [\![\boldsymbol{z}]\!]_i^{(e)} \leftarrow \mathtt{LinearCircuit}([\![\boldsymbol{s}]\!]_i^{(e)}, i, \boldsymbol{t}, \boldsymbol{f})$
20:         end for
21:     end for
22:     $\sigma_1 \leftarrow \big((\mathsf{com}^{(e,i)})_{i=1,\ldots,N} \| \mathtt{pack}(\boldsymbol{\Delta s}^{(e)}) \| \mathtt{pack}(\boldsymbol{\Delta c}^{(e)})\big)_{e=1,\ldots,\tau}$
```

```
**Phase 2**: Challenging the checking protocol
    $h_1 \leftarrow \texttt{H1}(\mathsf{salt}, \mathsf{msg}, \sigma_1)$
    $\boldsymbol{\varepsilon}^{(1)}, \ldots, \boldsymbol{\varepsilon}^{(\tau)} \leftarrow \texttt{Expand}(h_1, \tau, \mathbb{F}_q, C)$
**Phase 3**: Commit to simulation of checking protocol
    **for** $e \in [\tau]$ **do**
        **for** $i \in [N]$ **do**
            $[\![\boldsymbol{\alpha}]\!]_i^{(e)} \leftarrow [\![\boldsymbol{x}]\!]_i^{(e)} \odot \boldsymbol{\varepsilon}^{(e)} + [\![\boldsymbol{a}]\!]_i^{(e)}$
        **end for**
        $\boldsymbol{\alpha}^{(e)} \leftarrow \sum_{i=1}^{N} [\![\boldsymbol{\alpha}]\!]_i^{(e)}$
        **for** $i \in [N]$ **do**
            $[\![\boldsymbol{v}]\!]_i^{(e)} \leftarrow [\![\boldsymbol{y}]\!]_i^{(e)} \odot \boldsymbol{\alpha}^{(e)} - [\![\boldsymbol{z}]\!]_i^{(e)} \odot \boldsymbol{\varepsilon}^{(e)} - [\![\boldsymbol{c}]\!]_i^{(e)}$
        **end for**
    **end for**
    $\sigma_2 \leftarrow \left(\texttt{pack}(([\![\boldsymbol{\alpha}]\!]_i^{(e)})_{i=1,\ldots,N}) \| \texttt{pack}(([\![\boldsymbol{v}]\!]_i^{(e)})_{i=1,\ldots,N})\right)_{e=1,\ldots,\tau}$
**Phase 4**: Challenging the views of the MPC protocol
    $h_2 \leftarrow \texttt{H2}(\mathsf{salt}, h_1, \sigma_2)$
    $\bar{i}_1, \ldots, \bar{i}_\tau \leftarrow \texttt{Expand}(h_2, \tau, [N], 1)$
**Phase 5**: Opening the views of the checking protocol
    **for** $e \in [\tau]$ **do**
        $\mathsf{path}^{(e)} \leftarrow \texttt{GetPath}(\mathsf{root}^{(e)}, \mathsf{salt}, e, \bar{i}_e, N)$
    **end for**
    $\sigma \leftarrow \mathsf{salt} \| h_1 \| h_2 \| \left(\mathsf{path}^{(e)} \| \mathsf{com}^{(e,\bar{i}_e)}\right)_{e=1,\ldots,\tau} \| \texttt{pack}(\left(\boldsymbol{\Delta s}^{(e)} \| \boldsymbol{\Delta c}^{(e)}\right)_{e=1,\ldots,\tau} \| \left( [\![\boldsymbol{\alpha}]\!]_{\bar{i}_e}^{(e)} \right)_{e=1,\ldots,\tau})$
    **return** $\sigma$
**end procedure**
```

In more detail, the signer :

1. First expands the seed from his secret key in order to derive the structured polynomial equations $\boldsymbol{f} \in \mathbb{F}_q[x_1, \ldots, x_n]^m$ as in Definition 1. The signer also gets the secret input as well as the values $\boldsymbol{y}$ of the occurring multiplications.

2. Generates a 2λ bit salt as well as τ root seeds of λ bits. The $\mathsf{root}^{(e)}$ values will be used to derive all randomness in the MPC protocol for each iteration $e \in [\tau]$.

   These values are obtained from a PRF seeded with sk and msg. At this step, it is possible to have a randomized signature by pre-pending a fresh 2λ bits random value rnd, or a deterministic signature by skipping this randomness.

   Note that the values salt and $\mathsf{root}^{(e)}$ may be obtained by any implementation-dependent procedure (e.g. non-deterministic random generator, other PRF function, . . .) without affecting interoperability.

3. Starts the first part of the protocol and repeats the following procedure τ times:

   (a) Derive N seeds $\mathsf{seed}^{(e,i)}$ from $\mathsf{root}^{(e)}$ for $i \in [N]$. Each of them will be used to sample random values for one of the N MPC party.

   (b) Compute a commitment of each of the N seeds and use the seed to derive the party's input secret share $[\![\boldsymbol{s}]\!]_i^{(e)}$. In the same way, we derive random shares $[\![a]\!]_i^{(e)}$ and $[\![c]\!]_i^{(e)}$ as inputs and outputs for the multiplication checking protocol.

(c) For now, only random shares have been distributed to the parties. Let $\boldsymbol{c}^{(e)} = \mathsf{open}\left(\llbracket \boldsymbol{a} \rrbracket^{(e)}\right) \odot \boldsymbol{y}$. To make the sharing correct, we compute $\boldsymbol{\Delta s}^{(e)}$ (reps. $\boldsymbol{\Delta c}^{(e)}$) to correct the first share $\llbracket \boldsymbol{s} \rrbracket_1^{(e)}$ (resp. $\llbracket \boldsymbol{c} \rrbracket_1^{(e)}$) such that when we open $\llbracket \boldsymbol{s} \rrbracket^{(e)}$ (resp. $\llbracket \boldsymbol{c} \rrbracket^{(e)}$) we obtain $\boldsymbol{s}$ (resp. $\boldsymbol{c}^{(e)}$).

(d) Finally, for each party, we can simulate the circuit verification. From $\llbracket \boldsymbol{s} \rrbracket_i^{(e)}$, the party evaluates the circuit. Each time a multiplication should be performed, the input is written in $\llbracket \boldsymbol{x} \rrbracket_i^{(e)}$ and $\llbracket \boldsymbol{y} \rrbracket_i^{(e)}$. The expected output of the multiplications is computed backward from $\boldsymbol{t}$ and $\llbracket \boldsymbol{s} \rrbracket_i^{(e)}$.

4. The first phase is done. Prepares in σ_1 the values that should be checked by a verifier, namely, the commitments on all N seeds, and the correcting values $\boldsymbol{\Delta s}^{(e)}$, and $\boldsymbol{\Delta c}^{(e)}$.

5. σ_1 is hashed together with the salt and the message to produce h_1 that will be used to generate the challenges for the checking protocol.

6. Use the challenges for the second part of the protocol. We repeat τ times the following procedure:

   (a) Compute for all parties compute the values $\llbracket \boldsymbol{\alpha} \rrbracket_i^{(e)}$ that will be broadcast.

   (b) Open $\boldsymbol{\alpha}^{(e)}$ and compute the value $\llbracket \boldsymbol{v} \rrbracket_i^{(e)}$ for the verification.
   At this point, it holds that $\mathsf{open}(\llbracket \boldsymbol{v} \rrbracket^{(e)}) = \mathbf{0}$.

7. The third phase is done. Prepare in σ_2 the values that would have been sent to the verifier, namely, the N shares of $\boldsymbol{\alpha}^{(e)}$ and the N shares of $\mathbf{e}$.

8. σ_2 is hashed together with the salt and the h_1 to produce h_2 that will be used to generate the challenges for opening the views.

9. For all iterations, compute the seed generation path $\mathsf{path}^{(e)}$ that allows to recover all seeds except the $\bar{i}_e$-th one.

10. Finally, output in the signature the paths, the commitment values for $\bar{i}_e$ and the missing share of $\boldsymbol{\alpha}^{(e)}$ as well as the correcting values $\boldsymbol{\Delta s}^{(e)}$ and $\boldsymbol{\Delta c}^{(e)}$.

4.7 Biscuit Verification process

Given a message msg, a signature sig and a public-key pk, the detailed verification process corresponding to the high-level description Section 3.3 is given in Algorithm 13. The verification process is very similar to the signature process as the verifier has to replay the MPC protocol for each of the τ participants except one.

Algorithm 13 Verification

```
Require: $\lambda, \tau, N, q, n, m$ the Biscuit parameters, $C = m$, the number of multiplications occurring in the circuit.
1: procedure Biscuit.Verify(sig, msg, pk)
2:     Extract seedF, $\boldsymbol{t}$ from pk
3:     $\boldsymbol{f} \leftarrow$ ExpandCircuit(seedF, $\mathbb{F}_q, n, m$)
4:     Extract salt, $h_1, h_2$ from sig
5:     $\boldsymbol{\varepsilon}^{(1)}, \ldots, \boldsymbol{\varepsilon}^{(\tau)} \leftarrow$ Expand($h_1, \tau, \mathbb{F}_q, C$)
6:     $\bar{i}_1, \ldots, \bar{i}_\tau \leftarrow$ Expand($h_2, \tau, [N], 1$)
7:     for $e \in [\tau]$ do
8:         Extract path$^{(e)}$, com$^{(e,\bar{i}_e)}$ from sig
9:         Extract $\boldsymbol{\Delta s}^{(e)}, \boldsymbol{\Delta c}^{(e)}$ from sig
10:        seed$^{(e,1)}, \ldots,$ seed$^{(e,N)} \leftarrow$ GetPathSeeds(path$^{(e)}$, salt, $e, \bar{i}_e, N$)
11:        for $i \in [N] \setminus \{\bar{i}_e\}$ do
12:            com$^{(e,i)} \leftarrow$ Commit(salt, $e, i$, seed$^{(e,i)}$)     tape$^{(e,i)} \leftarrow$ ExpandTape(salt, $e, i$, seed$^{(e,i)}$)
13:            $[\![\boldsymbol{s}]\!]_i^{(e)} \leftarrow$ Sample(tape$^{(e,i)}, \mathbb{F}_q, n$)
14:            $[\![\boldsymbol{a}]\!]_i^{(e)} \leftarrow$ Sample(tape$^{(e,i)}, \mathbb{F}_q, C$)
15:            $[\![\boldsymbol{c}]\!]_i^{(e)} \leftarrow$ Sample(tape$^{(e,i)}, \mathbb{F}_q, C$)
16:            if $i \overset{?}{=} 1$ then
17:                $[\![\boldsymbol{s}]\!]_1^{(e)} \leftarrow [\![\boldsymbol{s}]\!]_1^{(e)} + \boldsymbol{\Delta s}^{(e)}$
18:                $[\![\boldsymbol{c}]\!]_1^{(e)} \leftarrow [\![\boldsymbol{c}]\!]_1^{(e)} + \boldsymbol{\Delta c}^{(e)}$
19:            end if
20:            $[\![\boldsymbol{x}]\!]_i^{(e)}, [\![\boldsymbol{y}]\!]_i^{(e)}, [\![\boldsymbol{z}]\!]_i^{(e)} \leftarrow$ LinearCircuit($[\![\boldsymbol{s}]\!]_i^{(e)}, \boldsymbol{t}, i, \boldsymbol{f}$)
21:        end for
22:        Extract $[\![\boldsymbol{\alpha}]\!]_{\bar{i}_e}^{(e)}$ from sig
23:        for $i \in [N] \setminus \{\bar{i}_e\}$ do
24:            $[\![\boldsymbol{\alpha}]\!]_i^{(e)} \leftarrow [\![\boldsymbol{x}]\!]_i^{(e)} \odot \boldsymbol{\varepsilon}^{(e)} + [\![\boldsymbol{a}]\!]_i^{(e)}$
25:        end for
26:        $\boldsymbol{\alpha}^{(e)} \leftarrow \sum_{i=1}^{N} [\![\boldsymbol{\alpha}]\!]_i^{(e)}$
27:        for $i \in [N] \setminus \{\bar{i}_e\}$ do
28:            $[\![\boldsymbol{v}]\!]_i^{(e)} \leftarrow [\![\boldsymbol{y}]\!]_i^{(e)} \odot \boldsymbol{\alpha}^{(e)} - [\![\boldsymbol{z}]\!]_i^{(e)} \odot \boldsymbol{\varepsilon}^{(e)} - [\![\boldsymbol{c}]\!]_i^{(e)}$
29:        end for
30:        $[\![\boldsymbol{v}]\!]_{\bar{i}_e}^{(e)} \leftarrow -\sum_{i=1, i \neq \bar{i}_e}^{N} [\![\boldsymbol{v}]\!]_i^{(e)}$
31:    end for
32:    $\sigma_1 \leftarrow \big((\mathsf{com}^{(e,i)})_{i=1,\ldots,N} \| \mathtt{pack}(\boldsymbol{\Delta s}^{(e)}) \| \mathtt{pack}(\boldsymbol{\Delta c}^{(e)})\big)_{e=1,\ldots,\tau}$
33:    $h_1' \leftarrow$ H1(salt, msg, $\sigma_1$)
34:    $\sigma_2 \leftarrow \Big(\mathtt{pack}(([\![\boldsymbol{\alpha}]\!]_i^{(e)})_{i=1,\ldots,N}) \| \mathtt{pack}(([\![\boldsymbol{v}]\!]_i^{(e)})_{i=1,\ldots,N})\Big)_{e=1,\ldots,\tau}$
35:    $h_2' \leftarrow$ H2(salt, $h_1, \sigma_2$)
36:    if $(h_1 \overset{?}{=} h_1')$ and $(h_2 \overset{?}{=} h_2')$ then
37:        return Valid
38:    end if
39:    return Invalid
40: end procedure
```

In more detail, the verifier :

1. Recovers the expected challenges $\varepsilon^{(e)}\ell$ and $\bar{i}_e$ by expanding the value. h_1 and h_2 of the signature.
2. Iterates the following procedure τ times:
   (a) Derive N seeds $\mathsf{seed}^{(e,i)}$ from $\mathsf{path}^{(e)}$ for $i \in [N]$ except $\bar{i}_e$. Each of them will be used to recover the random values for one of the N `MPC` party.
   (b) Replay the signature process for all opened N `MPC` party
   (c) Compute the commitment for each of the $N-1$ seeds and use the seed to derive the party's input secret share $[\![\boldsymbol{s}]\!]_i^{(e)}$, and the random shares $[\![a]\!]_i^{(e)}$ and $[\![c]\!]_i^{(e)}$ for the multiplication checking protocol. For participant number 1, correct the share $[\![\boldsymbol{s}]\!]_1^{(e)}$ and $[\![\boldsymbol{c}]\!]_1^{(e)}$ using the $\boldsymbol{\Delta s}^{(e)}$ and $\boldsymbol{\Delta c}^{(e)}$ from the signature.
   (d) Finally, for each party, simulate the circuit verification. From $[\![\boldsymbol{s}]\!]_i^{(e)}$, the party evaluates the circuit. Each time a multiplication should be performed, the input is written in $[\![\boldsymbol{x}]\!]_i^{(e)}$ and $[\![\boldsymbol{y}]\!]_i^{(e)}$. The expected output of the multiplications is computed backward from $\boldsymbol{t}$ and $[\![\boldsymbol{s}]\!]_i^{(e)}$.
   (e) Start the multiplication checking protocol by recovering the missing $[\![\boldsymbol{\alpha}]\!]_{\bar{i}_e}^{(e)}$ from the signature, and computing the other $[\![\boldsymbol{\alpha}]\!]_i^{(e)}, i \neq \bar{i}_e$ in order to recover $\boldsymbol{\alpha}^{(e)}$. Then use this value to compute the verification value $[\![\boldsymbol{v}]\!]_i^{(e)}, , i \neq \bar{i}_e$. The missing shares $[\![\boldsymbol{v}]\!]_{\bar{i}_e}^{(e)}$ is recovered from the others by supposing that the sum of the $[\![\boldsymbol{v}]\!]_i^{(e)}$ should be 0.
3. Checks that all opened seeds were correct by checking h_1.
4. Checks that multiplications were correct by checking h_2.

4.8 Performances and Memory Consumption

In this section, we show the performance and memory consumption of our instances. We present two implementations: one generic code optimized for generic little-endian 64-bit CPU, and one using vectorized instructions.

The code is compiled with `GCC` version 10.2.1 on Debian GNU/Linux. Number of cycles was measured by counting `PERF_HW_COUNT_CPU_CYCLES` events on an `11th Gen Intel(R) Core(TM) i7-1185G7 @ 3.00GHz` CPU. Even if frequency modification should not affect this metric, we deactivated Intel's TurboBoost feature anyway. The number of cycles is averaged over 1000 executions.

Table 5 and Table 6 give the figures for the generic 64-bit implementation `generic64`. Table 7 and Table 8 give the figures for the `sse2` optimized implementation. For this implementation, the code is the same as the `sse2` implementation except that we work with native 256-bit integer type and the compiler automatically issues vectorized instructions.

Several points can be noted:

- RAM consumption of the verification procedure is significantly lower than the signature. Indeed, the signature procedure can perform both parts of the algorithm independently for each iteration. We do not have to keep in memory all the intermediate values, only the one relative to the current iteration. This saves approximately a factor τ

- The `sse2` optimized implementation uses more RAM. Indeed, the granularity of the inner datatype is now 256-bits instead of 64-bits. Every vector element has to be encoded in a multiple of 256-bits element.

Name	Memory (bytes)			Performance (cycles)		
	`keygen`	`sign`	`verify`	`keygen`	`sign`	`verify`
`biscuit128s`	448	1 093 072	83 520	110 419	106 256 349	105 028 599
`biscuit192s`	448	2 245 344	106 896	274 275	775 763 124	770 260 803
`biscuit256s`	496	3 982 752	148 000	417 886	1 453 937 743	1 462 615 033

Table 5: Time performance and memory consumption of Biscuit-short on `generic64` impl.

Name	Memory (bytes)			Performance (cycles)		
	`keygen`	`sign`	`verify`	`keygen`	`sign`	`verify`
`biscuit128f`	448	137 040	14 400	108 608	12 991 223	11 995 363
`biscuit192f`	448	265 632	20 496	256 728	87 269 078	81 529 919
`biscuit256f`	496	477 120	32 800	419 094	167 453 309	155 923 813

Table 6: Time performance and memory consumption of Biscuit-fast on `generic64` impl.

Name	Memory (bytes)			Performance (cycles)		
	`keygen`	`sign`	`verify`	`keygen`	`sign`	`verify`
`biscuit128s`	512	1 657 200	125 392	82 632	80 555 671	78 899 797
`biscuit192s`	512	2 868 336	135 920	210 159	724 466 241	714 947 231
`biscuit256s`	512	3 982 800	148 016	332 165	1 291 111 734	1 274 568 395

Table 7: Time performance and memory consumption of Biscuit-short on `sse2` impl.

Name	Memory (bytes)			Performance (cycles)		
	`keygen`	`sign`	`verify`	`keygen`	`sign`	`verify`
`biscuit128f`	512	207 728	21 712	82 505	9 653 412	8 734 302
`biscuit192f`	512	339 504	26 480	210 150	81 492 308	75 826 788
`biscuit256f`	512	477 168	32 848	353 223	147 099 575	137 359 832

Table 8: Time performance and memory consumption of Biscuit-fast on `sse2` impl.

5 Security Analysis of Biscuit

This part is dedicated to the security analysis of Biscuit against *key-recovery* and *forgery* attacks. Let $\big(\boldsymbol{f} = (f_1, \ldots, f_m), \boldsymbol{t} = \boldsymbol{f}(\boldsymbol{s})\big) \in \mathbb{F}_q[x_1, \ldots, x_n]^m \times \mathbb{F}_q^m$ be a Biscuit public-key and $\boldsymbol{s} \in \mathbb{F}_q^n$ be the corresponding secret-key (Section 4.5).

A key-recovery attack against Biscuit consists of solving the system defined by

$$\boldsymbol{t} = \boldsymbol{f}(\boldsymbol{x}), \text{ with } \boldsymbol{x} = (x_1, \ldots, x_n). \tag{2}$$

In Section 5.1, we review generic algorithms for solving $\texttt{MQ}_q$ in the classical (Section 5.1.1 and Section 5.1.2) and quantum (Section 5.1.3) settings. Then, we introduce in Section 5.1.2 the tool we used to derive the parameters of Biscuit: `MQEstimator` [10]. Section 5.2 presents more specific results on the hardness of `PowAff2`. Namely, we justify and introduce the assumption about the semi-regularity of `PowAff2` instances (Section 5.2.1). A new algorithm for solving `PowAff2` that exploits the specific structure of the equations is presented in Section 5.2.2.

A forgery attack consists of finding message msg and a valid signature σ, where msg was not signed by the holder of the secret key. A forgery attack against Biscuit can be made by solving a subset of the equations (2) of size $m - u$, where u is a parameter to optimize, as explained in Section 5.3.

Then, we present in Section 5.4 the parameters used for Biscuit, as well as the corresponding security levels, that result from the analysis of this section. Section 5.5 concludes this part and discusses the `EUF-CMA` security of Biscuit. Finally, Section 5.6 is about side channel attacks.

5.1 Generic Algorithms for Solving $\texttt{MQ}_q$

5.1.1 Complexity of computing Gröbner bases

Gröbner basis [19, 18] is the classical technique for solving $\texttt{MQ}_q$. The historical method for computing such bases – known as Buchberger's algorithm – has been introduced by B. Buchberger in his PhD thesis [19, 18]. Many improvements on Buchberger's algorithm have been done leading – in particular – to more efficient algorithms such as the F_4 and F_5 algorithms of J.-C. Faugère [34, 35].

The fundamental conceptual breakthrough that leads from the historical Buchberger's algorithm to Faugère's algorithms is the intensive use of linear algebra. The bridge has been established by D. Lazard [56] who proved that computing a Gröbner basis for a system of homogeneous polynomials $f_1, \ldots, f_m \in \mathbb{F}_q[x_1, \ldots, x_n]$ is equivalent to perform Gaussian elimination on so-called *Macaulay matrices.* Given such a set of homogeneous polynomials $f_1 \ldots, f_m$, the associated Macaulay matrix $\mathcal{M}^{\text{acaulay}}_{D,m}(f_1 \ldots, f_m)$ of degree $D \geq \min\big(\deg(f_1), \ldots, \deg(f_m)\big)$ is defined as the coefficient matrix of $(t_{i,j} \cdot f_i)$ where $i \in [m]$ and $t_{i,j}$ runs through all monomials of degree $D - \deg(f_i)$.

Theorem 1 ([56])**.** *Let $f_1, \ldots, f_m \in \mathbb{F}_q[x_1, \ldots, x_n]$ be homogeneous polynomials. There exists a positive integer D_0 for which a row echelon computation on all $\mathcal{M}^{\text{acaulay}}_{D,m}(f_1, \ldots, f_m)$ matrices for* $\min\big(\deg(f_1), \ldots, \deg(f_m)\big) \leq D \leq D_0$ *computes a* DRL-*Gröbner basis of $\langle f_1, \ldots, f_m \rangle$.*

It is clear that the complexity of computing a Gröbner basis will be related to the maximum degree reached in Theorem 1 which corresponds to the *degree of regularity.*

Definition 2. *Given a positive integer D, we denote by R_D (resp. $\mathbb{F}_q[x_1,\ldots,x_n]_D$) the $\mathbb{F}_q$-vector spaces of homogeneous polynomials of degree D in $R = \mathbb{F}_q[x_1,\ldots,x_n]/\langle x_1^q - x_1,\ldots,x_n^q - x_n\rangle$ (resp. $\mathbb{F}_q[x_1,\ldots,x_n]$). $\dim_{\mathbb{F}_q}(R_d)$ represents the number of monomials in $\mathbb{F}_q[x_1,\ldots,x_n]$ of degree D. The degree of regularity of a sequence of homogeneous polynomials $f_1,\ldots,f_m \in \mathbb{F}_q[x_1,\ldots,x_n]$, denoted by $D_{\text{reg}}(f_1,\ldots,f_m)$, is the minimum integer D_0, if any, such that $\dim_{\mathbb{F}_q}(I_{D_0}) = \dim_{\mathbb{F}_q}(R_{D_0})$, where $\mathcal{I}_d = R_{D_0} \cap \mathcal{I}$, and R_{D_0} is the set of elements in R of degree D_0.*

Remark 1. R_D as defined in Definition 2 implicitly assumes that the field equations $x_1^q - x_1,\ldots,x_n^q - x_n$ are always added to the generator of the ideals. This is due to the fact that we are only looking for solutions over the base field.

Definition 3. *For non-homogeneous polynomials $f_1,\ldots,f_m \in \mathbb{F}_q[x_1,\ldots,x_n]$, the degree of regularity is defined (see [4, 8]) from the homogeneous components $f_1^h,\ldots,f_m^h \in \mathbb{F}_q[x_1,\ldots,x_n]$ of highest degree of the polynomials $f_1,\ldots,f_m$. We have then $D_{\text{reg}}(f_1,\ldots,f_m) = D_{\text{reg}}(f_1^h,\ldots,f_m^h)$.*

Once this notation is fixed, we can rather easily establish a rough upper bound on the cost of computing a Gröbner basis (see [58, 56, 4, 8, 6]).

Theorem 2. *Let $f_1,\ldots,f_m \in \mathbb{F}_q[x_1,\ldots,x_n]$ be such that the system of homogeneous components of highest degree $f_1^h,\ldots,f_m^h$ is a zero-dimensional and $D_{\text{reg}} = D_{\text{reg}}(f_1,\ldots,f_m)$. We can compute a* DRL-*Gröbner basis of $\langle f_1,\ldots,f_m\rangle$ in :*

$$O\left(m \cdot \binom{n + D_{\text{reg}}}{D_{\text{reg}}}^{\omega}\right) \textit{ arithmetic operations over } \mathbb{F}_q. \tag{3}$$

More precise statements about the number of arithmetic operations performed in F$_5$ can be found in [4, 6]. The complexity of computing a Gröbner basis is exponential in the degree of regularity. Unfortunately, this degree of regularity is difficult to compute in general (as difficult as computing the Gröbner basis). There is a particular class of systems for which this degree can be computed efficiently: *regular and semi-regular sequences.*

Regular sequences. The notion of regular sequences is classical (see [58, 59]), but restricted to $m \le n$. Such constraint typically excludes the use of field equations.

For a reason that will become clear later, we state the next results for a general field $\mathbb{F}$.

Definition 4. *The sequence $f_1,\ldots,f_m \in \mathbb{F}[x_1,\ldots,x_n]$ of homogeneous polynomials is regular if:*

1. *$\langle f_1,\ldots,f_m\rangle \neq \mathbb{F}[x_1,\ldots,x_n]$,*
2. *for all $i \in [m]$ and $g \in \mathbb{F}[x_1,\ldots,x_n]$: $g \cdot f_i \in \langle f_1,\ldots,f_{i-1}\rangle \Rightarrow g \in \langle f_1,\ldots,f_{i-1}\rangle$.*

Now, let $f_1,\ldots,f_m \in \mathbb{F}[x_1,\ldots,x_n]$ be affine polynomials of degrees $d_1,\ldots,d_m$ respectively. The sequence $f_1,\ldots,f_m$ is regular if the sequence $f_1^h,\ldots,f_m^h$ is regular.

The algebraic behavior (i.e. degree of regularity, Hilbert series, ...) of regular sequences is then well understood.

Property 1 ([4, 58, 56])**.** *Let $f_1,\ldots,f_m \in \mathbb{F}[x_1,\ldots,x_n]$ be a regular sequence of homogeneous polynomials of degree $d_1,\ldots,d_m$ respectively. Then, it holds that:*

- *The degree of regularity of a regular sequence is given by the so-called* Macaulay bound*, i.e.:*

$$\sum_{i=1}^{m}(d_i-1)+1.$$

- *The Hilbert series of $\mathcal{I}=\langle f_1,\dots,f_m\rangle$ is*

$$\sum_{d\geq 0}\dim_{\mathbb{F}}\left(R_d/\mathcal{I}_d\right)z^d=\frac{(1-z^{d_i})^m}{(1-z^n)},$$

  where $R=\mathbb{F}[x_1,\dots,x_n]$ and $R_d,\mathcal{I}_d$ are defined similarly than in Definition 2.

We will now prove that the instances of `PowAff2` are regular. By definition, this reduces to consider the homogeneous components of the highest degree of the polynomials considered in the `PowAff2` problem (Definition 1), namely we consider the sequence :

$$f_k^h=\left(\sum_{i=1}^{n}a_i^{(k,1)}x_i\right)\left(\sum_{i=1}^{n}a_i^{(k,2)}x_i\right),\text{ with } a_1^{(k,1)},\dots,a_n^{(k,2)}\in\mathbb{F},\forall k\in[m]. \tag{4}$$

The precise statement is as follows:

Theorem 3. *Let $f_1^h,\dots,f_m^h\in\mathbb{F}[x_1,\dots,x_n]$ be defined as in (4) and $h=n-m$. There exist $\lambda_{i,j}\in\mathbb{F}$ such that the sequence $g_1=f_1^h, g_2=f_2^h+\sum_{k=3}^{m}\lambda_{2,k}f_k^hg_k, g_3=f_3+\sum_{k=4}^{m}\lambda_{3,k}f_k^hg_k,\dots,g_h=f_h^h+\sum_{k=h+1}^{m}\lambda_{h,k}f_k^hg_k, g_{h+1}=f_{h+1}^h,\dots,g_m=f_m^h\in\mathbb{F}[x_1,\dots,x_n]$ is such that :*

- *$g_1,\dots,g_m$ generates the same ideal than $f_1^h\dots,f_m^h$, and*
- *$g_1,\dots,g_m$ is a regular sequence.*

These properties hold for all $\lambda_{i,j}\in\mathbb{F}$ except for finitely many values.

Theorem 3 is a particular case of a more general result demonstrated, for instance in [4, Proposition 1.7.3, page 23], for any sequence of polynomials. These results are typically stated for characteristic zero as they implicitly rely on the *Zariski topology*. It is the standard topology used in algebraic geometry where topology, *closed sets* are the algebraic sets. This requires to be adapted in finite fields, typically by using Schwartz-Zippel-DeMillo-Lipton lemma [28, 69, 65] (see discussion in Remark 2).

We conclude by providing experimental results illustrating Theorem 3. Table 9, we compare the degree of regularity D_{reg} of a system as in (6) with the theoretical degree of regularity of a regular sequence of quadratic polynomials (that is, $m+1$).

Semi-regular sequences. Semi-regular sequences have been introduced in [4, 8] to deal with over-defined systems.

Definition 5 (Semi-regular sequences, [10])**.** *A homogeneous sequence of quadratic polynomials $f_1,\dots,f_m\in\mathbb{F}_q[x_1,\dots,x_n]$ is called semi-regular if $m>n$ and*

$$\sum_{d\geq 0}\dim_{\mathbb{F}_q}\left(R_d/\mathcal{I}_d\right)z^d=\left[\frac{(1-z^q)^n}{(1-z)^n}\left(\frac{1-z^2}{1-z^{2q}}\right)^m\right]_+, \tag{5}$$

$m = m$	q	d	$m+1$	D_{reg}
10	2^8	2	11	11
11	2^8	2	12	12
12	2^8	2	13	13
13	2^8	2	14	14
14	2^8	2	15	15

Table 9: Theoretical degree of regularity vs experimental degree of regularity D_{reg} computed with Magma [15].

where $\mathcal{I} = \langle f_1, \dots, f_m \rangle$ and $[H(z)]_+$ means that the series $H(z)$ is truncated from the first non-positive coefficient.

This definition assumes that the field equations are always added to the sequence of quadratic polynomials. It also allows computing explicitly the degree of regularity of semi-regular sequences by expanding the power series (5) for specific values of m, n and q. We can also have asymptotic information about the trend of regularity. For instance :

Theorem 4 ([4, 5, 8])**.** *Let $k \geq 0$ be a constant. The degree of regularity of a semi-regular sequence $f_1, \dots, f_{n+k} \in \mathbb{F}_q[x_1, \dots, x_n]$ of quadratic polynomials behaves asymptotically as*

$$\frac{n}{2} + \alpha_k \sqrt{\frac{n}{6}} + o(\sqrt{n})$$

where α_k is the biggest root of k-th Hermite polynomial.

Remark 2. A fundamental question in algebraic geometry is whether semi-regular sequences as defined in Definition 5 indeed exists. For regular sequences ($m \leq n$), the question is solved and well understood [39]. The question remains open in the semi-regular case ($m > n$). A famous conjecture of algebraic geometry is then attached to the existence of semi-regular sequences: the so-called *Fröberg conjecture* [39]. The conjecture states that semi-regular sequences form a dense subset among the set of all sequences. This is equivalent to proving that there exists a *non-constant* polynomial F that vanishes the coefficients of non-semi-regular sequences. For semi-regular sequences, it is not difficult to find such a polynomial. However, the delicate point is to prove that the polynomial is not zero on the coefficients of a least one sequence of m polynomials. To prove Fröberg's conjecture, it is then sufficient to demonstrate that one particular family of $m > n$ polynomials in $\mathbb{F}_q[x_1, \dots, x_n]$ is semi-regular for any sufficiently big n and any $m > n$. In finite fields, Zariski's topology is meaningless since all sets are algebraic. However, the proof strategy is essentially similar. Given the non-constant polynomial F, we can use Schwartz-Zippel-DeMillo-Lipton lemma [28, 69, 65] to upper bound the probability that F vanishes; that is the probability that a random sequence is not semi-regular.

5.1.2 Estimating the bit-complexity of $\texttt{MQ}_q$ with the `MQEstimator`

Besides the Gröbner bases algorithms discussed in Section 5.1.1, there exists a considerable amount of algorithms solving the $\texttt{MQ}_q$ problems (see, for instance, [10] for an overview). The most relevant algorithms include:

- **Algorithms for underdetermined systems.** A system of equations is said to be *underdetermined* if it has more unknowns than equations, that is, in our notation, $m < n$. A first naive approach consists in fixing the values of $n - m$ unknowns and then solving the corresponding square system of equations. When the system is extremely underdetermined, i.e. m much smaller than n, then improved algorithms have been proposed, e.g. Kipnis, Patarin, and Goubin [55], Thomae and Wolf [66] and more recently Furue, Nakamura, and Takagi [40].

- **Fast exhaustive search.** In [17, 16], the authors proposed a more efficient way to perform an exhaustive search over $\mathbb{F}_2^n$ by enumerating the solution space with Gray codes. The time complexity of finding one solution to the $\texttt{MQ}_2$ problem with this algorithm is given by $4\log(n)2^n$. This approach can be extended to any field $\mathbb{F}_q$ using lexicographical ordering instead of Gray codes [41] with time complexity is given by $\mathcal{O}(dq^n)$.

- **Gröbner basis-based algorithms.** In addition to the Buchberger's, F_4 and F_5 algorithms mentioned in Section 5.1.1, there are many different approaches for computing Gröbner bases. Typically, the popular XL algorithm [27] was later proved to be a redundant variant of the F_4 algorithm [3]. More recently, a zoo of algorithms such as G2V [42], GVW [43], etc, flourished building on the core ideas of F_4 and F_5. This literature is vast, and we refer to [32] for a survey of these algorithms. In fact, [32] introduced a new general algorithmic framework – called RB – that includes as a special version many algorithms such as F_5, G2V, GVW, etc.

- **Hybrid approaches.** In a series of papers, e.g. [13, 12, 7, 51], the authors describe hybrid techniques which combine exhaustive search and a Gröbner basis-like computations; leading to asymptotically fast algorithms, e.g. Hybrid-(F_4/F_5) and Crossbed algorithms. The efficiency of such approaches is related to the choice of a *trade-off* between these two methods. We emphasize that all the complexity results for such hybrid techniques are obtained assuming a natural algebraic hypothesis, roughly all sub-systems arising from the computations are semi-regular (Definition 5).

- **Probabilistic algorithms.** In [57], Loskshtanov et al. were the first to introduce a *probabilistic algorithms* that, in the worst case, solves $\texttt{MQ}_q$ in time $\tilde{\mathcal{O}}(q^{\delta n})$, for some $\delta < 1$ depending only on q and the degree of the system, without relying on any unproven assumption. The approach Lokshtanov et al., was then improved by Björklund et al. in [14] and Dinur [30] for solving $\texttt{MQ}_2$ and leading to the current fastest asymptotic algorithm whose complexity is $\tilde{\mathcal{O}}\left(2^{0.6943n}\right)$.

In order to ease the security analysis of $\texttt{MQ}_q$-based, R. Makarim, C. Sanna, and J. Verbel [10] introduced a software tool called `MQEstimator`, that permits to derive the security level for solving a given $\texttt{MQ}_q$ instance taking into account all possible known attacks. A summary of the `MQEstimator` library is given in Table 10. Note that `MQEstimator` is part of the more general library, `CryptographicEstimators` [33], that provides estimators for a large variety of cryptographic hard computational problems[1].

The time and space complexities of each algorithm are given, assuming a computational model in which the operations of $\mathbb{F}_q$ (addition, multiplication, and division) are performed in constant time $\mathcal{O}(1)$ and in which every element of $\mathbb{F}_q$ is stored in constant space $\mathcal{O}(1)$ ($\mathbb{F}_q$-complexity). A more detailed analysis could assume a computational model in which the operations at the bit-level are performed in constant time $\mathcal{O}(1)$ and in which every bit is stored in constant space $\mathcal{O}(1)$

[1]The code is accessible at https://github.com/Crypto-TII/CryptographicEstimators

Multivariate Quadratic Estimator	
Name	`MQEstimator`
Parameters	(n, m, q): No. of variables, No. of equations, field size
Elementary operation	$\mathbb{F}_q$ multiplication
Memory unit	$\mathbb{F}_q$ element
Bit complexity factor time	$(\log_2 q)^\theta$, where $\theta \geq 0$
Bit complexity factor memory	$\log_2 q$
No. of algorithms	12

Table 10: Overview of the $\texttt{MQ}_q$ estimator

(bit-complexity). However, the bit-complexities of addition, multiplication, and division in $\mathbb{F}_q$ are different (with the addition being the least expensive and division the most) and depend on the algorithms implementing them, possible hardware optimizations, and eventually, q having a special form, like q being equal to a power of 2 or a Mersenne prime (see [45] for a survey). Therefore, there is no straightforward way to convert between $\mathbb{F}_q$-time complexity and bit-time complexity. Roughly, the bit-time complexity can be estimated by $(\log q)^\theta$ times the $\mathbb{F}_q$-time complexity, with $\theta \in [1, 2]$. On the other hand, the bit-space complexity is simply equal to $\log q$ times the $\mathbb{F}_q$-space complexity. Of course, for $q = 2$ the $\mathbb{F}_q$-complexity and the bit-complexity are equivalent.

5.1.3 Quantum algorithms

`MQEstimator` and the algorithms discussed before are classical techniques. Few quantum algorithms have been developed in the past years for solving $\texttt{MQ}_q$.

- **Quantum exhaustive search.** The first quantum algorithm for $\texttt{MQ}_2$ is due to P. Schwabe and B. Westerbaan in [64]. The authors described a quantum version of exhaustive search using Grover's algorithm [47]. Precise resource estimates for their algorithms are derived, demonstrating that a quantum computer can solve m binary quadratic equations in n binary variables using $\mathcal{O}(m+n)$ qubits and requiring the evaluation of $\mathcal{O}\big(mn^2 2^{n/2}\big)$ quantum gates. The authors also describe a variant using fewer qubits, i.e. $\mathcal{O}\big(n + \log_2(m)\big)$ but with twice as many quantum gates as the first approach.

- **GroverXL.** In [11], the authors proposed a quantized version of XL. More precisely, the authors considered a variant of FXL [27, 67], that combines XL with an exhaustive search. The principle of GroverXL is to combine FXL with Grover's algorithm. GroverXL solves $\texttt{MQ}_q$ with time complexity $2^{\left(t+o(1)\right)n}$ on a mesh-connected computer of area $2^{\left(a+o(1)\right)n}$. The values of t and a can be explicitly computed for given parameters of the $\texttt{MQ}_q$ instance considered.

- **QuantumMQSolve.** In a parallel but independent work from GroverXL, the authors of [36] propose a slightly different approach for combining quantum algorithms and Gröbner bases. To date, this is the fastest quantum algorithm known for solving $\texttt{MQ}_q$. Let $m = \lceil \alpha\, n \rceil$, with $\alpha \geq 1$. It holds that:

  – For $q = 2$, QuantumMQSolve has an average complexity of $\mathcal{O}(2^{\left(\frac{1}{2} - 0.0375\alpha\right)n})$. In particular, this gives $\mathcal{O}(2^{0.462n})$ for $\alpha = 1$.

– When $n = m \to \infty, q \to \infty$ and $\log_2(q) \ll n$, QuantumMQSolve has an average complexity of $\mathcal{O}(2^{n\left(2.76 - \frac{1.26}{\log_2(\sqrt{q})}\right)})$.

The complexities provided below are asymptotic and not precise for given parameters (q, n, m) . But the cost of QuantumMQSolve can be also explicitly computed by minimizing the tradeoff and using a more precise form of the complexity.

- **Quantum Gröbner bases algorithms.** In [25, 26], the authors proposed a new approach for solving $\mathtt{MQ}_2$ and more generally $\mathtt{MQ}_q$ in the quantum setting. The central idea of these algorithms is to use the quantum algorithm due to A. Harrow, A. Hassidim and S. Lloyd (HLL) for solving linear systems [48]. Ding et al [29] provided an improved complexity analysis of [25, 26] and concludes that quantum exhaustive search will be almost always faster than the HLL approach for $\mathtt{MQ}_2$ by Chen and Gao [25]. The authors of [29] also proposed an improved variant of the Chen and Gao algorithm [25] for $\mathtt{MQ}_2$ whose complexity is exponential in the Hamming weight. In particular, this version beats the quantum exhaustive search when the Hamming weight of the solution is logarithmic in the number of variables.

5.2 Hardness Analysis of PowAff2

5.2.1 On the genericity of PowAff2

The fundamental assumption on which Biscuit relies is that random instances of PowAff2 behave such as semi-regular sequences (Definition 5). We formalize this statement below.

Assumption 1. *Let $\boldsymbol{s} \in \mathbb{F}_q^n, A_{1,0}, \ldots, A_{m,2} \in \mathbb{F}_q[x_1, \ldots, x_n]$ be randomly sampled affine forms, i.e. $\forall i \in [m]$ and $\forall j, 0 \le j \le 2$ $A_{i,j} = a_0^{i,j} + \sum_{k=1}^{n} a_k^{(i,j)} x_k$, with $a_0^{(i,j)}, \ldots, a_n^{(i,j)} \in \mathbb{F}_q$. Let then:*

$$f_i(x_1, \ldots, x_n) = A_{i,0}(x_1, \ldots, x_n) + A_{i,1}(x_1, \ldots, x_n) A_{i,2}(x_1, \ldots, x_n), \forall i \in [m],$$

and $\boldsymbol{t} = \boldsymbol{f}(\boldsymbol{s}) = (t_1, \ldots, t_m) \in \mathbb{F}_q^m$. For any constant $k > 0$, the sequence $f_1(x_1, \ldots, x_n) - t_1, \ldots, f_{n+k}(x_1, \ldots, x_n) - t_{n+k} \in \mathbb{F}_q[x_1, \ldots, x_n]$ is semi-regular.

Remark 3. The use of over-defined systems of equations is mainly motivated by the forgery attack against Biscuit (Section 5.3). In particular, the number of extra equations k is a parameter allowing us to optimize Biscuit signature size as it is directly related to the soundness of the MPC protocol for PowAff2 (Section 3.2). It would be possible to derive secure parameters for regular instances PowAff2, but this leads to large signature sizes. Typically, $m - n = 3$ the parameters selected in Section 5.4).

Theorem 3 is proving the result in a restricted setting ($m \le n$ and large fields). As discussed in Remark 2, proving Assumption 1 for any $m > n$ is challenging as it is equivalent to solving Fröberg conjecture. However, we emphasize that a similar assumption was already used in [1] in the complexity analysis of algebraic attacks against the Learning With Errors (LWE) problem [63]. Indeed, given $(\boldsymbol{A} = \{a_{i,j}\}, \boldsymbol{c} = \boldsymbol{s}\boldsymbol{A} + \boldsymbol{e}) \in \mathbb{F}_q^{n \times m} \times \mathbb{F}_q^m$ where $\boldsymbol{s} \in \mathbb{F}_q^n$ is a secret and $\boldsymbol{e} \in \mathbb{F}_q^m$ is an error vector, (search) LWE asks to recover the secret $\boldsymbol{s}$. In [1, 2], the authors proved that LWE reduces to solve the following algebraic system:

$$f_1(x_1, \ldots, x_n) = P(c_1 - \sum_{k=1}^{n} a_{k,1} x_k) = 0, \ldots, f_m(x_1, \ldots, x_n) = P(c_1 - \sum_{k=1}^{n} a_{k,m} x_k) = 0, \quad (6)$$

where P depends on the error distribution. In particular, $P(X) = X(X-1) \in \mathbb{F}_q[X]$ for binary errors and [1] introduced the assumption that a system such as (6) behaves such as a semi-regular sequence. Our assumption is essentially similar and, as discussed in [1] is supported by the fact that the power of an affine of generic forms was already considered in the literature as a candidate for proving Fröberg conjecture and a candidate for a semi-regular sequence [61].

A consequence of Assumption 1 is that solving `PowAff`(2) is not easier than a random system of algebraic equations for generic algorithms. We can then rely on `MQEstimator` (Section 5.1.2) to derive parameters that guarantee the security of Biscuit against the most efficient generic algorithms solving $\texttt{MQ}_q$.

This assumption does not exclude the possibility to have potentially new faster algorithms that are taking into account the structure of the equations. In the next part, we describe a new dedicated algorithm for solving `PowAff`(2).

5.2.2 A specialized enumeration attack

We describe an enumeration attack specialized for systems as in Assumption 1. The main complexity result is as follows:

Theorem 5. *There exists an algorithm that solves* `PowAff`(2) *with parameters* (n, m, q) *whose (heuristic) complexity is*

$$\mathcal{O}(q^{\frac{3(n+1)}{4}}).$$

Proof. We want to solve the quadratic system

$$f_1(x_1, \ldots, x_n) = t_1, \ldots, f_m(x_1, \ldots, x_n) = t_m,$$

where $f_1, \ldots, f_m \in \mathbb{F}_q[x_1, \ldots, x_n]$ and $\boldsymbol{t} = (t_1, \ldots, t_m)$ are defined as in Assumption 1. From now on, we assume $\boldsymbol{t}$ has $k < n$ non-zero coordinates. Without loss of generality, we assume that $t_1, \ldots, t_k$ are non-zero.

1. For each $i \in [m]$, compute the homogenized polynomial

$$f_i^{(h)}(x_1, \ldots, x_n, h) = A_{i,0}^{(h)}(x_1, \ldots, x_n) + \prod_{j=1}^{2} A_{i,j}^{(h)}(x_1, \ldots, x_n) \in \mathbb{F}_q[x_1, \ldots, x_n, h].$$

2. For each $i \in [m]$, define
   (a) $\mathbf{O}_i = \{\boldsymbol{x} \in \mathbb{F}_q^n \mid A_{i,0}^{(h)}(\boldsymbol{x}) = A_{i,1}^{(h)}(\boldsymbol{x}) = 0\}$.
   (b) $\mathbf{K}_i = \mathsf{RightKernel}(\mathbf{A}_i)$ and $\mathbf{A}_i \in \mathbb{F}_q^{n \times n}$ the matrix of rank 2 representing the polar form of $f_i^{(h)}$.
   (c) $\mathbf{W}_i = \mathbf{O}_i \cap \mathbf{K}_i$.
3. $\mathbf{W} \leftarrow \mathbf{W}_1 \cap \cdots \cap \mathbf{W}_k$.
4. $\boldsymbol{f}_k \leftarrow (f_1^{(h)}, \ldots, f_k^{(h)})$ and $\boldsymbol{z}_k \leftarrow (z_1, \ldots, z_k)$, and set $\mathbf{W}^\top$ to be the complement of $\mathbf{W}$ in $\mathbb{F}_q^{n+1}$.

5. $\mathcal{P} \leftarrow \{\boldsymbol{x} \in \mathbf{W}^\top : \boldsymbol{f}_k(\boldsymbol{x}) = \boldsymbol{z}_k\}$.

6. Exhaustively search $\boldsymbol{x} \in \mathcal{P}$ and $\boldsymbol{y} \in \mathbf{W}$ such that

$$f_i^{(h)}(\boldsymbol{x}+\boldsymbol{y}) = z_i,\ \forall i = k+1,\dots,n, \tag{7}$$

and return $\boldsymbol{x}+\boldsymbol{y}$. Return $\perp$ if not vector $\boldsymbol{x}+\boldsymbol{y}$ fulfilling Equation (7) is found.

Now we analyze the complexity of the enumeration attack described above. To this end, we upper-bound the number of times a vector $\boldsymbol{x} \in \mathbb{F}_q^n$ is evaluated either in $\boldsymbol{f}_k$ or in $(f_{k+1}^{(h)},\dots,f_m^{(h)})$. Notice all vector evaluations come from the steps 5 and 6. In step 5, we evaluate a total $|\mathbf{W}^\top|$ vectors in $\boldsymbol{f}_k$, while in step 6, we evaluate a total of $|\mathcal{P}| \cdot |\mathbf{W}|$ in $(f_{k+1}^{(h)},\dots,f_n^{(h)})$. Therefore, we obtain an expected complexity $C(k)$ given by $C(k) := |\mathbf{W}| + |\mathcal{P}| \cdot |\mathbf{W}^\top|$.

From the description of the attack, we know that, with high probability, $\dim(\mathbf{W}_i) = n+1-3$ for each $i \in [k]$. Hence, we expect that $\dim(\mathbf{W}) = n+1-3k$ with high probability [2]. So, $|\mathbf{W}^\top|$ is expected to be q^{3k}, and the one of $|\mathbf{W}|$ is q^{n+1-3k}. Hence the complexity of the aforementioned attack is given by $C(k) = q^{3k} + |\mathcal{P}| \cdot q^{n+1-3k}$.

Now, we focus on determining the expected value of $|\mathcal{P}|$. Since every polynomial in $f_i \in \boldsymbol{f}$ has a randomly chosen linear polynomial summand. We expect $f_i(\boldsymbol{x})$ to be random for a random $\boldsymbol{x} \in \mathbb{F}_q^n$. Hence the expected number of elements in the set $\mathcal{P}$ is $q^{3k}/q^k = q^{2k}$.

Consequently, the expected number of vector evaluations by the algorithm described above is given by $C(k) = q^{3k} + q^{2k} \cdot q^{n+1-3k}$, which is minimized when $k = (n+1)/4$, and yield a minimum complexity $q^{\frac{3(n+1)}{4}}$.

The aforementioned algorithm successfully finds a solution to the input system whenever $\mathbf{W}^\top \cap \mathbf{W} = \emptyset$. We tested experimentally that this is indeed the case with high probability.

The correctness of the algorithm follows from two facts. First, for any $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{F}_q^{n+1}$ it holds

$$\boldsymbol{x}^\top \mathbf{A}_i \boldsymbol{y} = f_i^{(h)}(\boldsymbol{x}+\boldsymbol{y}) - f_i^{(h)}(\boldsymbol{x}) - f_i^{(h)}(\boldsymbol{y}).$$

Second, with high probability $\mathbf{W}_i \subseteq \mathbf{O}_i$, where $\mathbf{O}_i := \{\boldsymbol{x} \in \mathbb{F}_q^{n+1} \mid \ell_{i_0}^{(h)}(\boldsymbol{x}) = 0\}$. In this case, for any $\boldsymbol{y} \in \mathbf{W}_i$, it holds $f_i^{(h)}(\boldsymbol{x}+\boldsymbol{y}) = f_i^{(h)}(\boldsymbol{x})$. □

5.3 Forgery Attack

Let $\big(\boldsymbol{f} = (f_1,\dots,f_m), \boldsymbol{t} = \boldsymbol{f}(\boldsymbol{s})\big) \in \mathbb{F}_q[x_1,\dots,x_n]^m \times \mathbb{F}_q^m$ be a Biscuit public-key. Here we analyze the situation in which an attacker finds a vector $\boldsymbol{s}' \in \mathbb{F}_q^n$ that vanishes a subset of size $m-u$ of the system $\boldsymbol{t} = \boldsymbol{f}(\boldsymbol{x})$. Without loss of generality, we assume $\boldsymbol{s}'$ vanishes the first $m-u$ polynomials and not for the rest. That is, $f_k(\boldsymbol{s}') = t_k$, for $k \in [m-u]$, and $f_k(\boldsymbol{s}') \neq t_k$ for $k = m-u+1,\dots,m$.

By Proposition 1, a set of N parties that follows the `MPC` protocol in Figure 1 on inputs $[\![\boldsymbol{s}']\!]$ and $(\boldsymbol{f}, \boldsymbol{t})$ will output **accept** with probability $p_1 = 1/q^u$. In the context of `MPCitH`, the value p_1 is referred in the literature as the false positive rate of the `MPC` protocol.

[2] We verified this fact experimentally.

Thanks to the Kales-Zaverucha [52] forgery attack on 5-pass Fiat-Shamir, it is known that MPCitH-based signature scheme that consist of τ repetitions of a MPC protocol with false positive rate p_1 can be forged by computing of average

$$\mathtt{KZ}_\tau(p_1, p_2) = \min_{\{\tau_1,\tau_2 \mid \tau_1+\tau_2=\tau\}} \left\{ \frac{1}{\sum_{i=\tau_1}^{\tau} \binom{\tau}{i} p_1^i (1-p_1)^{\tau-i}} + \frac{1}{p_2^{\tau_2}} \right\},$$

calls to some hash functions, where p_2 is the probability of guessing some of the views of parties that remain unopened, e.g., $p_2 = 1/N$ for Biscuit.

Let $\mathtt{C}_u(q, n, m)$ denote the complexity of finding a preimage to a chosen subset S of the system $\boldsymbol{t} = \boldsymbol{f}(\boldsymbol{x})$ of size $m - u$ and $\boldsymbol{s}'$ a solution than vanishes the equations of S. Then, $\boldsymbol{s}'$ might, by chance, also be a solution of any of the equations in S^c, i.e., the equation is not in S. If there remain $k \in [u]$ incorrect values among u, then an attacker can try to mount an attack with complexity $\mathtt{KZ}_\tau(q^{-k}, N^{-1})$.

Let $(\boldsymbol{f}, \boldsymbol{t})$ a Biscuit public-key selected uniformly at random, and let S be a subset of the equations $\boldsymbol{t} = \boldsymbol{f}(\boldsymbol{x})$ of size $m - u$ selected uniformly at random. Then, a random solution $\boldsymbol{s}' \in \mathbb{F}_q^n$ of the equations in S follows a uniform distribution. Hence, $f_\ell(\boldsymbol{s}')$ is a uniform element in $\mathbb{F}_q$. Therefore, the probability that $\boldsymbol{s}'$ is a solution of exactly j equations in S^c is $\binom{u}{j} \cdot (q-1)^{u-j}/q^u$. Consequently, if p_k denotes the probability that $\boldsymbol{s}'$ is not the solution of at most k equations in S^c, then,

$$p_k = \frac{\sum_{j=u-k+1}^{u} \binom{u}{j} \cdot (q-1)^{u-j}}{q^u}.$$

In order to find proper parameters, we have to find a value (k, u) such that

1. $\mathtt{KZ}_\tau(q^{-k}, N^{-1}) > 2^\lambda$, and
2. $\frac{1}{p_k} \cdot \mathtt{C}_u(q, n, m) > 2^{\lambda + C_\lambda}$,

where C_λ is defined in the NIST call for additional signatures[3] : $C_\lambda = 15$ for categories I and III, $C_\lambda = 16$ for Category V.

5.4 Bit Security of Biscuit

Table 11 shows the parameters selected for Biscuit and the corresponding bit security against key-recovery and forgery attacks.

[3]https://csrc.nist.gov/csrc/media/Projects/pqc-dig-sig/documents/call-for-proposals-dig-sig-sept-2022.pdf

Name	Parameters						Security (classical)		Security (quantum)	
	λ	q	n	m	τ	N	recovery	forgery (u)	recovery	forgery (u)
`biscuit128f`	128	16	64	67	34	16	160	143 (17)	136	136 (0)
`biscuit128s`	128	16	64	67	18	256	160	143 (17)	136	136 (0)
`biscuit192f`	192	16	87	90	55	16	210	208 (8)	182	182 (0)
`biscuit192s`	192	16	87	90	31	256	210	208 (8)	182	182 (0)
`biscuit256f`	256	16	118	121	74	16	276	274 (8)	244	244 (0)
`biscuit256s`	256	16	118	121	42	256	276	274 (8)	244	244 (0)

Table 11: Bit security of Biscuit against key-recovery and forgery attacks.

The complexity of a key-recovery attack and the complexity $\mathtt{C}_u$ involved in the forgery attack of Section 5.3 are obtained from the cost of solving a random instance of the `PowAff2` problem with parameters (n, m, q) and $(n, m - u, q)$ respectively. As discussed before, we then use `MQEstimator` to compute the bit security in the classical setting (Section 5.1.2). The quantum bit security has been derived from quantum hybrid approaches (Section 5.1.3) and adding the complexity of these algorithms into the `MQEstimator`. Finally, for the parameter u involved in the cost of the forgery attack from Section 5.3, in the classical setting, we set $k = u - 1$, $u = 17$ for the `biscuit128` and $u = 8$ for `biscuit192`/`biscuit256`. For the quantum case, we found $u = 0$ is optimal for all the parameter sets.

5.5 `EUF-CMA` Security

Existential unforgeability under adaptive chosen-message attacks (`EUF-CMA`*)* is the classical security notion used for `DSS`. Since Biscuit is constructed using the `MPCitH` paradigm and Fiat-Shamir transform, we get:

Theorem 6. *Assuming the hardness of* `PowAff2`*,* Biscuit *is* `EUF-CMA`*-secure.*

The proof follows from [37, Theorem 5] (which in turn is highly inspired by that of [20, Theorem 6.2]).

5.6 Security Against Side-Channel Attacks

In this section, we briefly discuss the possible side-channel vulnerabilities of our proposition, and how to address them.

Timing Attacks. First of all, it is worth to notice that the signature procedure of Biscuit is independent on the secret value as long as the field arithmetic and the hash functions do not leak information on the manipulated data. Indeed, there are no branching that depend on the value of any secret in the algorithm. This allows to make an isochronous, and even constant-time implementation by focusing on the field arithmetic.

Side-Channel Attacks. The most popular technique to prevent these attacks is to use *masking*: we compute a sharing of the secret using fresh random value at each execution.

In our proposition, it happens that most of the time, the secret value $\boldsymbol{s}$ is already shared into N shares due to the MPC protocol. Nevertheless, this does not guarantee security at order $N-1$ because all the shares (except one) will finally become public during the verification process. This sharing is not secure from a side-channel attacker point of view.

However, as our construction uses fields of characteristic, classical boolean masking techniques can be applied throughout the scheme.

6 Advantages and Limitations

Since MQDSS and Picnic, the use of MPC and ZK techniques for designing DSS flourished in the literature, e.g [54, 9, 52, 53, 37, 20]. This leads to improved techniques and more and more efficient DSS. Typically, the performances of Picnic significantly improved between the first and third round of the previous NIST post-quantum standardization process.

Biscuit takes advantage of this abundant literature and selected a problem that seems rather optimal for current MPCitH-based proof systems. We believe that Biscuit has a rather interesting performance profile compared to current post-quantum standards and is based on different hardness assumption (multivariate-based). It can be noted that the signature+public-key size is comparable to Dilithium and smaller than SPHINCS+.

Biscuit achieves EUF-CMA-security which is not really common in multivariate-based cryptography. On the other hand, this security proof relies on the hardness of a non-standard problem : PowAff2. As pointed out in the document, we assume that this problem is as hard as solving random equations (and also prove the statement in some restricted cases). The use of such a problem was also motivated by the fact that faster algorithms against PowAff2 would lead to faster algebraic attacks against LWE.

References

[1] Martin R. Albrecht, Carlos Cid, Jean-Charles Faugère, and Ludovic Perret. Algebraic algorithms for lwe. Cryptology ePrint Archive, Paper 2014/1018, 2014. https://eprint.iacr.org/2014/1018.

[2] Sanjeev Arora and Rong Ge. New algorithms for learning in presence of errors. In Luca Aceto, Monika Henzinger, and Jirí Sgall, editors, *Automata, Languages and Programming - 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4-8, 2011, Proceedings, Part I*, volume 6755 of *Lecture Notes in Computer Science*, pages 403–415. Springer, 2011.

[3] Gwénolé Ars, Jean-Charles Faugère, Hideki Imai, Mitsuru Kawazoe, and Makoto Sugita. Comparison between XL and gröbner basis algorithms. In Pil Joong Lee, editor, *Advances in Cryptology - ASIACRYPT 2004, 10th International Conference on the Theory and Application of Cryptology and Information Security, Jeju Island, Korea, December 5-9, 2004, Proceedings*, volume 3329 of *Lecture Notes in Computer Science*, pages 338–353. Springer, 2004.

[4] Magali Bardet. *Étude des systèmes algébriques surdéterminés. Applications aux codes correcteurs et à la cryptographie.* PhD thesis, Université de Paris VI, 2004.

[5] Magali Bardet, Jean-Charles Faugère, and Bruno Salvy. On the complexity of Gröbner basis computation of semi-regular overdetermined algebraic equations. In *International Conference on Polynomial System Solving – ICPSS*, pages 71–75, 2004.

[6] Magali Bardet, Jean-Charles Faugère, and Bruno Salvy. On the Complexity of the F5 Gröbner basis Algorithm. *Journal of Symbolic Computation*, pages 1–24, September 2014. 24 pages.

[7] Magali Bardet, Jean-Charles Faugère, Bruno Salvy, and Pierre-Jean Spaenlehauer. On the Complexity of Solving Quadratic Boolean Systems. *J. Complex.*, 29(1):53–75, 2013.

[8] Magali Bardet, Jean-Charles Faugère, Bruno Salvy, and Bo-Yin Yang. Asymptotic behaviour of the degree of regularity of semi-regular polynomial systems. In *The Effective Methods in Algebraic Geometry Conference – MEGA 2005*, pages 1–14, 2005.

[9] Carsten Baum and Ariel Nof. Concretely-efficient zero-knowledge arguments for arithmetic circuits and their application to lattice-based cryptography. In Aggelos Kiayias, Markulf Kohlweiss, Petros Wallden, and Vassilis Zikas, editors, *Public-Key Cryptography – PKC 2020*, pages 495–526, Cham, 2020. Springer International Publishing.

[10] Emanuele Bellini, Rusydi H. Makarim, Carlo Sanna, and Javier A. Verbel. An estimator for the hardness of the MQ problem. In Lejla Batina and Joan Daemen, editors, *Progress in Cryptology - AFRICACRYPT 2022: 13th International Conference on Cryptology in Africa, AFRICACRYPT 2022, Fes, Morocco, July 18-20, 2022, Proceedings*, Lecture Notes in Computer Science, pages 323–347. Springer Nature Switzerland, 2022.

[11] Daniel J. Bernstein and Bo-Yin Yang. Asymptotically Faster Quantum Algorithms to Solve Multivariate Quadratic Equations. In Tanja Lange and Rainer Steinwandt, editors, *Post-Quantum Cryptography - 9th International Conference, PQCrypto 2018, Fort Lauderdale, FL, USA, April 9-11, 2018, Proceedings*, volume 10786 of *Lecture Notes in Computer Science*, pages 487–506. Springer, 2018.

[12] Luk Bettale, Jean-Charles Faugère, and Ludovic Perret. Hybrid approach for solving multivariate systems over finite fields. *J. Math. Cryptol.*, 3(3):177–197, 2009.

[13] Luk Bettale, Jean-Charles Faugère, and Ludovic Perret. Solving polynomial systems over finite fields: improved analysis of the hybrid approach. In Joris van der Hoeven and Mark van Hoeij, editors, *International Symposium on Symbolic and Algebraic Computation, ISSAC'12, Grenoble, France - July 22 - 25, 2012*, pages 67–74. ACM, 2012.

[14] Andreas Björklund, Petteri Kaski, and Ryan Williams. Solving Systems of Polynomial Equations over GF(2) by a Parity-Counting Self-Reduction. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece*, volume 132 of *LIPIcs*, pages 26:1–26:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[15] Wieb Bosma, John J. Cannon, and Catherine Playoust. The Magma algebra system I: The user language. *Journal of Symbolic Computation*, 24(3-4):235–265, 1997.

[16] Charles Bouillaguet. Boolean polynomial evaluation for the masses. Cryptology ePrint Archive, Paper 2022/1412, 2022. https://eprint.iacr.org/2022/1412.

[17] Charles Bouillaguet, Hsieh-Chung Chen, Chen-Mou Cheng, Tung Chou, Ruben Niederhagen, Adi Shamir, and Bo-Yin Yang. Fast exhaustive search for polynomial systems in F_2. In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*, volume 6225 of *Lecture Notes in Computer Science*, pages 203–218. Springer, 2010.

[18] Bruno Buchberger. Bruno Buchberger's PhD thesis 1965: An algorithm for finding the basis elements of the residue class ring of a zero dimensional polynomial ideal. *Journal of Symbolic Computation*, 41(3-4):475–511, 2006.

[19] Bruno Buchberger, Georges E. Collins, Rudiger G. K. Loos, and Rudolph Albrecht. Computer algebra symbolic and algebraic computation. *SIGSAM Bull.*, 16(4):5–5, 1982.

[20] Melissa Chase, David Derler, Steven Goldfeder, Jonathan Katz, Vladimir Kolesnikov, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, Xiao Wang, and Greg Zaverucha. The picnic signature scheme. Design Document. Version 3.0, 2020. https://github.com/microsoft/Picnic/blob/master/spec/spec-v3.0.pdf.

[21] Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. Post-quantum zero-knowledge and signatures from symmetric-key primitives. In Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1825–1842. ACM, 2017.

[22] Lily Chen, Stephen Jordan, Yi-Kai Liu, Dustin Moody, Rene Peralta, Ray Perlner, and Daniel Smith-Tone. Report on Post-Quantum Cryptography. Research report NISTIR 8105, NIST, 2016.

[23] Ming-Shing Chen, Andreas Hülsing, Joost Rijneveld, Simona Samardjiska, and Peter Schwabe. `MQDSS` specifications.

[24] Ming-Shing Chen, Andreas Hülsing, Joost Rijneveld, Simona Samardjiska, and Peter Schwabe. From 5-pass *MQ* -based identification to *MQ* -based signatures. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part II*, volume 10032 of *Lecture Notes in Computer Science*, pages 135–165, 2016.

[25] Yu-Ao Chen and Xiao-Shan Gao. Quantum algorithm for boolean equation solving and quantum algebraic attack on cryptosystems. *J. Syst. Sci. Complex.*, 35(1):373–412, 2022.

[26] Yu-Ao Chen, Xiao-Shan Gao, and Chun-Ming Yuan. Quantum algorithms for optimization and polynomial systems solving over finite fields. *CoRR*, abs/1802.03856, 2018.

[27] Nicolas T. Courtois, Alexander Klimov, Jacques Patarin, and Adi Shamir. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In Bart Preneel, editor, *Advances in Cryptology - EUROCRYPT 2000, International Conference on the Theory and Application of Cryptographic Techniques, Bruges, Belgium, May 14-18, 2000, Proceeding*, volume 1807 of *Lecture Notes in Computer Science*, pages 392–407. Springer, 2000.

[28] R. DeMillo and R. Lipton. A probabilistic remark on algebraic program testing. *Information Processing Letters*, 7(4):192–194, 1978.

[29] Jintai Ding, Vlad Gheorghiu, András Gilyén, Sean Hallgren, and Jianqiang Li. Limitations of the Macaulay Matrix Approach for using the `HLL` Algorithm to Solve Multivariate Polynomial Systems, 2021.

[30] Itai Dinur. Improved Algorithms for Solving Polynomial Systems over GF(2) by Multiple Parity-Counting. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 2550–2564. SIAM, 2021.

[31] NIST Computer Security Division. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. FIPS Publication 202, National Institute of Standards and Technology, U.S. Department of Commerce, May 2014.

[32] Christian Eder and Jean-Charles Faugère. A survey on signature-based algorithms for computing gröbner bases. *J. Symb. Comput.*, 80:719–784, 2017.

[33] Andre Esser, Javier Verbel, Floyd Zweydinger, and Emanuele Bellini. `CryptographicEstimators`: a software library for cryptographic hardness estimation. Cryptology ePrint Archive, Paper 2023/589, 2023. https://eprint.iacr.org/2023/589.

[34] J.-C. Faugère. A new efficient algorithm for computing gröbner bases (F4). *Journal of Pure and Applied Algebra*, 139(1-3):61–88, 1999.

[35] J.-C. Faugère. A new efficient algorithm for computing gröbner bases without reduction to zero : F5. In *ISSAC'02*, pages 75–83. ACM press, 2002.

[36] Jean-Charles Faugère, Kelsey Horan, Delaram Kahrobaei, Marc Kaplan, Elham Kashefi, and Ludovic Perret. Fast quantum algorithm for solving multivariate quadratic equations. *IACR Cryptol. ePrint Arch.*, page 1236, 2017.

[37] Thibauld Feneuil, Antoine Joux, and Matthieu Rivain. Syndrome decoding in the head: Shorter signatures from zero-knowledge proofs. Cryptology ePrint Archive, Paper 2022/188, 2022.

[38] Amos Fiat and Adi Shamir. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194. Springer, 1986.

[39] Ralf Fröberg. An inequality for Hilbert series of graded algebras. *Mathematica Scandinavica*, 56:117–144, 1985.

[40] H. Furue, S. Nakamura, and T. Takagi. Improving Thomae-Wolf algorithm for solving underdetermined multivariate quadratic polynomial problem. In J. H. Cheon and J.-P. Tillich, editors, *PQcrypto 2021. LNCS*, pages 65–78, Cham, 2021. Springer International Publishing.

[41] Hiroki Furue and Tsuyoshi Takagi. Fast enumeration algorithm for multivariate polynomials over general finite fields. Cryptology ePrint Archive, Paper 2023/619, 2023. https://eprint.iacr.org/2023/619.

[42] Shuhong Gao, Yinhua Guan, and Frank Volny, IV. A new incremental algorithm for computing groebner bases. In *Proceedings of the 2010 International Symposium on Symbolic and Algebraic Computation*, ISSAC '10, pages 13–19, New York, NY, USA, 2010. ACM.

[43] Shuhong Gao, Frank Volny IV, and Mingsheng Wang. A new framework for computing gröbner bases. *Math. Comput.*, 85(297), 2016.

[44] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[45] S. B. Gashkov and I. S. Sergeev. Complexity of computations in finite fields. *Fundam. Prikl. Mat.*, 17(4):95–131, 2011/12.

[46] Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. ZKBoo: Faster Zero-Knowledge for Boolean Circuits. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 1069–1083, 2016.

[47] Lov K. Grover. A fast quantum mechanical algorithm for database search. In Gary L. Miller, editor, *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 212–219. ACM, 1996.

[48] Aram W. Harrow, Avinatan Hassidim, and Seth Lloyd. Quantum algorithm for linear systems of equations. *Phys. Rev. Lett.*, 103:150502, Oct 2009.

[49] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 21–30, 2007.

[50] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge proofs from secure multiparty computation. *SIAM Journal on Computing*, 39(3):1121–1152, 2009.

[51] Antoine Joux and Vanessa Vitse. A crossbred algorithm for solving boolean polynomial systems. In Jerzy Kaczorowski, Josef Pieprzyk, and Jacek Pomykała, editors, *Number-Theoretic Methods in Cryptology*, pages 3–21, Cham, 2018. Springer International Publishing.

[52] Daniel Kales and Greg Zaverucha. An attack on some signature schemes constructed from five-pass identification schemes. In Stephan Krenn, Haya Shulman, and Serge Vaudenay, editors, *Cryptology and Network Security*, pages 3–22, Cham, 2020. Springer International Publishing.

[53] Daniel Kales and Greg Zaverucha. Efficient lifting for shorter zero-knowledge proofs and post-quantum signatures. Cryptology ePrint Archive, Paper 2022/588, 2022. https://eprint.iacr.org/2022/588.

[54] Jonathan Katz, Vladimir Kolesnikov, and Xiao Wang. Improved non-interactive zero-knowledge with applications to post-quantum signatures. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 525–537, 2018.

[55] Aviad Kipnis, Jacques Patarin, and Louis Goubin. Unbalanced oil and vinegar signature schemes. In Jacques Stern, editor, *Advances in Cryptology — EUROCRYPT '99*, pages 206–222, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[56] Daniel Lazard. Gröbner-bases, Gaussian elimination and resolution of systems of algebraic equations. In *Proceedings of the European Computer Algebra Conference on Computer Algebra*, volume 162 of *Lecture Notes in Computer Science*, Berlin, Heidelberg, New York, 1983. Springer Verlag.

[57] Daniel Lokshtanov, Ramamohan Paturi, Suguru Tamaki, R. Ryan Williams, and Huacheng Yu. Beating Brute Force for Systems of Polynomial Equations over Finite Fields. In Philip N. Klein, editor, *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 2190–2202. SIAM, 2017.

[58] Francis S. Macaulay. On some formula in elimination. *London Mathematical Society*, 1(33):3–27, 1902.

[59] F.S. Macaulay. *The Algebraic Theory of Modular Systems*. Cambridge University Press, 1916.

[60] Daniele Micciancio and Chris Peikert. Hardness of SIS and LWE with small parameters. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 21–39. Springer, 2013.

[61] Lisa Nicklasson. On the hilbert series of ideals generated by generic forms. *Communications in Algebra*, 45(8):3390–3395, oct 2016.

[62] NIST. Proposed Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process. http://csrc.nist.gov/groups/ST/post-quantum-crypto/documents/call-for-proposals-draft-aug-2016.pdf.

[63] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *J. ACM*, 56(6):34:1–34:40, 2009.

[64] Peter Schwabe and Bas Westerbaan. Solving binary MQ with grover's algorithm. In Claude Carlet, M. Anwar Hasan, and Vishal Saraswat, editors, *Security, Privacy, and Applied Cryptography Engineering - 6th International Conference, SPACE 2016, Hyderabad, India, December 14-18, 2016, Proceedings*, volume 10076 of *Lecture Notes in Computer Science*, pages 303–322. Springer, 2016.

[65] J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM*, 27(4):701–717, 1980.

[66] Enrico Thomae and Christopher Wolf. Solving underdetermined systems of multivariate quadratic equations revisited. In Marc Fischlin, Johannes Buchmann, and Mark Manulis, editors, *Public Key Cryptography – PKC 2012*, pages 156–171. Springer Berlin Heidelberg, 2012.

[67] Bo-Yin Yang, Jiun-Ming Chen, and Nicolas T. Courtois. On asymptotic security estimates in XL and gröbner bases-related algebraic cryptanalysis. In Javier López, Sihan Qing, and Eiji Okamoto, editors, *Information and Communications Security, 6th International Conference, ICICS 2004, Malaga, Spain, October 27-29, 2004, Proceedings*, volume 3269 of *Lecture Notes in Computer Science*, pages 401–413. Springer, 2004.

[68] Greg Zaverucha, Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, Jonathan Katz, Xiao Wang, Vladmir Kolesnikov, and Daniel Kales. `Picnic` : Algorithm specification and design document.

[69] R. Zippel. Probabilistic algorithms for sparse polynomials. In *Symbolic and algebraic computation (EUROSAM'79), Internat. Sympos.*, volume 72 of *Lecture Notes in Computer Science*, pages 216–226. Springer Verlag, 1979.