Biscuit: New MPCitH Signature Scheme from Structured Multivariate Polynomials

Luk Bettale[1], Delaram Kahrobaei[2,3,4,5], Ludovic Perret[6], and Javier Verbel[7]

[1] IDEMIA, France
[2] Departments of Computer Science and Mathematics, Queens College, City University of New York, USA
[3] Initiative for the Theoretical Sciences, Graduate Center, City University of New York, USA
[4] Department of Computer Science, University of York, UK
[5] Department of Computer Science and Engineering, Tandon School of Engineering, New York University, USA
[6] Sorbonne University, CNRS, LIP6, PolSys, Paris, France
[7] Technology Innovation Institute, UAE

Abstract. This paper describes Biscuit, a new multivariate-based signature scheme derived using the MPC-in-the-Head (MPCitH) approach. The security of Biscuit is related to the problem of solving a set of structured quadratic algebraic equations. These equations are highly compact and can be evaluated using very few multiplications (one multiplication per equation). The core of Biscuit is a rather simple MPC protocol for secure multiplications using standard optimized multiplicative triples. This paper also includes several improvements toward the initial version of Biscuit submitted to the NIST PQC standardization process for additional signature schemes. Notably, we introduce a new hypercube variant of Biscuit, refine the security analysis with recent third-party attacks, and present a new AVX2 implementation of Biscuit.

Keywords: Post-Quantum · Digital Signature · MPC-in-the-Head · Multivariate Polynomials

1 Introduction

Biscuit is a new multivariate-based digital signature scheme submitted to the recent NIST standardization process for additional post-quantum signature schemes [1]. The security of Biscuit is proven assuming the hardness of the so-called PowAff2 problem (Definition 1), which is a structured version of the well-known Multivariate Quadratic (MQ) problem [16].
Biscuit is in the lineage of the Picnic signature scheme [21,36], which was selected as an alternate candidate in the first NIST post-quantum cryptography standardization process [6]. The security of Picnic relies on the hardness of a key-recovery attack for a lightweight block cipher. The design of Picnic builds over a Multi-Party Computation (MPC) protocol for multiplicative triples and

follows the MPC-in-the-Head (MPCitH) paradigm [28] to obtain a Zero-Knowledge Proof-of-Knowledge (ZKPoK) for the key-recovery problem. Finally, the signature scheme is obtained by applying the Fiat-Shamir transformation [26] to the ZKPoK protocol.

As in Picnic, the design of Biscuit follows the MPCitH paradigm and relies essentially on the same MPC protocol to check multiplicative triples. Biscuit is build on top of a ZKPoK for the problem of finding a pre-image $\boldsymbol{s} \in \mathbb{F}_q^n$ of a system of structured quadratic multivariate polynomial equations $\boldsymbol{f} \in \mathbb{F}_q[x_1, \ldots, x_n]^m$ over a finite field. The private and public keys in Biscuit are respectively $\boldsymbol{s} \in \mathbb{F}_q^n$ and $(\boldsymbol{f}, \boldsymbol{t}) \in \mathbb{F}_q[x_1, \ldots, x_n]^m \times \mathbb{F}_q^m$, where $\boldsymbol{t} = \boldsymbol{f}(\boldsymbol{s})$.

The performance of Picnic is proportional to the number of multiplications required to evaluate the circuit defining the underlying block-cipher with the secret-key. This fact motivates the use of a set $\boldsymbol{f} = (f_1 \ldots, f_m) \in \mathbb{F}_q[x_1, \ldots, x_n]^m$ of polynomial equations that require a small number of multiplications to be evaluated. Biscuit considers polynomials of the form $f_i = A_0 + A_1 \cdot A_2$, where each $A_i \in \mathbb{F}_q[x_1, \ldots, x_n]$ is an affine polynomial. These polynomials can be evaluated using only one multiplication, while a random quadratic polynomial would require $O(n^2)$ multiplications.

1.1 Overview of MPCitH-Based Signature Schemes

Since Picnic, the use of MPCitH for designing post-quantum signature schemes has become extremely popular. This is evidenced in the new NIST standardization process for post-quantum signature schemes, where eight[8] among forty of the submitted schemes are using the MPCitH framework. These schemes follow the same design methodology but differ in the hard problems considered.

AIMer is based on the hardness of key-recovery of a MPC-friendly block-cipher [32], MIRA and MiRitH are based on the MinRank problem [9,4], MQOM is based on the problem of solving random quadratic equations [24], PERK is based on the Permuted Kernel Problem [3], RYDE is based on the rank syndrome decoding problem [8], and SDith relies on the syndrome decoding problem [33]. All these schemes proposed several parameter sets to optimize either the signature size (short variant) or the signing and verification times (fast variant). In Table 1, we overview the performances of these NIST candidates with the version of Biscuit described in this paper. The table also includes FAEST [13] whose security is based on AES but uses a new zero-knowledge technique, named VOLE-in-the head, that improves the MPCitH approach.

For each scheme[9], we report on a short variant achieving NIST level-I security (i.e. equivalent to the security of AES128). The key-generation (keygen), signature generation (sign), and verification (verify) times are shown in clock-cycles (cycles). These numbers have been extracted directly from the corresponding submissions and we refer to these documents for details. The purpose of these numbers is to give a rough global

[8] https://csrc.nist.gov/projects/pqc-dig-sig/round-1-additional-signatures

[9] A few days before finalizing this manuscript a new preprint appeared [25] that seems to significantly improve MQOM as well as many MPCitH-based signature schemes (including Biscuit).

Name	Performance (cycles)			Size (bytes)		
	`keygen`	`sign`	`verify`	`sk`	`pk`	σ
`AIMer-L1PARAM4`	54 435	78 022 625	73 813 256	16	32	3 840
`MIRA-128s`	112 000	46 800 000	43 900 000	16	84	5 640
`MiRitH-Ias`	108 903	41 220 707	40 976 634	16	129	5 673
`MQOM-L1-gf31-short`	67 000	44 360 000	41 720 000	78	47	6 352
`PERK-I-short5`	91 000	36 000 000	25 000 000	16	24	6 006
`RYDE128s`	33 100	23 400 000	20 100 000	32	86	5 956
`SDith-L1-hyp`	7 083 000	13 400 000	12 500 000	404	120	8 260
`Biscuit-128s (this work)`	62 484	27 922 077	28 484 726	16	68	5 748
`FAEST-128s`	200 000	25 580 000	25 830 000	32	32	5 006

Table 1: Performance of level-I short variants of `MPCitH`-based candidates submitted to the first round of the new `NIST` call for post-quantum signature schemes.

perspective as the methodology to derive clock-cycles, as well as the level of optimization, could differ between submissions. Table 1 also includes secret-key (sk), public-key (pk) and signature (σ) sizes in bytes.

1.2 Organization of the Paper and Main Results

After this introduction, the paper is organized as follows. Section 2 introduces basic notations, the new hard problem considered in Biscuit (`PowAff2` problem, Section 2.2), as well as the basic cryptography building blocks underlying its design: Multi-Party Computation (`MPC`), `MPC`-in-the-Head approach (`MPCitH`), Zero-Knowledge Proof of Knowledge (`ZKPoK`), proof systems using multiplicative triples and the hypercube technique for `MPCitH`-based signature schemes.
Section 3 describes the core sub-protocols underlying Biscuit. Due to the structure of the algebraic systems considered in Biscuit, the evaluation of a `PowAff2` solution requires only one multiplication per equation. This leads to a rather simple `MPC` protocol (Section 3.1) for `PowAff2` that is based on the parallel execution of secure multiplication using Beaver multiplicative triples [15] with some optimizations from [14,30]. Then, we derive a new `ZKPoK` for `PowAff2` (Section 3.2) using the `MPCitH` approach. Note that the protocol presented here (Figure 3) differs from the one described in the initial Biscuit submission [19]. In particular, we use the hypercube technique [34] and also include a security proof (Theorem 1) of the new `ZKPoK`.
Section 4 presents the Biscuit signature scheme and details the key generation, signature generation (Figure 7) and verification (Figure 8) algorithms. Biscuit is constructed using the traditional Fiat-Shamir transform from the `ZKPoK` described in Figure 3. We conclude this part with Table 2 that summarizes the secret-key, public-key, and signature sizes for the three security levels of `NIST`. In particular, Biscuit achieves a signature of 5.7KB for the first security level. This is comparable to other recent `MPCitH`-based signature schemes (Section 1.1).
Section 5 analyzes the security of the parameters proposed in Table 2. This section revisits the security analysis performed in the initial submission of Biscuit by taking into account a new third-party analysis [20]. In Section 5.1, we first explain the connection between the hardness of `PowAff2` and the difficulty of solving the Learning

With (bounded) Errors (LWE) problem [35]. In Section 5.2, we consider the key-recovery problem where the best attack against is a new dedicated hybrid approach, i.e. that combines exhaustive search and Gröbner bases [18,17,12], for solving PowAff2 equations described in [20]. In Section 5.3, we refine the analysis of Kales and Zaverucha [29] for forgery attacks against 5-pass Fiat-Shamir based signature schemes. This leads us to introduce a variant of the PowAff2 problem where the attacker has to solve a sub-system with fewer equations; leading to the introduction of the $\texttt{PowAff2}_u$ problem (Definition 1).

Finally, Section 6 presents an optimized implementation of Biscuit which outperforms the previous implementation. First, we use a new canonical representation of the PowAff2 equations (Lemma 1), which allows us to simplify their evaluation further. Then, we integrate the hypercube framework for even further improvements.

2 Preliminaries

This section presents preliminary concepts and notations used in this paper.

2.1 Notations

Throughout this paper, we use λ for the security parameter. Also, $[n]$ refers to the set $\{1,\dots,n\}$ for an integer $n \in \mathbb{N}$, $\mathbb{F}_q$ is the finite field of q elements (where q is prime or a prime power), $\mathbb{F}_q^m$ denotes the vector space of dimension m over $\mathbb{F}_q$ and $\mathbb{F}_q[x_1,\dots,x_n]$ is the ring of polynomials in the variables $x_1,\dots,x_n$ over the field $\mathbb{F}_q$.

Bold lower-case letters denote vectors, $\mathbf{x}+\mathbf{y}$ denotes the element-wise addition. We use $a \leftarrow \mathcal{A}(x)$ to indicate that a is the output of an algorithm $\mathcal{A}$ on input x, $a \xleftarrow{\$} \mathcal{S}$ means that a is sampled uniformly at random from a set $\mathcal{S}$.

Let $\mathcal{R}$ be a ring and $a \in \mathcal{R}$. The additive sharing of a, denoted by $[\![a]\!]$, is a tuple $[\![a]\!] := ([\![a]\!]_1,\dots,[\![a]\!]_N) \in \mathcal{R}^N$ such that $a = \sum_{i=1}^N [\![a]\!]_i$. Each component $[\![a]\!]_i$ of $[\![a]\!]$ is called a *share* of a. Throughout this paper, we only consider additive sharing and use the word sharing to refer to additive sharing.

A *Multi-Party Computation* (MPC) protocol is an interactive protocol executed by a set of N parties knowing a public function f. Its goal is to compute the image $z = f(x_1,\dots,x_N)$, where the value x_i is only known by the i-th party. A 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.

2.2 The $\texttt{PowAff2}_u$ Problem

The core problem considered in Biscuit is the one of solving a system of multivariate equations defined as the product of two affine forms. Denoted by $\texttt{PowAff2}_u$, the problem is parameterized by a tuple of positive integers (n,m,u,q), where n is the number of variables, m the number of equations, u is a parameter related to forgery (Section 5.3), and q is the finite field size.

Definition 1 (The $\texttt{PowAff2}_u$ problem).
Let $A_{1,0},A_{1,1},A_{1,2},\dots,A_{m,0},A_{m,1},A_{m,2} \in \mathbb{F}_q[x_1,\dots,x_n]$ be affine forms, i.e.:

$$A_{k,j}(x_1,\dots,x_n) = a_0^{(k,j)} + \sum_{i=1}^{n} a_i^{(k,j)} x_i, \ \text{with } a_0^{(k,j)},\dots,a_n^{(k,j)} \in \mathbb{F}_q. \tag{1}$$

Input. *A vector* $\mathbf{t} = (t_1, \ldots, t_m) \in \mathbb{F}_q^m$ *and multivariate polynomials* $\mathbf{f} = (f_1, \ldots, f_m) \in \mathbb{F}_q[x_1, \ldots, x_n]^m$ *defined as:*

$$f_k(x_1, \ldots, x_n) = A_{k,0}(x_1, \ldots, x_n) + \prod_{j=1}^{2} A_{k,j}(x_1, \ldots, x_n), \forall k \in [m]. \tag{2}$$

Question. *Find – if any – a vector* $(s_1, \ldots, s_n) \in \mathbb{F}_q^n$ *and set* $J \subseteq [m]$ *of size* $m - u$ *such that:*

$$f_j(s_1, \ldots, s_n) = t_j,\ \forall j \in J.$$

Definition 2 (The PowAff2 problem). *We use* PowAff*2 to denote the* PowAff2_0 *problem. We call* PowAff*2 algebraic system the set of non-linear equations* $f_1, \ldots, f_m \in \mathbb{F}_q[x_1, \ldots, x_n]$ *defined as in* (2).

PowAff2 is the problem corresponding to key-recovery whilst PowAff2_u, with $u > 0$, is a relaxation that corresponds to signature forgery whose hardness is detailed in Section 5. The current best attack against Biscuit has been described in [20]. In particular, it was mentioned that the multivariate equations defined as in Definition 1 can be reduced to a simple, but equivalent, structure.

Lemma 1. *Let* $\mathbf{f} = (f_1, \ldots, f_m) \in \mathbb{F}_q[x_1, \ldots, x_n]^m$ *be a* PowAff*2 algebraic system. Then, with high probability, there exists an invertible matrix* $\mathbf{L} \in \mathrm{GL}_n(\mathbb{F}_q)$ *such that :*

$$\begin{aligned}\mathbf{f}\big(\mathbf{x} \cdot \mathbf{L}\big) = \big(&u_1(\boldsymbol{x}) \cdot (x_1 + c_1) + w_1(\mathbf{x}), \ldots, u_n(\mathbf{x}) \cdot (x_n + c_n) + w_n(\boldsymbol{x}),\\ &A'_{n+1,0}(\mathbf{x}) + \prod_{j=1}^{2} A'_{n+1,j}(\mathbf{x}), \ldots, A'_{m,0}(x_1, \ldots, x_n) + \prod_{j=1}^{2} A'_{m,j}(\mathbf{x})\big)\end{aligned}$$

where $\mathbf{x} = (x_1, \ldots, x_n), A_{n+1,0}, A_{n+1,1}, A_{n+1,2}, \ldots, A_{m,0}, A_{m,1}, A_{m,2}, u_1, \ldots, u_n, v_1, \ldots, v_n \in \mathbb{F}_q[x_1, \ldots, x_n]$ *are affine polynomials and* $c_1, \ldots, c_n \in \mathbb{F}_q$.

Proof. By construction, we have :

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

with $A_{1,0}, A_{1,1}, A_{1,2}, \ldots, A_{m,0}, A_{m,1}, A_{m,2} \in \mathbb{F}_q[x_1, \ldots, x_n]$ affine forms as in (1). Thus, we can write $A_{k,2}(x_1, \ldots, x_n) = (x_1, \ldots, x_n) \cdot \mathbf{b}_k + c_k$, where $\mathbf{b}_k = (a_1^{(k,2)}, \ldots, a_n^{(k,2)}) \in \mathbb{F}_q^n$ and $c_k = a_0^{(k,2)} \in \mathbb{F}_q$. Let $\mathbf{C} \in \mathbb{F}_q^{n \times n}$ be the matrix whose rows are $\mathbf{b}_1, \ldots, \mathbf{b}_n$. We want to find a non-singular matrix $\mathbf{L} \in \mathrm{GL}_n(\mathbb{F}_q)$ such that $\mathbf{I}_n = \mathbf{C} \cdot \mathbf{L}$, where $\mathbf{I}_n$ is the identity matrix of size n. This reduces to compute, if any, the inverse of $\mathbf{C}$. ◻

2.3 Digital Signature Scheme

Definition 3. *A Digital Signature Scheme (*DSS*) is a tuple of three probabilistic polynomial-time algorithms (KeyGen, Sign, Verify) verifying:*

1. $(pk, sk) \leftarrow$ KeyGen(1^λ). *The key-generation algorithm KeyGen takes as input a security parameter* 1^λ *and outputs a pair of public/private keys* (pk, sk).
2. $\sigma \leftarrow$ Sign(sk, msg). *The signing algorithm Sign takes a private key sk and a message* $msg \in \{0, 1\}^*$ *and outputs a signature* σ.

3. $b \leftarrow \mathsf{Sign}(\mathsf{pk}, \sigma, \mathsf{msg})$. *The verification algorithm* Verify *is deterministic. It takes as input a message* $\mathsf{msg} \in \{0,1\}^*$, *a signature* σ, *and a public key* pk. *It outputs a bit* $b \in \{0,1\}$, 1 *means that it* ***accepts*** σ *as a valid signature for* msg, *otherwise it* ***rejects*** *returning* 0.

A signature scheme is correct if for every security parameter $\lambda \in \mathbb{N}$, every $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$, and every message $\mathsf{msg} \in \{0,1\}^*$, it holds that

$$1 \leftarrow \mathsf{Verify}\big(\mathsf{pk}, \mathsf{msg}, \mathsf{Sign}(\mathsf{sk}, \mathsf{msg})\big).$$

The standard security notion for a DSS is Existential Unforgeability under Adaptive Chosen-Message Attacks (EU-CMA). We say that a signature scheme is EU-CMA-secure if for all probabilistic polynomial-time adversaries $\mathcal{A}$, the probability

$$\Pr\left[1 \leftarrow \mathsf{Verify}(\mathsf{pk}, \mathsf{msg}^*, \sigma^*) \,\middle|\, \begin{array}{c} (\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\lambda) \\ (\mathsf{msg}^*, \sigma^*) \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{Sign}(\mathsf{sk},\cdot)}}(\mathsf{pk}) \end{array}\right]$$

is a negligible function in λ, where $\mathcal{A}$ is given access to a signing oracle $\mathcal{O}_{\mathsf{Sign}(\mathsf{sk},\cdot)}$, and msg^* has not been queried to $\mathcal{O}_{\mathsf{Sign}(\mathsf{sk},\cdot)}$.

Auxiliary Functions. Biscuit also relies on further basic cryptographic building blocks that we do not explicitly introduce such as commitments, collision-resistant hash functions, key-derivation functions, and pseudo-random number generators. As explained in [19], we can use the SHAKE256 [22] extendable-output function (XOF) to instantiate these functions.

During signature, the signer must generate a set of N seeds and reveal $N-1$ of them to the verifier for each iteration (TreePRG). 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 refer to [19] for the description of TreePRG.

2.4 5-Pass Identification Schemes

An Identification Scheme (IDS) is an interactive protocol between a *prover* P and a *verifier* V, where P wants to prove its knowledge of a secret value sk to V using a public value pk.

Definition 4 (5-pass identification scheme). *A* 5*-pass* IDS *is a tuple of three probabilistic polynomial-time algorithms* (KeyGen, P, V) *such that*

1. $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$. *The key-generation algorithm* KeyGen *takes as input a security parameter* 1^λ *and outputs a pair of public/private keys* $(\mathsf{pk}, \mathsf{sk})$.
2. P *and* V *follow the protocol in Figure 1, and at the end of this,* V *outputs* 1, *if it* ***accepts*** *that* P *knows* sk, *otherwise it* ***rejects*** *returning* 0.

A transcript of a 5*-pass* IDS *is a tuple* $(\mathsf{com}, \mathsf{ch}_1, \mathsf{rsp}_1, \mathsf{ch}_2, \mathsf{rsp}_2)$, *as in Figure 1, includes all the messages exchanged between* P *and* V *in one execution of the* IDS.

We require an IDS to fulfill the following security properties.

- **Correctness:** if for any security parameter $\lambda \in \mathbb{N}$ and $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$ it holds, $\Pr[1 \leftarrow \mathsf{V}(\mathsf{pk}, \mathsf{com}, \mathsf{ch}_1, \mathsf{rsp}_1, \mathsf{ch}_2, \mathsf{rsp}_2)] = 1$, where $(\mathsf{com}, \mathsf{ch}_1, \mathsf{rsp}_1, \mathsf{ch}_2, \mathsf{rsp}_2)$ is the transcript of an execution of the protocol between $\mathsf{P}(\mathsf{pk}, \mathsf{sk})$ and $\mathsf{V}(\mathsf{pk})$.
- **Soundness (with soundness error ε):** if, given a key pair $(\mathsf{pk}, \mathsf{sk})$, for every polynomial-time adversary $\mathcal{A}$ the difference

$$\Pr\left[\begin{array}{c}(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\lambda)\\ 1 \leftarrow V(\mathsf{pk}, \mathsf{com}_\mathcal{A}, \mathsf{ch}_1, \mathsf{rsp}_{1,\mathcal{A}}, \mathsf{ch}_2, \mathsf{rsp}_{2,\mathcal{A}})\end{array}\right] - \varepsilon$$

  is a negligible function in λ, where $(\mathsf{com}_\mathcal{A}, \mathsf{ch}_1, \mathsf{rsp}_{1,\mathcal{A}}, \mathsf{ch}_2, \mathsf{rsp}_{2,\mathcal{A}})$ is the transcript of one execution of the protocol between $\mathcal{A}$ and V both with input pk.
- **Honest-verifier zero-knowledge:** if there exists a polynomial-time probabilistic algorithm $\mathcal{S}(\mathsf{pk})$, called a *simulator*, that can produce transcripts (sequences of the form $(\mathsf{com}, \mathsf{ch}_1, \mathsf{rsp}_1, \mathsf{ch}_2, \mathsf{rsp}_2)$), that are computationally indistinguishable from the distribution of transcripts of an honest execution of the protocol between $\mathsf{P}(\mathsf{pk}, \mathsf{sk})$ and $\mathsf{V}(\mathsf{pk})$.

$$\begin{array}{lcl}
\mathsf{P}(\mathsf{pk}, \mathsf{sk}) & & \mathsf{V}(\mathsf{pk}) \\
\mathsf{com} \leftarrow P_0(\mathsf{pk}, \mathsf{sk}) & & \\
 & \xrightarrow{\mathsf{com}} & \\
 & & \mathsf{ch}_1 \xleftarrow{\$} \mathsf{ChallengeSet}_1 \\
 & \xleftarrow{\mathsf{ch}_1} & \\
\mathsf{rsp}_1 \leftarrow P_1(\mathsf{pk}, \mathsf{sk}, \mathsf{com}, \mathsf{ch}_1) & & \\
 & \xrightarrow{\mathsf{rsp}_1} & \\
 & & \mathsf{ch}_2 \xleftarrow{\$} \mathsf{ChallengeSet}_2 \\
 & \xleftarrow{\mathsf{ch}_2} & \\
\mathsf{rsp}_2 \leftarrow P_2(\mathsf{pk}, \mathsf{sk}, \mathsf{com}, \mathsf{ch}_1, \mathsf{rsp}_1, \mathsf{ch}_2) & & \\
 & \xrightarrow{\mathsf{rsp}_2} & \\
 & & V(\mathsf{pk}, \mathsf{com}, \mathsf{ch}_1, \mathsf{rsp}_1, \mathsf{ch}_2, , \mathsf{rsp}_2)
\end{array}$$

Fig. 1: Canonical 5-pass IDS.

2.5 MPC-in-the-Head : From MPC to Zero-Knowledge

MPC-in-the-Head (MPCitH) is a generic technique, introduced as "IKOS" [28], that allows to build a Zero-Knowledge Proof of Knowledge (ZKPoK) from a secure MPC protocol.

Consider a MPC protocol where N parties $P_1 \ldots, P_N$ collaborate to securely evaluate a public function f on a secret input x. Assuming that the protocol is perfectly correct and that the views of $t < N$ parties leak no information on x, then one can construct a ZKPoK from the MPC protocol as follows:

1. **Simulation**.
   - Prover P generates a random sharing $[\![x]\!] := ([\![x]\!]_1, \ldots, [\![x]\!]_N)$ of x such that $x = \sum_{i=1}^N [\![x]\!]_i$ and assign a share $[\![x]\!]_i$ to each party P_i.

  - P emulates "in his/her mind" execution of the MPC protocol with N parties $P_1 \ldots, P_N$.
  - P commits on the *views* of each P_i, meaning the messages they send/receive during the protocol execution and their internal states. These commitments are sent to the verifier V.
2. **Challenges**.
  - P possibly receives random challenges from V on the MPC, executes local computations accordingly and sends the results to V. This step can be repeated several times.
  - V challenges P to open a random subset of t parties.
  - P returns the requested views.
3. **Verification**.
  - P then checks that the views[10] are consistent, and the output of the circuit corresponds to the result expected.

Since its introduction, the initial approach for MPCitH from [28] has been improved in different ways. In particular, Katz, Kolesnikov and Wang (KKW, [31]) extended the MPCitH paradigm to support the *preprocessing model*, where MPC protocols are split into an offline phase that is independent of the sensitive inputs, and an online phase, with the former being typically the bottleneck in terms of efficiency. The benefit is that the prover does not need to include the preprocessing as part of the views of the parties, and instead, the preprocessing can be checked. As an application, KKW allowed to significantly decrease the signature size of the initial Picnic version.

In [34], the authors described the so-called hypercube variant of MPCitH that allows improving efficiency for a large number of parties in the MPC protocol. Indeed, a large number of parties leads to shorter signatures but increases signature generation and verification times. We detail the approach in the case of Biscuit in Section 3.1. Note that the hypercube technique is generic and could be then used for most MPCitH-based signature schemes.

2.6 Proof Systems for Arbitrary Circuits

In [27], Giacomelli, Madsen and Orlandi demonstrated the efficiency of the MPCitH approach for generating ZKPoK. Doing so, the authors also introduced a new generic proof system, called ZKBoo, which ultimately resulted in the first version of the Picnic signature scheme. In such work, the virtual/emulated parties actually *execute* some MPC protocols, and the verifier checks this execution. In [14], Baum and Nof proposed an improved proof system, called BN, for arithmetic circuits. The authors of [14] observed that the prover knows all the wire values in the circuit, and instead of computing a protocol, the prover can distribute sharings for each intermediate wire value, and the virtual parties only need to execute a protocol that checks the correctness of the multiplication gates. This allows batching the checks by taking random linear combinations. In [30], Kales and Zaverucha built on top of BN with several optimizations leading to BN++ with roughly 2.5× communication improvement.

The BN and BN++ proof systems rely on the concept of multiplicative triple (or Beaver triple [15]). Given $x, y, z \in \mathbb{F}_q$, we say that the triple $(\llbracket x \rrbracket, \llbracket y \rrbracket, \llbracket z \rrbracket) \in \mathbb{F}_q^N \times \mathbb{F}_q^N \times \mathbb{F}_q^N$ is

[10] If only one party is opened then there are no pairs to check consistency. In this case, the prover does not commit to the views, but actually to the point-to-point channels between the parties.

a *multiplicative triple* if it holds that $z = x \cdot y$. The Biscuit MPC protocol will rely on a somewhat standard protocol introduced in [14] (along with the optimization given in [30, Section 2.5]) to check multiplicative triples of sharing (Section 2.6). A multiplicative triple $(\llbracket x \rrbracket, \llbracket y \rrbracket, \llbracket z \rrbracket) \in \mathbb{F}_q^N \times \mathbb{F}_q^N \times \mathbb{F}_q^N$ can be checked using a *helping* triple $(\llbracket a \rrbracket, \llbracket y \rrbracket, \llbracket c \rrbracket) \in \mathbb{F}_q^N \times \mathbb{F}_q^N \times \mathbb{F}_q^N$ with $a \in \mathbb{F}_q$ and $c = a \cdot y \in \mathbb{F}_q$ as follows:

1. The parties get a random element $\varepsilon \xleftarrow{\$} \mathbb{F}_q$.
2. The parties locally set $\llbracket \alpha \rrbracket \leftarrow \llbracket x \rrbracket \cdot \varepsilon + \llbracket a \rrbracket$.
3. The parties open $\llbracket \alpha \rrbracket$ so that they all obtain α.
4. The party locally compute $\llbracket v \rrbracket = \llbracket y \rrbracket \cdot \alpha - \llbracket z \rrbracket \cdot \varepsilon - \llbracket c \rrbracket$.
5. The parties open $\llbracket v \rrbracket$ to obtain v.
6. The parties output **accept** if $v = 0$ and **reject** otherwise.

The security of this simple protocol has been proven in [30]. In particular, the false success probability is given by:

Lemma 2. *Let $x, y, z, a, c \in \mathbb{F}_q$. If the shared multiplicative triple $(\llbracket x \rrbracket, \llbracket y \rrbracket, \llbracket z \rrbracket) \in \mathbb{F}_q^N \times \mathbb{F}_q^N \times \mathbb{F}_q^N$ is incorrect, i.e. $z \neq x \cdot y$, or the helping multiplicative triple $(\llbracket a \rrbracket, \llbracket y \rrbracket, \llbracket c \rrbracket) \in \mathbb{F}_q^N \times \mathbb{F}_q^N \times \mathbb{F}_q^N$ is incorrect, i.e. $c \neq a \cdot y$, then the parties output **accept** with probability at most $1/q$.*

3 Interactive Protocols for PowAff2

This section describes the MPC protocol underlying Biscuit (Section 3.1) and the corresponding ZKPoK (Section 3.2) obtained using the MPCitH paradigm (Section 2.5) together with the hypercube technique [5].

3.1 Multi-Party Computation Protocol for PowAff2

In Figure 2, we detail the MPC protocol used in Biscuit to check a solution of a PowAff2 algebraic system. The protocol is executed by N parties sharing a secret vector $\mathbf{s} \in \mathbb{F}_q^n$. Every party knows the target vector $\mathbf{t} = (t_1, \ldots, t_m) \in \mathbb{F}_q^m$, affine forms $A_{1,0}, A_{1,1}, A_{1,2}, \ldots, A_{m,0}, A_{m,1}, A_{m,2} \in \mathbb{F}_q[x_1, \ldots, x_n]$ as in (1) and the corresponding PowAff2 algebraic equations $\mathbf{f} = (f_1, \ldots, f_m) \in \mathbb{F}_q[x_1, \ldots, x_n]^m$ defined as:

$$f_k = A_{k,0} + A_{k,1} \cdot A_{k,2}, \forall k \in [m]. \tag{3}$$

The MPC protocol (Figure 2) consists of m iterations of the multiplicative checking protocol described in Section 2.6. At the end of the protocol, the parties output **accept** indicating they are convinced that the shared vector $\mathbf{s}$ satisfies $\mathbf{t} = \mathbf{f}(\mathbf{s})$. Otherwise, they output **reject**.

The following proposition follows easily from Lemma 2.

Proposition 1. *Suppose that a set of N parties genuinely follow the MPC protocol given in Figure 2 with inputs $\mathbf{t} \in \mathbb{F}_q^m, \mathbf{f} = (f_1, \ldots, f_m) \in \mathbb{F}_q[x_1, \ldots, x_n]^m$, and $\llbracket \mathbf{s} \rrbracket \in \left(\mathbb{F}_q^n\right)^N$. Suppose $\mathbf{s} \in \mathbb{F}_q^n$ is a solution to $\mathtt{PowAff2}_u(\mathbf{f}, \mathbf{t})$ but not a solution to the $\mathtt{PowAff2}_{u-1}(\mathbf{f}, \mathbf{t})$. If $u = 0$, i.e., $\mathbf{t} = \mathbf{f}(\mathbf{s})$, then the parties **accept**. Otherwise, the parties **accept** with probability at most $1/q^u$.*

Public data: $\mathbf{t} = (t_1, \ldots, t_m) \in \mathbb{F}_q^m$, affine polynomials $A_{1,0}, \ldots, A_{m,2} \in \mathbb{F}_q[x_1, \ldots, x_n]$ and $\mathbf{f} = (f_1, \ldots, f_m) \in \mathbb{F}_q[x_1, \ldots, x_n]^m$ as defined in (3).

Inputs : The i-th party knows $[\![\mathbf{s}]\!]_i \in \mathbb{F}_q^n$, $[\![\mathbf{a}]\!]_i \in \mathbb{F}_q^m$ where $\mathbf{a} = (a_1, \ldots, a_m) \xleftarrow{\$} \mathbb{F}_q^m$, and $[\![\mathbf{c}]\!]_i \in \mathbb{F}_q^m$ where $\mathbf{c} = (c_1, \ldots, c_m) \in \mathbb{F}_q^m$ such that $c_k = A_{k,2}(\mathbf{s}) \cdot a_k, \forall k \in [m]$.

`MPC` **protocol**:

for $k \in [m]$ **do**

1: Each party compute $[\![z_k]\!] \leftarrow t_k - A_{k,0}([\![\mathbf{s}]\!]), [\![x_k]\!] \leftarrow A_{k,1}([\![\mathbf{s}]\!])$, and $[\![y_k]\!] \leftarrow A_{k,2}([\![\mathbf{s}]\!])$.

2: The parties get a random element $\varepsilon_k \xleftarrow{\$} \mathbb{F}_q$.

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 α_k.

5: The parties 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 $v_k = 0, \forall k \in [n]$ and **reject** otherwise.

Fig. 2: `MPC` protocol Π to check that $\mathbf{t} = \mathbf{f}(\mathbf{s})$.

3.2 Zero-Knowledge Proof of Knowledge for `PowAff2`

In Figure 3, we derive a zero-knowledge proof of knowledge (`ZKPoK`) for the `PowAff2` problem using the `MPC` protocol Π of Figure 2. We use the traditional `MPCitH` approach combined with the recent hypercube technique. To do so, let D be such that $N = 2^D$.

In Phase 1, for each $\ell \in [D]$: the prover generates an input set $S_\ell = \left([\![\mathbf{s}]\!]_{(\ell,j)}, [\![\mathbf{c}]\!]_{(\ell,j)}, [\![\mathbf{a}]\!]\right)_{j \in [2]}$ for a two parties instance the `MPC` protocol Π (Figure 2). The set S_ℓ is called the ℓ-th set of *main shares*. The sets of main shares are computed in two steps. First, the prover generates and commits to inputs $([\![\mathbf{s}]\!]_i, [\![\mathbf{c}]\!]_i, [\![\mathbf{a}]\!]_i)$ of one of $N = 2^D$ parties instance of Π. Then, for each $(\ell, j) \in [D] \times [2]$, the main share $[\![\mathbf{s}]\!]_{(\ell,j)}$ is computed as the sum of the shares $[\![\mathbf{s}]\!]_i$ for which j equals the ℓ-th bit of i plus 1. Similarly, the main shares $[\![\mathbf{c}]\!]_{(\ell,j)}$ and $[\![\mathbf{a}]\!]_{(\ell,j)}$). In Phase 3, the prover executes the protocol Π for every set of main shares using $\varepsilon_1, \ldots, \varepsilon_m \in \mathbb{F}_q$ as the random elements for all D executions. This particular execution of the protocol Π on the set of main shares S_ℓ is shown in Figure 4. The outputs of ℓ-th execution are the shares $\left([\![\alpha_k]\!]_{(\ell,j)}, [\![v_k]\!]_{(\ell,j)}\right)_{(k,j) \in [m] \times [2]}$ and its corresponding hash $H_\ell = \mathtt{H}\left(\left([\![\alpha_k]\!]_{(\ell,j)}, [\![v_k]\!]_{(\ell,j)}\right)_{(k,j) \in [m] \times [2]}\right)$ [11]. In Phase 5, the prover sends $\left((\mathsf{seed}^{(i)}, \rho_i)_{i \neq \bar{i}}, \mathsf{com}^{(\bar{i})}, \mathbf{\Delta s}, \mathbf{\Delta c}, [\![\boldsymbol{\alpha}]\!]_{\bar{i}}\right)$ to the verifier, where $[\![\boldsymbol{\alpha}]\!]_{\bar{i}} = ([\![\alpha_1]\!]_{\bar{i}}, \ldots, [\![\alpha_m]\!]_{\bar{i}})$, $[\![\alpha_k]\!]_{\bar{i}} = [\![x_k]\!]_{\bar{i}} \cdot \varepsilon_k + [\![a_k]\!]_{\bar{i}}$ and $[\![x_k]\!]_{\bar{i}} = A_{k,0}([\![s]\!]_{\bar{i}})$. We highlight that the prover does not send explicitly instead of sending $N-1$ strings of the form $(\mathsf{seed}^{(i)}, \rho_i)$ but it sends instead the $\log_2(N)$ nodes of the tree TreePRG(root) so that the verifier can recompute the values $(\mathsf{seed}^{(i)}, \rho_i)_{i \neq \bar{i}}$. Finally, in the verification phase, the verifier recomputes

[11] As noted in [10], the security of proof knowledge protocols using the hypercube technique with additive shares is the same with or without these intermediate hash values H_ℓ. Still, it might help reduce the protocol's memory demand when the implementation of the hash `H` is not incremental.

$(\mathsf{seed}^{(i)}, \rho_i)_{i \neq \bar{i}}$, and uses them to recompute the sets main shares partially. We say partially recompute and not just recompute because for each set S_ℓ one of the main shares triples (either the one corresponding to $j = 1$ or $j = 2$) is missing the addition of the shares corresponding to the $\bar{i}$-th party. After, for every set of main parties, the verifier follows the algorithm in Figure 5 to check the execution of the MPC protocol Π. Finally, the verifier recomputes h_0 and h_2 and outputs accept if these two values match the ones the prover sent. Otherwise, the verifier rejects.

The result below establishes the zero-knowledge property of the protocol described in Figure 3.

Theorem 1. *The protocol described in Figure 3 has the following properties:*

- ***Completeness.*** *A Prover with the knowledge of a solution $\mathbf{s} \in \mathbb{F}_q^n$ to an instance $(\mathbf{f}, \mathbf{t}) \in \mathbb{F}_q[x_1, \ldots, x_n]^m \times \mathbb{F}_q^m$ of the* `PowAff2` *is always accepted by the Verifier.*
- ***Soundness.*** *Let $\epsilon = \frac{1}{N} + \frac{1}{q^u} \cdot \left(1 - \frac{1}{N}\right)$, where $p = 1/q^u$. Suppose there exists a prover $\tilde{\mathcal{P}}$ who convinces the verifier to accept with probability $\tilde{\epsilon} > \epsilon$. Then there is an efficient probabilistic extraction algorithm $\mathcal{E}$, which has rewindable black-box access to $\tilde{\mathcal{P}}$, that, in expectation, with at most*

$$\frac{4}{\tilde{\epsilon} - \epsilon} \cdot \left(1 + \tilde{\epsilon} \cdot \frac{2\ln(2)}{\tilde{\epsilon} - \epsilon}\right),$$

  calls to $\tilde{P}$ outputs either a solution to an instance $(\mathbf{f}, \mathbf{t})$ of the $\mathtt{PowAff2}_{u-1}$ *problem or a collision to the commitment scheme Com or the hash* `H`*.*
- ***Honest-verifier zero-knowledge.*** *If the outputs of the pseudo-random generator PRG and the commitment scheme com are indistinguishable from the uniform random distribution, then the protocol of Figure 3 is honest-verifier zero-knowledge.*

Proof. (sketch) The proof is similar to, for instance, [10, Theorem 1]. Here, we describe the main parts of the proof and will refer [10, Theorem 1] for similar details.

- **Completeness.** By following, step by step, the protocol in Figure 3, it is not hard to see that a Prover that follows the protocol with inputs $(\mathbf{f}, \mathbf{t}, \mathbf{s})$ such that $\mathbf{t} = \mathbf{f}(\mathbf{s})$ will always be accepted.
- **Soundness.** The structure of the proof is as follows:
  1. We prove that a prover $\tilde{\mathcal{P}}$ who does not know any solution for the $\mathtt{PowAff2}_{u-1}$ problem can cheat with probability at most $\epsilon = \frac{1}{N} + \frac{1}{q^u} \cdot \left(1 - \frac{1}{N}\right)$.
  2. Assuming that
     (a) No collisions to Com nor `H` can be found.
     (b) There exists a cheater $\tilde{\mathcal{P}}$ who has cheating probability $\tilde{\epsilon} > \epsilon$.
     We show how to extract a solution for the $\mathtt{PowAff2}_{u-1}$ problem whenever rewindable black-box access to $\tilde{\mathcal{P}}$ is given.

  For part 1, suppose that at step 7 the vector $\boldsymbol{s} = [\![\boldsymbol{s}]\!]_1 + \cdots + [\![\boldsymbol{s}]\!]_N$ is not a solution of the $\mathtt{PowAff2}_{u-1}$ problem defined by $(\boldsymbol{f}, \boldsymbol{t})$. With such a vector $\boldsymbol{s}$ the prover can be accepted by the verifier in only two situations:
  - (*False-positive case*) The prover honestly follows the protocol, and for each $k \in [m]$, the value $v_k = y_k \alpha_k - z_k \varepsilon_k - c_k$, which is the value that would be obtained from a genuine execution of the MPC protocol with challenges ε_k (see Figure 2), equals to zero, or
  - (*Cheating case*) The prover dishonestly deviates from the protocol, yet the verifier believes that all the honest v_k are zero, but in reality, at least one of them is not.

$\mathsf{PoK}(\mathsf{Prover}(\mathbf{f},\mathbf{t},\mathbf{s}),\ \mathsf{Verifier}(\mathbf{f},\mathbf{t}))$

Phase 1: Prover commits to the inputs of the MPC protocol in Figure 4

$\mathsf{root} \xleftarrow{\$} \{0,1\}^\lambda, \quad \left(\mathsf{seed}^{(i)}, \rho^{(i)}\right)_{i\in[N]} \leftarrow \mathrm{TreePRG}(\mathsf{root})$

for $i \in [N]$ **do**

  $[\![\mathbf{s}]\!]_i\,, [\![\mathbf{c}]\!]_i\,, [\![\mathbf{a}]\!]_i\,, \leftarrow \mathtt{PRG}(\mathsf{seed}^{(i)})$

  $\mathsf{com}^{(i)} \leftarrow \mathsf{Com}\left(\mathsf{seed}^{(i)}, \rho_i\right)$

$h_0 \leftarrow \mathtt{H}(\mathsf{com}^{(1)}, \ldots, \mathsf{com}^{(N)})$, and send h_0 to $\mathsf{Verifier}$

$\mathbf{a} \leftarrow \sum_{i\in[N]} [\![\mathbf{a}]\!]_i\,, \quad \mathbf{c} \leftarrow \left(A_{k,2}(\mathbf{s}) \cdot a_k\right)_{k\in[m]}$

$\boldsymbol{\Delta}\mathbf{s} \leftarrow \mathbf{s} - \sum_{i\in[N]} [\![\mathbf{s}]\!]_i\,, \quad \boldsymbol{\Delta}\mathbf{c} \leftarrow \mathbf{c} - \sum_{i\in[N]} [\![\mathbf{c}]\!]_i$

$[\![\mathbf{s}]\!]_1 \leftarrow [\![\mathbf{s}]\!]_1 + \boldsymbol{\Delta}\mathbf{s}$ and $[\![\mathbf{c}]\!]_1 \leftarrow [\![\mathbf{c}]\!]_1 + \boldsymbol{\Delta}\mathbf{c}$

Initialize $[\![\mathbf{s}]\!]_p\,, [\![\mathbf{c}]\!]_p$ and $[\![\mathbf{a}]\!]_p$ to zero objects for each $p \in [D] \times [2]$

for $i \in [N]$ **do**

  $(i_1, \ldots, i_D) \leftarrow i$ // Binary representation of i.

  for $\ell \in [D]$ **do**

    $[\![\mathbf{s}]\!]_{(\ell, i_\ell+1)} \leftarrow [\![\mathbf{s}]\!]_{(\ell, i_\ell+1)} + [\![\mathbf{s}]\!]_i\,,\ [\![\mathbf{c}]\!]_{(\ell, i_\ell+1)} \leftarrow [\![\mathbf{c}]\!]_{(\ell, i_\ell+1)} + [\![\mathbf{c}]\!]_i$ and

    $[\![\mathbf{a}]\!]_{(\ell, i_\ell+1)} \leftarrow [\![\mathbf{a}]\!]_{(\ell, i_\ell+1)} + [\![\mathbf{a}]\!]_i$

Phase 2: First challenge

$\mathsf{Verifier}$ samples $\varepsilon_1, \ldots, \varepsilon_m \xleftarrow{\$} \mathbb{F}_q$ and sends them to Prover

Phase 3: Prover's first response // Prover executes MPC protocol for every set of main shares.

for $\ell \in [D]$ **do**

  Prover gets H_ℓ and $\left([\![\alpha_k]\!]_{(\ell,j)}\,, [\![v_k]\!]_{(\ell,j)} \right)_{(k,j)\in[m]\times[2]}$ from algo. in Figure 4

$h_1 \leftarrow \mathtt{H}\left(H_1, \ldots, H_D\right)$ and send h_1 to $\mathsf{Verifier}$

Phase 4: Second challenge

$\mathsf{Verifier}$ samples $\bar{i} \xleftarrow{\$} [N]$ and sends it to Prover

Phase 5: Prover's second response

$[\![\boldsymbol{\alpha}]\!]_{\bar{i}} \leftarrow ([\![\alpha_1]\!]_{\bar{i}}, \ldots, [\![\alpha_m]\!]_{\bar{i}})$, where $[\![\alpha_k]\!]_{\bar{i}} = [\![x_k]\!]_{\bar{i}} \cdot \varepsilon_k + [\![a_k]\!]_{\bar{i}}$, $[\![x_k]\!]_{\bar{i}} = A_{k,0}([\![s]\!]_{\bar{i}})$

$\mathsf{rsp} \leftarrow \left((\mathsf{seed}^{(i)}, \rho_i)_{i\neq\bar{i}}\,, \mathsf{com}^{(\bar{i})}, \boldsymbol{\Delta}\mathbf{s}, \boldsymbol{\Delta}\mathbf{c}, [\![\boldsymbol{\alpha}]\!]_{\bar{i}}\right)$ and send rsp to $\mathsf{Verifier}$

Verification:

$\mathsf{Verifier}$ partially recomputes $\left([\![\mathbf{s}]\!]_p\,, [\![\mathbf{c}]\!]_p\,, [\![\mathbf{a}]\!]_p \right)_{p\in[D]\times[2]}$ from $(\mathsf{seed}^{(i)}, \rho_i)_{i\neq\bar{i}}$ by following Phase 1 but skipping the steps involving a $\bar{i}$-th share or $\mathsf{seed}^{(\bar{i})}$

for $\ell \in [D]$ **do**

  $\mathsf{Verifier}$ gets H_ℓ and $\left([\![\alpha_k]\!]_{(\ell,j)}\,, [\![v_k]\!]_{(\ell,j)} \right)_{(k,j)\in[m]\times[2]}$ from algo. in Figure 5

$\mathsf{Verifier}$ accepts if and only if $h_0 = \mathtt{H}(\mathsf{com}^{(1)}, \ldots, \mathsf{com}^{(N)})$ and $h_1 = \mathtt{H}\left(H_1, \ldots, H_D\right)$, where $\mathsf{com}^{(i)} = \mathsf{Com}(\mathsf{seed}^{(i)}, \rho_i)$ for each $i \neq \bar{i}$.

Fig. 3: Proof of Knowledge protocol for PowAff2.

Inputs : A set of main shares $\left(\left([\![\mathbf{s}]\!]_{(\ell,j)}, [\![\mathbf{c}]\!]_{(\ell,j)}, [\![\mathbf{a}]\!]_{(\ell,j)} \right) \right)_{j \in [2]}$ and the challenges $\varepsilon_1, \ldots, \varepsilon_m$

Outputs : H_ℓ and $\left([\![\alpha_k]\!]_{(\ell,j)}, [\![v_k]\!]_{(\ell,j)} \right)_{(k,j) \in [m] \times [2]}$

for $k \in [m]$ **do**

  for $j \in [2]$ **do**

    $[\![x_k]\!]_{(\ell,j)} \leftarrow A_{k,1}([\![\mathbf{s}]\!]_{(\ell,j)})$

    $[\![\alpha_k]\!]_{(\ell,j)} \leftarrow [\![x_k]\!]_{(\ell,j)} \cdot \varepsilon_k + [\![a_k]\!]_{(\ell,j)}$

  $\alpha_k \leftarrow [\![\alpha_k]\!]_{(\ell,1)} + [\![\alpha_k]\!]_{(\ell,2)}$ // The parties open $[\![\alpha_k]\!]_{(\ell,j)}$ to obtain α_k.

  $[\![z_k]\!]_{(\ell,1)} \leftarrow t_k - A_{k,0}([\![\mathbf{s}]\!]_{(\ell,1)})$

  $[\![y_k]\!]_{(\ell,1)} \leftarrow A_{k,2}([\![\mathbf{s}]\!]_{(\ell,1)})$

  $[\![v_k]\!]_{(\ell,1)} \leftarrow [\![y_k]\!]_{(\ell,1)} \cdot \alpha_k - [\![z_k]\!]_{(\ell,1)} \cdot \varepsilon_k - [\![c_k]\!]_{(\ell,1)}$

  $[\![v_k]\!]_{(\ell,2)} \leftarrow - [\![v_k]\!]_{(\ell,1)}$

$H_\ell \leftarrow \mathtt{H}\left(\left([\![\alpha_k]\!]_{(\ell,j)}, [\![v_k]\!]_{(\ell,j)} \right)_{(k,j) \in [m] \times [2]} \right)$

Fig. 4: Simulation of the MPC protocol Π for the ℓ-th set of main shares.

In the first case, we would have a false positive case of the MPC protocol in Figure 2. By Proposition 1, this happens with probability at most $1/q^u$. In the second case, the prover cheats during the simulation of at least one party. Since the verifier checks the correct execution of all the parties but one, the prover has to cheat on exactly one party. Otherwise, the verifier rejects. Cheating in one party i' means that the prover uses a set of different shares than an honest party, holding the same input seed $\mathsf{seed}^{(i')}$, would use. Since every party aggregates to exactly one of the main shares for all of the D bi-party protocols. For each of these bi-party protocols, one share has been dishonestly computed, i.e., not following the MPC protocol. Thus, the prover won't be detected with probability $\frac{1}{N}$. Consequently, a prover without a correct solution of the $\mathtt{PowAff2}_{u-1}$ problem will be accepted with probability at most $\epsilon = \frac{1}{N} + \frac{1}{q^u} \cdot \left(1 - \frac{1}{N}\right)$.

Now, for the second part, we assume that no collisions to Com nor H can be found and there exists a cheater $\tilde{\mathcal{P}}$ who has cheating probability $\tilde{\epsilon} > \epsilon$. First, we prove that a solution $\boldsymbol{s}$ of the $\mathtt{PowAff2}_{u-1}$ problem can be extracted from two valid transcripts of the form $\mathcal{T}_1$ and $\mathcal{T}_2$ produced by $\tilde{\mathcal{P}}$ that have the same initial commitment h_0 and different second challenges $\bar{i}_1$ (for $\mathcal{T}_1$) and $\bar{i}_1$. Finally, we prove that such transcripts $\mathcal{T}_1$ and $\mathcal{T}_2$ can be extracted from $\tilde{P}$ (assuming rewindable black-box access to $\tilde{P}$) with an expected number of calls upper bounded by

$$\frac{4}{\tilde{\epsilon} - \epsilon} \cdot \left(1 + \tilde{\epsilon} \cdot \frac{2\ln(2)}{\tilde{\epsilon} - \epsilon}\right).$$

This second part is proven analogously as in [10, Theorem 1].

- **Honest-verifier zero-knowledge**: Now we sketch the proof of the honest-verifier zero-knowledge property of the protocol in Figure 3. The goal here is to show that the distribution of the transcripts output by the simulator described in Figure 6 on input $(\boldsymbol{f}, \boldsymbol{t})$ are indistinguishable from those coming from a genuine interaction

Inputs: Partially computed main shares $\Big(\big([\![\mathbf{s}]\!]_{(\ell,j)}, [\![\mathbf{c}]\!]_{(\ell,j)}, [\![\mathbf{a}]\!]_{(\ell,j)} \big) \Big)_{j \in [2]}$, the first challenges $\varepsilon_1, \ldots, \varepsilon_m$, the second challenge $\bar{i}$, and the $[\![\boldsymbol{\alpha}]\!]_{\bar{i}}$

Outputs: H_ℓ and $\big([\![\alpha_k]\!]_{(\ell,j)}, [\![v_k]\!]_{(\ell,j)} \big)_{(k,j) \in [m] \times [2]}$

$(\bar{i}_1, \ldots, \bar{i}_D) \leftarrow \bar{i}$ // Binary representation of $\bar{i}$.

$[\![\alpha_1]\!]_{\bar{i}}, \ldots, [\![\alpha_m]\!]_{\bar{i}} \leftarrow [\![\boldsymbol{\alpha}]\!]_{\bar{i}}$

for $k \in [m]$ **do**

  for $j \in [2]$ **do**

    $[\![x_k]\!]_{(\ell,j)} \leftarrow A_{k,1}([\![\mathbf{s}]\!]_{(\ell,j)})$

    $[\![\alpha_k]\!]_{(\ell,j)} \leftarrow [\![x_k]\!]_{(\ell,j)} \cdot \varepsilon_k + [\![a_k]\!]_{(\ell,j)}$

  $[\![\alpha_k]\!]_{(\ell,i_\ell+1)} \leftarrow [\![\alpha_k]\!]_{(\ell,i_\ell+1)} + [\![\alpha_k]\!]_{\bar{i}}$ // Adding missing share of $[\![\alpha_k]\!]_{(\ell,i_\ell+1)}$.

  $\alpha_k \leftarrow [\![\alpha_k]\!]_{(\ell,1)} + [\![\alpha_k]\!]_{(\ell,2)}$ // The parties open $[\![\alpha_k]\!]_{(\ell,j)}$ to obtain α_k.

  Set $i^* = 2$ if $\bar{i}_\ell = 0$, otherwise set $i^* = 1$.

  $[\![y_k]\!]_{(\ell,i^*)} \leftarrow A_{k,2}([\![\mathbf{s}]\!]_{(\ell,i^*)})$

  $[\![z_k]\!]_{(\ell,i^*)} \leftarrow t_k - A_{k,0}([\![\mathbf{s}]\!]_{(\ell,i^*)})$

  $[\![v_k]\!]_{(\ell,i^*)} \leftarrow [\![y_k]\!]_{(\ell,i^*)} \cdot \alpha_k - [\![z_k]\!]_{(\ell,i^*)} \cdot \varepsilon_k - [\![c_k]\!]_{(\ell,i^*)}$

  $[\![v_k]\!]_{(\ell,\bar{i}_\ell+1)} \leftarrow -[\![v_k]\!]_{(\ell,i^*)}$

$H_\ell \leftarrow \mathtt{H}\Big(\big([\![\alpha_k]\!]_{(\ell,j)}, [\![v_k]\!]_{(\ell,j)} \big)_{(k,j) \in [m] \times [2]} \Big)$

Fig. 5: Check the simulation of the MPC protocol Π in the ℓ-th set of main shares.

between a prover and an honest verifier, where the prover input is $(\boldsymbol{f}, \boldsymbol{t}, \boldsymbol{s})$ and $\boldsymbol{t} = \boldsymbol{f}(\boldsymbol{s})$.

The idea is to create a sequence of simulators that ends with the simulator described in Figure 6. The first simulator of the sequence consists of a legitimate prover, which holds a solution $\boldsymbol{s}$ and simulates the verifier by randomly sampling the challenges, as an honest verifier would do. These transcripts are indistinguishable from those coming from a legitimate execution of the protocol in proof of knowledge protocol.

Finally, the proof is completed by showing that the transcripts outputs by any simulator in the sequence are indistinguishable from those in the previous simulator. This implies that the transcripts of the simulator in Figure 6 are indistinguishable from those produced by the actual protocol. Details of this part follow similarly as shown in [10, Theorem 1].

□

4 Biscuit Signature Scheme

In this part, we describe the Biscuit signature scheme. It is obtained by applying the Fiat-Shamir transformation [26] to the zero-knowledge protocol given in Figure 3. The

Simulator$(\mathbf{f}, \mathbf{t})$

Sample first challenge: $\boldsymbol{\varepsilon} = (\varepsilon_1, \ldots, \varepsilon_m) \xleftarrow{\$} \mathbb{F}_q^m$
Sample second challenge: $\bar{i} \xleftarrow{\$} [N]$
$\mathsf{root} \xleftarrow{\$} \{0,1\}^\lambda$
$\left(\mathsf{seed}^{(i)}, \rho^{(i)}\right)_{i \in [N]} \leftarrow \mathrm{TreePRG}(\mathsf{root})$
for $i \in [N]$ **do**
  $[\![\mathbf{s}]\!]_i, [\![\mathbf{c}]\!]_i, [\![\mathbf{a}]\!]_i, \leftarrow \mathtt{PRG}(\mathsf{seed}^{(i)})$
  $\mathsf{com}^{(i)} \leftarrow \mathsf{Com}\left(\mathsf{seed}^{(i)}, \rho_i\right)$
$h_0 \leftarrow \mathtt{H}(\mathsf{com}^{(1)}, \ldots, \mathsf{com}^{(N)})$
$\boldsymbol{\Delta}\mathbf{s} \xleftarrow{\$} \mathbb{F}_q^n, \quad \boldsymbol{\Delta}\mathbf{c} \xleftarrow{\$} \mathbb{F}_q^m$
$[\![\mathbf{s}]\!]_1 \leftarrow [\![\mathbf{s}]\!]_1 + \boldsymbol{\Delta}\mathbf{s}$ and $[\![\mathbf{c}]\!]_1 \leftarrow [\![\mathbf{c}]\!]_1 + \boldsymbol{\Delta}\mathbf{c}$
Initialize $[\![\mathbf{s}]\!]_p, [\![\mathbf{c}]\!]_p$ and $[\![\mathbf{a}]\!]_p$ to zero objects for each $p \in [D] \times [2]$
for $i \in [N] \setminus \{\bar{i}\}$ **do**
  Simulate the i party to obtain $[\![\alpha_k]\!]_i$ and $[\![v_k]\!]_i$ for each $k \in [m]$
$[\![\alpha_k]\!]_{\bar{i}} \xleftarrow{\$} \mathbb{F}_q$ and $[\![v_k]\!]_i \xleftarrow{\$} \mathbb{F}_q$ for each $k \in [m]$
$\mathsf{com}^{(\bar{i})} \xleftarrow{\$} \{0,1\}^\lambda$
For each $(k, \ell, j) \in [m] \times [D] \times [2]$ compute $[\![\alpha_k]\!]_{(\ell,j)}$ and $[\![v_k]\!]_{(\ell,j)}$
Set $H_\ell \leftarrow \mathtt{H}\left(\left([\![\alpha_k]\!]_{(\ell,j)}, [\![v_k]\!]_{(\ell,j)}\right)_{(k,j) \in [m] \times [2]}\right)$ for each $\ell \in [D]$
$h_1 \leftarrow \mathtt{H}\left(H_1, \ldots, H_D\right)$
$\mathsf{rsp} \leftarrow \left((\mathsf{seed}^{(i)}, \rho_i)_{i \neq \bar{i}}, \mathsf{com}^{(\bar{i})}, \boldsymbol{\Delta}\mathbf{s}, \boldsymbol{\Delta}\mathbf{c}, [\![\boldsymbol{\alpha}]\!]_{\bar{i}}\right)$, where $[\![\boldsymbol{\alpha}]\!]_{\bar{i}} = ([\![\alpha_1]\!]_{\bar{i}}, \ldots, [\![\alpha_m]\!]_{\bar{i}})$
Output $(h_0, \boldsymbol{\varepsilon}, h_1, \bar{i}, \mathsf{rsp})$

Fig. 6: Honest-verifier zero-knowledge simulator.

corresponding signing, and verification algorithms are described in Figures 7 and 8, respectively.

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

$$f_k(x_1, \ldots, x_n) = A_{k,0}(x_1, \ldots, x_n) + A_{k,1}(x_1, \ldots, x_n) \cdot A_{k,2}(x_1, \ldots, x_n), \tag{4}$$

where $A_{1,0}, \ldots, A_{m,2} \in \mathbb{F}_q[x_1, \ldots, x_n]$ are random affine forms as in (1).

We use two seeds $\mathsf{seed}_{\mathbf{f}}, \mathsf{seed}_{\mathbf{s}} \in \{0,1\}^\lambda$ that are extended via $\mathtt{PRG}$ to obtain the public polynomials $\mathbf{f} \in \mathbb{F}_q[x_1, \ldots, x_n]^m$ and the secret vector $\mathbf{s} \in \mathbb{F}_q[x_1, \ldots, x_n]^m$. Finally, the vector $\mathbf{t} \in \mathbb{F}_q^m$ is computed as $\mathbf{t} = \mathbf{f}(\mathbf{s})$.

The signing procedure Biscuit.Sign is given in Figure 7. It takes as input a key-pair $(\mathsf{sk}, \mathsf{pk})$ and the message $\mathsf{msg} \in \{0,1\}^*$ to sign. It is obtained by applying the Fiat-

Shamir transformation to the ZKPoK for PowAff2 (Section 3.2) with $N = 2^D$ parties.

Remark 1. The notation $\mathbf{f} \leftarrow \texttt{PRG}(\mathsf{seed}_{\mathbf{f}})$ is a shortcut for extending the seed from a PRG and casting the bit string into a set of algebraic equations as in (4). Similarly, $\mathbf{s} \leftarrow \texttt{PRG}(\mathsf{seed}_{\mathsf{sk}})$ stands for extending the seed and interpreting the bit string as a vector in $\mathbb{F}_q^n$.

The verification process (Figure 8) is very similar to the signature process (Figure 7) as the verifier has to replay the MPC protocol for each of the N participants except one. The algorithm takes as input a message $\mathsf{msg} \in \{0,1\}^*$, a signature sig and a public-key pk. It returns a bit $b \in \{0,1\}$.

4.1 Parameters

Table 2 provides the parameter sets Biscuit, along with the corresponding size of the keys and signatures. Each parameter set aims to provide a security level of either I, III or V according to the NIST guidelines. A more detailed description of the claimed security level of each parameter set is given in Section 5.

Level	Version	λ	q	n	m	N	τ	Bit-Security	sk	pk	σ
I	short	128	256	50	52	256	18	143	16	68	5748
	fast					32	28	143			7544
III	short	192	256	89	92	256	25	207	24	116	12969
	fast					32	40	210			17784
V	short	256	256	127	130	256	33	272	32	162	23523
	fast					32	53	275			32575

Table 2: Parameters of Biscuit, bit security, public-key (pk), secret-key (sk) and signature (σ) sizes in bytes.

The size of the public-key is $\lambda + \log_2(q) \cdot m$ bits, the size of the secret-key is λ bits and the bit-size of the signature is:

$$\underbrace{6\lambda}_{\mathsf{salt},h_1,h_2} + \tau\left(\underbrace{(n+2m)\log_2 q}_{\mathbf{\Delta s}^{(e)},\mathbf{\Delta c}^{(e)},[\![\boldsymbol{\alpha}]\!]_{\bar{i}_e}^{(e)}} + \underbrace{\lambda \cdot D}_{(\mathsf{seed}^{(e,i)})_{i\neq \bar{i}_e}} + \underbrace{2\lambda}_{\mathsf{com}^{(e,\bar{i}_e)}}\right).$$

5 Security Analysis

This part is dedicated to the security analysis of Biscuit against key-recovery (Section 5.2) and forgery (Section 5.3) attacks. Before that, Section 5.1 discusses the motivations for using structured systems as PowAff and the connection with the Learning With Errors (LWE, [35]) problem.

$\mathsf{Sign}(\mathsf{pk}, \mathsf{sk}, \mathsf{msg})$

$\big(\mathsf{seed}_{\mathbf{f}}, \mathbf{t}\big) \leftarrow \mathsf{pk},\ \ \mathsf{seed}_{\mathsf{sk}} \leftarrow \mathsf{sk}$

$\mathbf{f} \leftarrow \mathtt{PRG}(\mathsf{seed}_{\mathbf{f}}),\ \mathbf{s} \leftarrow \mathtt{PRG}(\mathsf{seed}_{\mathsf{sk}})$

Step 1: Commit to the inputs of the $\mathtt{MPC}$ protocol in Figure 4

$\mathsf{salt} \xleftarrow{\$} \{0,1\}^{2\lambda}$

for $e \in [\tau]$

$\mathsf{root}^{(e)} \xleftarrow{\$} \{0,1\}^{\lambda},\ \big(\mathsf{seed}^{(e,i)}\big)_{i\in[N]} \leftarrow \mathrm{TreePRG}(\mathsf{salt}, \mathsf{root}^{(e)})$

for $i \in [N]$ **do**

$[\![\mathbf{s}]\!]_i^{(e)}, [\![\mathbf{c}]\!]_i^{(e)}, [\![\mathbf{a}]\!]_i^{(e)} \leftarrow \mathtt{PRG}(\mathsf{seed}^{(e,i)})$

$\mathsf{com}^{(e,i)} \leftarrow \mathtt{H}_0\left(\mathsf{salt}, e, i, \mathsf{seed}^{(e,i)}\right)$

$\mathbf{a}^{(e)} \leftarrow \sum_{i\in[N]} [\![\mathbf{a}]\!]_i^{(e)}, \quad \mathbf{c}^{(e)} \leftarrow \left(A_{k,2}(\mathbf{s}) \cdot a_k^{(e)}\right)_{k\in[m]}$

$\mathbf{\Delta s}^{(e)} \leftarrow \mathbf{s} - \sum_{i\in[N]} [\![\mathbf{s}]\!]_i^{(e)}, \quad \mathbf{\Delta c}^{(e)} \leftarrow \mathbf{c}^{(e)} - \sum_{i\in[N]} [\![\mathbf{c}]\!]_i^{(e)}$

$[\![\mathbf{s}]\!]_1^{(e)} \leftarrow [\![\mathbf{s}]\!]_1^{(e)} + \mathbf{\Delta s}^{(e)}$ and $[\![\mathbf{c}]\!]_1^{(e)} \leftarrow [\![\mathbf{c}]\!]_1^{(e)} + \mathbf{\Delta c}^{(e)}$

$h_0^{(e)} \leftarrow \mathtt{H}_1(\mathsf{salt}, e, \mathsf{com}^{(e,1)}, \ldots, \mathsf{com}^{(e,N)}, \mathbf{\Delta s}^{(e)}, \mathbf{\Delta c}^{(e)})$

Initialize $[\![\mathbf{s}]\!]_p^{(e)}, [\![\mathbf{c}]\!]_p^{(e)}$ and $[\![\mathbf{a}]\!]_p^{(e)}$ to zero objects for each $p \in [D] \times [2]$

for $i \in [N]$ **do**

$(i_1, \ldots, i_D) \leftarrow i$ // Binary representation of i.

for $\ell \in [D]$ **do**

$[\![\mathbf{s}]\!]_{(\ell, i_\ell+1)}^{(e)} \leftarrow [\![\mathbf{s}]\!]_{(\ell, i_\ell+1)}^{(e)} + [\![\mathbf{s}]\!]_i^{(e)},\ [\![\mathbf{c}]\!]_{(\ell, i_\ell+1)}^{(e)} \leftarrow [\![\mathbf{c}]\!]_{(\ell, i_\ell+1)}^{(e)} + [\![\mathbf{c}]\!]_i^{(e)}$ and

$[\![\mathbf{a}]\!]_{(\ell, i_\ell+1)}^{(e)} \leftarrow [\![\mathbf{a}]\!]_{(\ell, i_\ell+1)}^{(e)} + [\![\mathbf{a}]\!]_i^{(e)}$

$h_1 \leftarrow \mathtt{H}_2\left(\mathsf{salt}, \mathsf{msg}, h_0^{(1)}, \ldots, h_0^{(\tau)}\right)$

Step 2: First challenge

$\left((\varepsilon_1^{(e)}, \ldots, \varepsilon_m^{(e)})\right)_{e\in[\tau]} \xleftarrow{\$} \mathtt{PRG}(h_1)$

Step 3: First response

for $e \in [\tau]$ **do**

for $\ell \in [D]$ **do**

Follow the algorithm in Figure 4 to get $H_\ell^{(e)}$, which is defined instead as

$H_\ell^{(e)} = \mathtt{H}_3\left(\mathsf{salt}, \ell, [\![\alpha_k]\!]_{(\ell,j)}^{(e)}, [\![v_k]\!]_{(\ell,j)}^{(e)}\,\right)_{(k,j)\in[m]\times[2]}\Big)$

$h_2 \leftarrow \mathtt{H}_4\Big(\mathsf{salt}, \mathsf{msg}, h_1, (H_1^{(e)}, \ldots, H_D^{(e)})_{e\in[\tau]}\Big)$

Step 4: Second challenge

$\bar{i}_1, \ldots, \bar{i}_\tau \xleftarrow{\$} \mathtt{PRG}(h_2)$

Step 5: Second response

for $e \in [\tau]$ **do**

$[\![\boldsymbol{\alpha}]\!]_{\bar{i}_e}^{(e)} \leftarrow ([\![\alpha_1]\!]_{\bar{i}_e}^{(e)}, \ldots, [\![\alpha_m]\!]_{\bar{i}_e}^{(e)})$, where $[\![\alpha_k]\!]_{\bar{i}_e}^{(e)} = [\![x_k]\!]_{\bar{i}_e}^{(e)} \cdot \varepsilon_k^{(e)} + [\![a_k]\!]_{\bar{i}_e}^{(e)}$, and $[\![x_k]\!]^{(e)} = A_{k,1}([\![\mathbf{s}]\!]_{\bar{i}_e}^{(e)})$

$\sigma \leftarrow \Big(\mathsf{salt}, h_1, h_2, \big((\mathsf{seed}^{(e,i)})_{i\neq\bar{i}_e}\ , \mathsf{com}^{(e,\bar{i}_e)}\big)_{e\in[\tau]}, \big(\mathbf{\Delta s}^{(e)}, \mathbf{\Delta c}^{(e)}, [\![\boldsymbol{\alpha}]\!]_{\bar{i}_e}^{(e)}\,\big)_{e\in[\tau]}\Big)$

Output σ

Fig. 7: Biscuit signing algorithm.

$\mathsf{Verify}(\mathsf{pk}, \sigma, \mathsf{msg})$

$\big(\mathsf{seed}_{\mathbf{f}}, \mathbf{t}\big) \leftarrow \mathsf{pk},\ \mathbf{f} \leftarrow \mathtt{PRG}(\mathsf{seed}_{\mathbf{f}})$

Step 1: Parse signature

$\Big(\mathsf{salt}, h_1, h_2, \big((\mathsf{seed}^{(e,i)})_{i \neq \bar{i}_e}\ , \mathsf{com}^{(e,\bar{i}_e)}\big)_{e \in [\tau]}, \big(\boldsymbol{\Delta}\mathbf{s}^{(e)}, \boldsymbol{\Delta}\mathbf{c}^{(e)}, [\![\boldsymbol{\alpha}]\!]^{(e)}_{\bar{i}_e}\big)_{e \in [\tau]}\Big) \leftarrow \sigma$

$\big((\varepsilon_1^{(e)}, \ldots, \varepsilon_m^{(e)})\big)_{e \in [\tau]} \xleftarrow{\$} \mathtt{PRG}(h_1)$

$\bar{i}_1, \ldots, \bar{i}_\tau \xleftarrow{\$} \mathtt{PRG}(h_2)$

Step 2: Recompute h_1 and the inputs of the $\mathtt{MPC}$ protocol

  for $e \in [\tau]$

    for $i \in [N] \setminus \{\bar{i}_e\}$ **do**

      $[\![\mathbf{s}]\!]_i^{(e)}, [\![\mathbf{c}]\!]_i^{(e)}, [\![\mathbf{a}]\!]_i^{(e)} \leftarrow \mathtt{PRG}(\mathsf{seed}^{(e,i)})$

      $\mathsf{com}^{(e,i)} \leftarrow \mathtt{H}_0\big(\mathsf{salt}, e, i, \mathsf{seed}^{(e,i)}\big)$

    $h_0^{(e)} \leftarrow \mathtt{H}_1(\mathsf{salt}, e, \mathsf{com}^{(e,1)}, \ldots, \mathsf{com}^{(e,N)}, \boldsymbol{\Delta}\mathbf{s}^{(e)}, \boldsymbol{\Delta}\mathbf{c}^{(e)})$

    if $\bar{i}_e \neq 1$ **then**

      $[\![\mathbf{s}]\!]_1^{(e)} \leftarrow [\![\mathbf{s}]\!]_1^{(e)} + \boldsymbol{\Delta}\mathbf{s}^{(e)}$ and $[\![\mathbf{c}]\!]_1^{(e)} \leftarrow [\![\mathbf{c}]\!]_1^{(e)} + \boldsymbol{\Delta}\mathbf{c}^{(e)}$

    Initialize $[\![\mathbf{s}]\!]_p^{(e)}, [\![\mathbf{c}]\!]_p^{(e)}$ and $[\![\mathbf{a}]\!]_p^{(e)}$ to zero objects for each $p \in [D] \times [2]$

    for $i \in [N] \setminus \{\bar{i}_e\}$ **do**

      $(i_1, \ldots, i_D) \leftarrow i$ // Binary representation of i.

      for $\ell \in [D]$ **do**

        $[\![\mathbf{s}]\!]^{(e)}_{(\ell, i_\ell+1)} \leftarrow [\![\mathbf{s}]\!]^{(e)}_{(\ell, i_\ell+1)} + [\![\mathbf{s}]\!]_i^{(e)}$, $[\![\mathbf{c}]\!]^{(e)}_{(\ell, i_\ell+1)} \leftarrow [\![\mathbf{c}]\!]^{(e)}_{(\ell, i_\ell+1)} + [\![\mathbf{c}]\!]_i^{(e)}$ and

        $[\![\mathbf{a}]\!]^{(e)}_{(\ell, i_\ell+1)} \leftarrow [\![\mathbf{a}]\!]^{(e)}_{(\ell, i_\ell+1)} + [\![\mathbf{a}]\!]_i^{(e)}$

$\bar{h}_1 \leftarrow \mathtt{H}_2\big(\mathsf{salt}, \mathsf{msg}, h_0^{(1)}, \ldots, h_0^{(\tau)}\big)$

Step 3: Recompute h_2

  for $e \in [\tau]$ **do**

    for $\ell \in [D]$ **do**

      Use $(\varepsilon_1^{(e)}, \ldots, \varepsilon_m^{(e)})$, $[\![\boldsymbol{\alpha}]\!]^{(e)}_{\bar{i}_e}$ and the ℓ-th set of main shares as inputs in

      the algorithm in Figure 5 to get $\overline{H}_\ell^{(e)}$, which is defined instead as

      $\overline{H}_\ell^{(e)} = \mathtt{H}_3\big(\mathsf{salt}, \ell, [\![\alpha_k]\!]^{(e)}_{(\ell,j)}, [\![v_k]\!]^{(e)}_{(\ell,j)}\big)_{(k,j) \in [m] \times [2]}\big)$

$\bar{h}_2 \leftarrow \mathtt{H}_4\Big(\mathsf{salt}, \mathsf{msg}, \bar{h}_1, \big(\overline{H}_1^{(e)}, \ldots, \overline{H}_D^{(e)}\big)_{e \in [\tau]}\Big)$

Step 4: Verify signature

Output $(\bar{h}_1 = h_1) \wedge (\bar{h}_2 = h_2)$

Fig. 8: Biscuit verification algorithm.

From now on, let $\big(\mathbf{f} = (f_1,\dots,f_m), \mathbf{t} = \mathbf{f}(\mathbf{s})\big) \in \mathbb{F}_q[x_1,\dots,x_n]^m \times \mathbb{F}_q^m$ be a Biscuit public-key and $\mathbf{s} \in \mathbb{F}_q^n$ be the corresponding secret-key.

5.1 About the Hardness of PowAff2

A fundamental assumption in the design of Biscuit is that solving algebraic systems generated essentially from the power of affine forms are not much easier to solve than a random system of quadratic equations. Whilst the complexity of solving structured equations can be difficult to assess in general, the hardness of solving random quadratic equations has been deeply investigated and only exponential algorithms are known, e.g. [12,16,17,18].

We emphasize PowAff2 algebraic equations already appeared previously in the literature. In particular, the authors of [7,11] demonstrated that attacking the Learning With Errors (LWE) problem [35] reduces to solve a structured algebraic system similar to PowAff2. An instance of LWE is given by a pair $(\boldsymbol{A} = \{a_{i,j}\}, \boldsymbol{c} = \boldsymbol{s}\boldsymbol{A} + \boldsymbol{e}) \in \mathbb{F}_q^{n\times m} \times \mathbb{F}_q^m$ where $\boldsymbol{s} \in \mathbb{F}_q^n$ is a secret and $\boldsymbol{e} \in \mathbb{F}_q^m$ is an error vector. LWE (search) asks to recover the secret $\boldsymbol{s}$. Arora and Ge exhibit in [7,11] a rather natural algebraic modeling of LWE. More precisely, Arora and Ge show that LWE secrets can be recovered by solving:

$$f_1(x_1,\dots,x_n) = P(c_1 - \sum_{k=1}^{n} a_{k,1}x_k) = 0, \dots, f_m(x_1,\dots,x_n) = P(c_1 - \sum_{k=1}^{n} a_{k,m}x_k) = 0, \quad (5)$$

where P depends on the error distribution. In particular, $P(X) = X(X-1) \in \mathbb{F}_q[X]$ for binary errors and [7] introduced the assumption that a system such as (5) behaves such as a semi-regular sequence. As a consequence, a new fast algorithm for PowAff2 will lead to a new fast algebraic algorithm for binary LWE.

5.2 Key Recovery Attacks

A key-recovery attack against Biscuit consists of solving the PowAff2 problem, i.e. recovering $\mathbf{s} \in \mathbb{F}_q^m$ from the system defined as :

$$\mathbf{t} = \mathbf{f}(\mathbf{x}), \text{ with } \mathbf{x} = (x_1,\dots,x_n). \quad (6)$$

Currently, the best attack against Biscuit is a dedicated hybrid approach for solving PowAff2 equations described in [20]. The hybrid approach is a classical technique for solving algebraic systems that combines exhaustive search and a Gröbner basis-like computations [12,17,18]. The efficiency of such approach is related to the choice of a *trade-off*, denoted $k \le n$, between these two methods.

We sketch below the approach described in [20]. Let $\mathbf{g} = \big(g_1(\mathbf{x}) = u_1(\mathbf{x})\cdot(x_1 + c_1) + w_1(\mathbf{x}),\dots,g_n(\mathbf{x}) = u_n(\mathbf{x})\cdot(x_n + c_n) + w_n(\mathbf{x})\big) \in \mathbb{F}_q[x_1,\dots,x_n]^n$, with $\mathbf{x} = (x_1,\dots,x_n), u_1,\dots,u_n, v_1,\dots,v_n \in \mathbb{F}_q[x_1,\dots,x_n]$ affine polynomials and $c_1,\dots,c_n \in \mathbb{F}_q$. According to Lemma 1, with high probability, there exists $\mathbf{L} \in \mathrm{GL}_n(\mathbb{F}_q)$ such that:

$$\mathbf{f}\big(\mathbf{x}\cdot\mathbf{L}\big) = \Big(\mathbf{g}, A'_{n+1,0}(\mathbf{x}) + \prod_{j=1}^{2} A'_{n+1,j}(\mathbf{x}),\dots,A'_{m,0}(\mathbf{x}) + \prod_{j=1}^{2} A'_{m,j}(\mathbf{x})\Big)$$

where $A_{n+1,0}, A_{n+1,1}, A_{n+1,2},\dots,A_{m,0}, A_{m,1}, A_{m,2} \in \mathbb{F}_q[x_1,\dots,x_n]$ affine forms.

Then, for every guess $(a_1,\dots,a_k) \in \mathbb{F}_q^k$ of the k first variables $(x_1,\dots,x_k)$, we obtain k linear polynomials, namely $g_1(a_1,\dots,a_k,x_{k+1},\dots,x_n),\dots,g_k(a_1,\dots,a_k,x_{k+1},\dots,x_n)$.

These k linear polynomials are expected to be linearly independent with a probability close to $1 - 1/q$. Hence we can use them to substitute k additional variables in the remaining polynomials. The attack is finalized by solving the resulting quadratic system of $m - k$ equations in $n - 2k$ variables.

Complexity. The cost of the attack is dominated by

$$\min_{0 \le k < \frac{n}{2}} q^k \cdot \mathrm{MQ}(n - 2k, m - k, q), \tag{7}$$

where $\mathrm{MQ}(n, m, q)$ denotes the complexity of solving a random system of m quadratic equations over n variables over $\mathbb{F}_q$. To compute the exact complexity, we rely on the `MQEstimator` software tool, which is part of the more general `CryptographicEstimators`[12] library [23].

5.3 Forgery Attacks

In the context of forgery, the attacker has to solve the $\mathtt{PowAff2}_u$ problem (Definition 1), which is a variant of the problem considered before for key-recovery (Section 5.2). In the $\mathtt{PowAff2}_u$ problem, the goal is to find a vector $\mathbf{s}' \in \mathbb{F}_q^n$ that vanishes a subset of size $m - u$ of the system (6). Without loss of generality, we assume that $\mathbf{s}'$ vanishes the first $m - u$ polynomials and not the remaining equations. That is, $f_k(\mathbf{s}') = t_k$, for $k \in [m - u]$, and $f_k(\mathbf{s}') \ne t_k$ for $k = m - u + 1, \ldots, m$.

By Proposition 1, a set of N parties that follows the `MPC` protocol in Figure 2 on inputs $[\![\mathbf{s}']\!]$ and $(\mathbf{f}, \mathbf{t})$ will output **accept** with false positive rate $p_1 = 1/q^u$.

Thanks to Kales and Zaverucha, [30], it is known that `MPCitH`-based signature scheme that consists of τ repetitions of a `MPC` protocol with false positive rate p_1 can be forged by computing on average

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

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 $\mathbf{t} = \mathbf{f}(\mathbf{x})$ of size $m - u$ and $\mathbf{s}' \in \mathbb{F}_q^n$ be a solution that vanishes the equations of S. Then, $\mathbf{s}'$ might, by chance, be a solution of any equation in S^c, i.e., any equation that is not in S. If there remain $k \in [u]$ equations in S^c for which $\mathbf{s}'$ is not a solution, then an attacker can mount a forgery attack with complexity $\mathtt{KZ}_\tau(q^{-k}, N^{-1})$.

Let $(\mathbf{f}, \mathbf{t})$ be a Biscuit public-key selected uniformly at random, and let S be a subset of the equations $\mathbf{t} = \mathbf{f}(\mathbf{x})$ of size $m - u$ selected uniformly at random. Then, a random solution $\mathbf{s}' \in \mathbb{F}_q^n$ of the equations in S follows a uniform distribution. Hence, $f_k(\mathbf{s}')$ is a uniform element in $\mathbb{F}_q$. Therefore, the probability that $\mathbf{s}'$ 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 $\mathbf{s}'$ is not the solution of at most k equations in S^c, then,

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

[12] https://github.com/Crypto-TII/CryptographicEstimators

In order to secure Biscuit against forgery attacks, we must have for every pair (k, u), where $0 \leq k \leq u \leq m$:

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

where $C_\lambda = 15$ if $\lambda = 128$ or 192 and $C_\lambda = 16$ otherwise.

Following these analyses, we propose in Table 2 a set of 3 parameters for $128, 192$ and 256 bits of classical security.

5.4 Existential Unforgeability

The existential unforgeability of Biscuit is stated in Theorem 2.

Theorem 2 (EU-CMA security). *Let PRG be a (t, ϵ_{PRG})-secure pseudo-random generator function, and that any adversary running in time t has an advantage of at most $\epsilon_{\mathtt{PowAff2}}$ against the underlying $\mathtt{PowAff2}_{u-1}$ problem. Suppose that the hash functions $\mathtt{H}_0, \mathtt{H}_1, \mathtt{H}_2$ $\mathtt{H}_4$ behave as random oracles that output binary strings of size 2λ. Let $\mathcal{A}$ be an adversary who has access to a signing oracle, making q_i queries to $\mathtt{H}_i$ and q_s queries to the signing oracle. Then, the probability that $\mathcal{A}$ outputs a forgery for the* Biscuit *signature scheme (Figure 7) is:*

$$\Pr[\mathit{Forge}] \leq \frac{3(q + \tau N \cdot q_s)^2}{2 \cdot 2^{2\lambda}} + \frac{q_s(q_s + 5q)}{2^{2\lambda}} + \epsilon_{PRG} + \epsilon_{\mathtt{PowAff2}} + \Pr[X + Y = \tau],$$

where τ is the number of repetitions of the ZKPoK *protocol (Figure 3), $X = \max_{i \in [q_2]}\{X_i\}$ with $X_i \sim \mathcal{B}(\tau, \frac{1}{q^u})$, and $Y = \max_{i \in [q_4]}\{Y_i\}$ with $Y_i \sim \mathcal{B}(\tau - X, \frac{1}{N})$.*

Proof. Overall the proof works as follows: First, we assume the existence of an adversary $\mathcal{A}$ that can forge Biscuit signatures with probability $\Pr[\mathsf{Forge}]$ after interacting with a signing oracle and the random oracles $\mathtt{H}_0, \mathtt{H}_1, \mathtt{H}_2, \mathtt{H}_3$ and $\mathtt{H}_4$. Then, we show how to simulate such an interaction so that we can use $\mathcal{A}$ to either:

1. Find collisions on the oracles $\mathtt{H}_0, \mathtt{H}_1$, or $\mathtt{H}_3$.
2. query an oracle $\mathtt{H}_i$ with an input used to query $\mathtt{H}_i$ while replaying signing query,
3. distinguish between outputs of PRG from random ones,
4. solve an instance of the $\mathtt{PowAff2}_{u-1}$ problem, or
5. obtain an event that happens with probability at most $\Pr[X + Y = \tau]$.

In $\mathbf{Game}_1$, we simulate for $\mathcal{A}$ a real interaction with the signature scheme and the random oracles $\mathtt{H}_i$.

$\mathbf{Game}_1$: We generate a pair $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}()$, give pk to the adversary $\mathcal{A}$, simulate the random oracles $\mathtt{H}_i$, and any signing query msg from $\mathcal{A}$ is replied with $\mathsf{Sign}(\mathsf{pk}, \mathsf{sk}, \mathsf{msg})$, where Sign is the algorithm shown in Figure 7. We allow $\mathcal{A}$ to make q_i queries to $\mathtt{H}_i$ and q_s queries to the signing oracle. At the end, $\mathcal{A}$ outputs a pair (msg, σ). We denote by Forge the event where (msg, σ) is a forgery, i.e., σ is a valid signature for the message msg, and msg was not queried for signing.

For each of the subsequent games, $\Pr_i[\mathsf{Forge}]$ denotes the probability that Forge happens in $\mathbf{Game}_i$. In particular, we are interested in an upper bound for $\Pr[\mathsf{Forge}] = \Pr_1[\mathsf{Forge}]$.

Game$_2$: We proceed as in **Game**$_1$ with the only exception that we abort if, during the game, a collision of $\mathtt{H}_0$, $\mathtt{H}_1$, or $\mathtt{H}_3$ is found.

Every signing query yields τN queries to $\mathtt{H}_0$, τ to $\mathtt{H}_1$, and τD to $\mathtt{H}_3$, and one to $\mathtt{H}_2$ and $\mathtt{H}_4$. Hence, during this game, the total number of queries to $\mathtt{H}_0$, $\mathtt{H}_1$ or $\mathtt{H}_3$ is at most $q + \tau N q_s$, where $q = \max\{q_0, q_1, q_3\}$. Therefore, using the classic bound for the probability of a collision of a hash function[13], we have that

$$\left|\Pr_1[\mathsf{Forge}] - \Pr_2[\mathsf{Forge}]\right| \leq \frac{3(q + \tau N q_s)^2}{2^{2\lambda+1}}.$$

Game$_3$: We proceed as in **Game**$_2$, but we abort if, while replying to a signing query, the input to any $\mathtt{H}_i$ was used to answer a previous query to $\mathtt{H}_i$ made either directly by $\mathcal{A}$ or by another signing query.

For each signing query, the probability of aborting in this game is, at most, the probability that the salt sampled in the signature query is equal to a salt used in a previous query to any $\mathtt{H}_i$. Therefore, we have that

$$\left|\Pr_2[\mathsf{Forge}] - \Pr_3[\mathsf{Forge}]\right| \leq \frac{q_s(q_s + q_0 + q_1 + q_2 + q_3 + q_4)}{2^{2\lambda}} \leq \frac{q_s(q_s + 5 \cdot q)}{2^{2\lambda}}.$$

Game$_4$: This game differs from the previous one in how the signing queries are replied. In this case, instead of querying $\mathtt{H}_2$ and $\mathtt{H}_4$ to obtain h_1 and h_2, respectively. The values h_1 and h_2 are sampled uniformly at random from $\{0,1\}^{2\lambda}$.

Notice that **Game**$_3$ and **Game**$_4$ differ only in the case of a query to either $\mathtt{H}_2$ or $\mathtt{H}_4$ is repeated while answering a signing query. This cannot happen since we would have already aborted. So,

$$\Pr_4[\mathsf{Forge}] = \Pr_3[\mathsf{Forge}].$$

Game$_5$: This game changes how the signing queries are answered. We highlight that, in this game, the private key is no longer used to answer signing queries. Here, the values h_1, h_2, the salt and all the seeds $(\mathsf{seed}^{(e,i)})$ are computed as in **Game**$_4$. Contrarily, for each $e \in [\tau]$, the values $(\varepsilon_1^{(e)}, \ldots, \varepsilon_m^{(e)}), \bar{i}_e$, $\mathsf{com}^{(e,\bar{i}_e)}, \mathbf{\Delta s}^{(e)}, \mathbf{\Delta c}^{(e)}$ and $[\![\boldsymbol{\alpha}]\!]_{\bar{i}_e}^{(e)}$ are sampled uniformly at random as it is done by the $\mathsf{Simulator}$ (see Figure 6). From the security of the $\mathtt{PRG}$ we obtain that

$$\left|\Pr_4[\mathsf{Forge}] - \Pr_5[\mathsf{Forge}]\right| \leq \varepsilon_{\mathtt{PRG}}.$$

Now we introduce a definition. Let $e^* \in [\tau]$ and Q_4 be a query to $\mathtt{H}_4$ with input

$$\left(\mathsf{salt}, \mathsf{msg}, \mathsf{pk}, h_1, (H_1^{(e)}, \ldots, H_D^{(e)})_{e \in [\tau]}\right).$$

We say that the e^*-th execution of Q_4 defines a *good witness* $\mathbf{s}$ if

1. Each $H_\ell^{(e)}$ is an output of a query to $\mathtt{H}_3$.
2. There is a previous query $h_1 \leftarrow \mathtt{H}_2\left(\mathsf{salt}, \mathsf{msg}, h_0^{(1)}, \ldots, h_0^{(\tau)}\right)$.
3. There are previous queries $h_0^{(e)} \leftarrow \mathtt{H}_1(\mathsf{salt}, e, \mathsf{com}^{(e,1)}, \ldots, \mathsf{com}^{(e,N)}, \mathbf{\Delta s}^{(e)}, \mathbf{\Delta c}^{(e)})$, for $e \in [\tau]$.
4. For each $(e,i) \in [\tau] \times [N]$, there is a query of the form $\mathsf{com}^{(e,i)} \leftarrow \mathtt{H}_0\left(\mathsf{salt}, e, i, \mathsf{seed}^{(e,i)}\right)$.

[13] By mathematical induction, we can prove that probability to find at least one collision of random oracle $H : \{0,1\}^* \to \{0,1\}^{2\lambda}$ after n calls is at most $n(n-1)/2^{2\lambda+1}$.

5. A solution $\mathbf{s}$ to the $\texttt{PowAff2}_{u-1}$ instance $(\mathbf{f}, \mathbf{t})$ can be extracted from $(\mathsf{seed}^{(e^*,i)})_{i\in[N]}$ and $\boldsymbol{\Delta}\mathbf{s}^{(e^*)}$.

At the end of $\mathbf{Game}_5$, for each Forge, i.e., whenever $\mathcal{A}$ outputs a forgery (msg, σ), one can check if any execution $e \in [\tau]$ defines a good witness. We define by Solve the event in which there exists at least one good execution $e^* \in [\tau]$, where query to $\texttt{H}_4$ is built from σ and following the verification algorithm (see Figure 8), and the $(\boldsymbol{\Delta}\mathbf{s}^{(1)}, \ldots, \boldsymbol{\Delta}\mathbf{s}^{(\tau)})$ are the one in σ. Consequently, $\Pr_5[\mathsf{Forge} \cap \mathsf{Solve}] = \varepsilon_{\texttt{PowAff2}}$.

We finalize the proof by showing that $\Pr_5[\mathsf{Forge} \cap \overline{\mathsf{Solve}}] \le \Pr[X + Y = \tau]$, where $X = \max_{i\in[0,q_2]}\{X_i\}$ $X_i \sim \mathcal{B}(\tau, \frac{1}{q^u})$, and $Y = \max_{i\in[0,q_4]}\{Y_i\}$ with $Y_i \sim \mathcal{B}(\tau - X, \frac{1}{N})$.

In the event $\mathsf{Forge} \cap \overline{\mathsf{Solve}}$, (by the soundness part of Theorem 1) we either get a false-positive case of the $\texttt{MPC}$ protocol (see Figure 2), or $\mathcal{A}$ have cheated in exactly one party. We analyze each scenario separately.

(*False-positive case*) We denote by h_1 the output of a given query Q_2 to $\texttt{H}_2$ made by $\mathcal{A}$. After the $\texttt{MPC}$ protocol is executed in the main shares as described in Figure 4, $\mathcal{A}$ can count the number of indexes $e \in [\tau]$ for which the e-th execution yields a false-positive, we use $F_2(h_1)$ to denote that number. Since the first challenge $\boldsymbol{\varepsilon}^{(e)} = (\varepsilon_1^{(e)}, \ldots, \varepsilon_m^{(e)})$ is sampled uniformly at random independently of h_1, by Proposition 1, we have that $\Pr[e \in F_2(h_1) \mid \overline{\mathsf{Solve}}] \le \frac{1}{q^u}$ for any $e \in [\tau]$. Therefore, $X_i \sim \mathcal{B}(\tau, \frac{1}{q^u})$, where X_i denotes $\#F_2(h_1)$ in the i-th query Q_2 of $\mathcal{A}$ to $\texttt{H}_2$. Let us define the random variable $X = \max_{i\in[q_2]} X_i$.

(*Cheating case*) Let us assume $X = \tau_1 = \#F_2(h_1)$. For any $e \in [\tau] \setminus F_2(h_1)$, by the soundness part of Theorem 1, we know that $\mathcal{A}$ has to cheat in exactly one party in order to have a nonzero probability (which is $\frac{1}{N}$) that the e-th execution is accepted. Notice, the verification is accepted if and only if the e-th execution is accepted for each $e \in [\tau] \setminus F_2(h_1)$. Now, let us define the random variable $Y = \max_{i\in[q_4]} Y_i$, where Y_i is the random variable returning the number of indexes $e \in [\tau] \setminus F_2(h_1)$ for which the e-th execution is accepted in the i-th query to $\texttt{H}_4$. Hence, in the particular case $X = \tau_1$, the probability that the verification is accepted is given by $\Pr[Y = \tau - \tau_1 \mid X = \tau_1]$. Therefore, by summing over all possible values of X, we obtain that

$$\Pr_5[\mathsf{Forge} \cap \overline{\mathsf{Solve}}] \le \Pr[X + Y = \tau].$$

The proof is concluded by the fact that.

$$\begin{aligned}
\Pr[\mathsf{Forge}] = \Pr_1[\mathsf{Forge}] &\le \sum_{j=1}^{4} \left|\Pr_j[\mathsf{Forge}] - \Pr_{j+1}[\mathsf{Forge}]\right| + \Pr_5[\mathsf{Forge}] \\
&= \sum_{j=1}^{4} \left|\Pr_j[\mathsf{Forge}] - \Pr_{j+1}[\mathsf{Forge}]\right| \\
&\quad + \Pr_5[\mathsf{Forge} \cap \mathsf{Solve}] + \Pr_5[\mathsf{Forge} \cap \overline{\mathsf{Solve}}].
\end{aligned}$$

6 Implementation

6.1 Canonical Representation Optimization

As seen in Lemma 1, an equivalent system where, for the first n equations, one of the affine forms is only composed of one variable. Without loss of generality, we can choose to have this variable in $A_{k,0}$. In other words, we can choose for the algorithm a system $f_1, \ldots, f_m$ as

$$f_k(x_1, \ldots, x_n) = (x_k + a_k) + A_{k,1}(x_1, \ldots, x_n) \cdot A_{k,2}(x_1, \ldots, x_n),$$

for $k \leqslant n$, and

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

for $n < k \leqslant m$, where $A_{k,j}$ are affine forms.

The effect is that the evaluation of the polynomial will be much faster as only 2 affine form evaluations have to be performed instead of 3 for most of the equations. In the implementation, we chose to simplify $A_{k,0}$ to save some code, as $A_{k,1}$ and $A_{k,2}$ can be computed in the same way in a loop.

6.2 Hypercube Optimization

The algorithms described in Figures 7 and 8 use the hypercube variant. The simulation of the MPC protocol does not need to compute all the values as in Figure 4. We first compute α_k using directly the opened values $\mathbf{s}$ and $\mathbf{a}$. Then, we need to compute $[\![\alpha_k]\!]_{(\ell,j)}$ only for $j = 1$. The value for $j = 2$ can be derived from α. Similarly, we can do the same for $[\![v_k]\!]_{(\ell,j)}$. This can also be applied to the verification. All in all, we usually require to keep only $\log_2(N)$ shares.

6.3 Vectorization

The main data structure in the algorithm is a vector of value in $\mathbb{F}_q$. We have:

- The secret value, which is a vector of n elements in $\mathbb{F}_q$.
- The public key, which is a vector of m elements in $\mathbb{F}_q$.
- Intermediate values, which are vectors of m elements in $\mathbb{F}_q$.

For each of these vectors, we need to compute operations component-wise. We can then pack all elements in the largest possible integer handled by the CPU. Typically, this could be a 64-bit word that can contain 8 elements in $\mathbb{F}_{2^8}$ for instance.

When vectorized instructions are available (`SSE`, `AVX`, ...), even larger integer types can be used. For instance, with `AVX2` a 256-bit integer can be used to pack a vector of $\mathbb{F}_q$ elements. In characteristic 2, the component-wise addition of a vector of elements can be done in one instruction using the `VPXOR` instruction.

6.4 Performances and Memory Consumption

In this section, we show the performance and memory consumption of our instances. Our implementation is optimized to use `AVX2` vectorized instructions on a little-endian 64-bit CPU.

The code is compiled with `GCC` version 12.2.0 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 (Tiger Lake). Even if frequency modification should not affect this metric, we deactivated Intel's TurboBoost feature anyway. The number of cycles is taken as the median over 1000 executions.

Name	Memory (bytes)			Performance (cycles)		
	`keygen`	`sign`	`verify`	`keygen`	`sign`	`verify`
`biscuit128s`	512	1 654 288	122 480	88 484	69 418 295	68 984 920
`biscuit128f`	512	329 904	25 712	88 477	13 711 517	13 007 550
`biscuit192s`	608	3 438 832	194 544	251 806	191 442 370	190 138 451
`biscuit192f`	608	708 944	49 392	252 106	38 677 691	37 087 201
`biscuit256s`	800	7 414 000	335 312	504 021	635 749 877	632 271 590
`biscuit256f`	800	1 537 904	98 768	504 983	128 098 892	124 921 246

Table 3: Time performance and memory consumption of Biscuit on `avx2` impl.

In Table 3, we give the figures for the implementation strictly following the description in the `NIST` submission but with the new parameters proposed in Table 2.

In Table 4, we include the canonical representation optimization as described in Section 6.1. This improves the performances by 18 to 28 percent.

Name	Memory (bytes)			Performance (cycles)		
	`keygen`	`sign`	`verify`	`keygen`	`sign`	`verify`
`biscuit128s`	512	1 651 088	122 480	61 755	60 785 166	59 198 143
`biscuit128f`	512	326 704	25 712	61 757	11 507 884	10 695 367
`biscuit192s`	608	3 430 288	194 544	172 825	151 956 515	152 714 889
`biscuit192f`	608	700 400	49 392	172 446	30 476 727	29 191 279
`biscuit256s`	800	7 393 680	335 312	343 001	472 774 277	468 258 145
`biscuit256f`	800	1 517 584	98 768	341 156	93 221 776	89 507 805

Table 4: Time performance and memory consumption of Biscuit on `avx2` impl. using canonical optimization.

Finally, in Table 5, in addition to the previous optimization, we integrated the hypercube variant. With this variant, the memory consumption is greatly improved especially for large values of N. This is because we have to keep track of only $\log_2(N)$ shares instead of N. The performances are improved by 50 to 83 percent for the small variant, and by 41 to 69 percent for the fast variant. The code is available in [2].

Name	Memory (bytes)			Performance (cycles)		
	`keygen`	`sign`	`verify`	`keygen`	`sign`	`verify`
`biscuit128s`	576	814 256	40 144	61 697	27 930 795	28 323 314
`biscuit128f`	576	201 744	14 096	61 682	6 581 004	6 166 694
`biscuit192s`	704	1 686 416	67 376	173 044	49 890 911	49 914 321
`biscuit192f`	704	433 008	28 272	172 667	13 594 397	12 916 931
`biscuit256s`	960	3 556 624	117 424	341 657	77 620 375	77 447 430
`biscuit256f`	960	928 368	57 648	340 649	28 219 223	27 341 671

Table 5: Time performance and memory consumption of Biscuit on `avx2` impl. using canonical and hypercube optimization.

7 Acknowledgements

The authors would like to thank Daniel Escudero for meaningful insights on an early version of this paper and the referees of ACNS24 that helped to improve the paper.

The third author would like to thank Charles Bouillaguet and Julia Sauvage for discussions on the hardness of `PowAff2`, Google which partially supported this work thanks to a gift for supporting post-quantum research, and the European Union's Horizon Europe research and innovation program that partially supported this research under the project "Quantum Secure Networks Partnership" (QSNP, grant agreement No 101114043).

References

1. `NIST`. Call for Additional Digital Signature Schemes for the Post-Quantum Cryptography Standardization Process . https://csrc.nist.gov/csrc/media/Projects/pqc-dig-sig/documents/call-for-proposals-dig-sig-sept-2022.pdf.
2. Biscuit github repository, 2023. https://github.com/BiscuitTeam/Biscuit.
3. Najwa Aaraj, Slim Bettaieb, Loïc Bidoux, Alessandro Budroni, Victor Dyseryn, Andre Esser, Philippe Gaborit, Mukul Kulkarni, Victor Mateu, Marco Palumbi, Lucas Perin, and Jean-Pierre Tillich. PERK specification. 2023. available at https://pqc-perk.org/assets/downloads/PERK_specifications.pdf.
4. Gora Adj, Stefano Barbero, Emanuele Bellini, Andre Esser, Luis Rivera-Zamarripa, Carlo Sanna, Javier Verbel, and Floyd Zweydinger. MiRitH specification. 2023. available at https://pqc-mirith.org/assets/downloads/mirith_specifications_v1.0.0.pdf.
5. Carlos Aguilar Melchor, Nicolas Gama, James Howe, Andreas Hülsing, David Joseph, and Dongze Yue. The return of the SDitH. pages 564–596, 2023.
6. Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, John Kelsey, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, and Angela Robinson abd Daniel Smith-Tone. Status Report on the Second Round

of the NIST Post-Quantum Cryptography Standardization Process. Technical Report NISTIR 8309, NIST, 2022. https://nvlpubs.nist.gov/nistpubs/ir/2020/NIST.IR.8309.pdf.

7. 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.
8. Nicolas Aragon, Magali Bardet, Loïc Bidoux, Jesús-Javier Chi-Domínguez, Victor Dyseryn, Thibauld Feneuil, Philippe Gaborit, Antoine Joux, Matthieu Rivain, Jean-Pierre Tillich, and Adrien Vinçotte. RYDE specification. 2023. available at https://pqc-ryde.org/assets/downloads/RYDE_Specifications.pdf.
9. Nicolas Aragon, Magali Bardet, Loïc Bidoux, Jesús-Javier Chi-Domínguez, Victor Dyseryn, Thibauld Feneuil, Philippe Gaborit, Romaric Neveu, Matthieu Rivain, and Jean-Pierre Tillich. MIRA specification. 2023. available at https://pqc-mira.org/assets/downloads/mira_spec.pdf.
10. Nicolas Aragon, Loïc Bidoux, Jesús-Javier Chi-Domínguez, Thibauld Feneuil, Philippe Gaborit, Romaric Neveu, and Matthieu Rivain. Mira: a digital signature scheme based on the minrank problem and the mpc-in-the-head paradigm, 2023.
11. 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.
12. 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.
13. Carsten Baum, Lennart Braun, Michael Klooß, Christian Majenz, Shibam Mukherjee, Emmanuela Orsini, Sebastian Ramacher, Christian Rechberger, Lawrence Roy, and Peter Scholl. FAEST specification. 2023. available at https://faest.info/faest-spec-v1.1.pdf.
14. 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.
15. Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1991, Proceedings*, volume 576 of *Lecture Notes in Computer Science*, pages 420–432. Springer, 1991.
16. Emanuele Bellini, Rusydi H. Makarim, Carlo Sanna, and Javier A. Verbel. An estimator for the hardness of the MQ problem. pages 323–347, 2022.
17. 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.
18. 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.

19. Luk Bettale, Ludovic Perret, Delaram Kahrobaei, and Javier Verbel. Biscuit: Shorter MPC-based Signature from PoSSo, June 2023. Specification of NIST post-quantum signature.
20. Charles Bouillaguet and Julia Sauvage. Preliminary Cryptanalysis of the Biscuit Signature Scheme. Preprint, January 2024.
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. pages 1825–1842, 2017.
22. 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.
23. 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.
24. Thibauld Feneuil and Matthieu Rivain. MQOM specification. 2023. available at https://mqom.org/docs/mqom-v1.0.pdf.
25. Thibauld Feneuil and Matthieu Rivain. Threshold computation in the head: Improved framework for post-quantum signatures and zero-knowledge arguments. Cryptology ePrint Archive, Paper 2023/1573, 2023. https://eprint.iacr.org/2023/1573.
26. 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.
27. 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.
28. 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.
29. 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.
30. 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.
31. 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.
32. Seongkwang Kim, Jihoon Cho, Mingyu Cho, Jincheol Ha, Jihoon Kwon, Byeonghak Lee, Joohee Lee, Jooyoung Lee, Sangyub Lee, Dukjae Moon, Mincheol Son, and Hyojin Yoon. AIMER specification. 2023. available at https://aimer-signature.org/docs/AIMer-NIST-Document.pdf.
33. Carlos Aguilar Melchor, Thibauld Feneuil, Nicolas Gama, Shay Gueron, James Howe, David Joseph, Antoine Joux, Edoardo Persichetti, Tovohery H., Randrianarisoa, Matthieu Rivain, and Dongze Yue. SDITH specification. 2023. available at https://sdith.org/docs/sdith-v1.0.pdf.

34. Carlos Aguilar Melchor, Nicolas Gama, James Howe, Andreas Hülsing, David Joseph, and Dongze Yue. The return of the sdith. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology - EUROCRYPT 2023 - 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Lyon, France, April 23-27, 2023, Proceedings, Part V*, volume 14008 of *Lecture Notes in Computer Science*, pages 564–596. Springer, 2023.
35. Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *J. ACM*, 56(6):34:1–34:40, 2009.
36. 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.