BBQ: Using AES in Picnic Signatures

Cyprien Delpech de Saint Guilhem[1,2][0000−0002−0147−2566], Lauren De Meyer[1][0000−0002−3519−2722], Emmanuela Orsini[1][0000−0002−1917−1833], and Nigel P. Smart[1,2][0000−0003−3567−3304]

[1] imec-COSIC, KU Leuven, Belgium.
[2] Dept Computer Science, University of Bristol, United Kingdom.
cyprien.delpechdesaintguilhem@kuleuven.be, lauren.demeyer@kuleuven.be, emmanuela.orsini@kuleuven.be, nigel.smart@kuleuven.be.

Abstract. This works studies the use of the AES block-cipher for Picnic-style signatures, which work in the multiparty-computation-in-the-head model. It applies advancements to arithmetic circuits for the computation of the AES S-box over multiparty computation in the preprocessing model to obtain an improvement of signature sizes of 40% on average compared to using binary circuits for AES-128, AES-192 and AES-256 in combination with previous techniques. This work also discusses other methods for the computation of the S-box and provides insights into the reaches and limits of the multiparty-computation-in-the-head paradigm.

1 Introduction

With the possible advent of a quantum computer, cryptographers have turned their attention to building "post-quantum" variants of standard cryptographic functionalities such as public key encryption and digital signatures. Among the various paradigms for producing post-quantum signatures, one of the most promising is that of MPC-in-the-head (MPCitH), which was introduced in [IKOS07]. In this paradigm, the security of signatures is based on the security of a semi-honest multiparty computation (MPC) protocol against a dishonest majority, and of a given PRF family $F_k(\cdot)$, such as a block-cipher. Given the nature of the MPC protocol, the concrete security of the resulting signature scheme relies on the underlying PRF.

The secret key for such signature schemes is the key k used to select the PRF $F_k(\cdot)$, with the public key being an input-output pair of the PRF, i.e. values (x, y) such that $y = F_k(x)$. The signature itself is then a non-interactive zero-knowledge proof of knowledge (NIZKPoK) of the secret key k produced using the MPCitH paradigm. To do so, the signer first computes a commitment to an execution of an MPC evaluation of the PRF on input x, which constitutes the first component of the signature; then they hash the commitment to obtain the NIZKPoK challenge, according to the Fiat-Shamir heuristic, and include the message to be signed in the hash of the commitment; finally they respond to this challenge by outputting the views of a subset of parties as the final component of the signature. That this process reveals nothing about the secret key k follows from the security of the MPC protocol, and the security of the PRF. That it also convinces a verifier that the signer knows k follows from [IKOS07].

While originally considered as a theoretical construction, MPCitH has been used to successfully create practical NIZKPoKs. ZKBoo [GMO16], ZKBoo++ [CDG+17] and Ligero [AHIV17] are examples of such schemes. While ZKBoo and Ligero provide schemes for proofs of knowledge for SHA-256 pre-images, the ZKBoo++ paper makes use of the LowMC cipher [ARS+15], which was designed as a cipher optimized for evaluation in FHE and MPC applications. This construction (ZKBoo++ and LowMC) is the main idea behind the submission to the NIST Post-Quantum Cryptography project [Nat16] titled Picnic [CDG+19b].

The first applications of MPCitH drew inspiration from traditional MPC protocols. In recent years many practical improvements to these have been made in the so-called preprocessing model (such as VIFF, BDOZ and SPDZ [BDOZ11,DGKN09,DPSZ12]). Following the paradigm of MPC with preprocessing, an adaptation of the MPCitH paradigm to use the preprocessing model was presented recently in [KKW18]. This technique allows for both smaller and more compact signatures together with faster verification and signing times. It is this later technique which, combined with the LowMC cipher, is used in the Round 2 updated Picnic submission [CDG+19a].

The choice of the LowMC block-cipher as the PRF was made as it has a particularly simple circuit design. However, using LowMC comes with a risk. It is a less studied cipher, compared to standardized ones such as AES, and thus it is possible that future cryptanalytic attacks be developed which would then render the above digital signature schemes insecure. However, using a standard block cipher such as AES comes with a penalty. For example, the standard circuit for AES is far more complex than that for LowMC in terms of non-linear AND gates, and this results in larger signature sizes and computation times.

However, the above intuitive comparison assumes that a binary circuit for AES would be used within the MPCitH. There are however other ways to evaluate AES over MPC, see for example [DK10,DKL+12,KOR+17]. Many of these approaches use arithmetic circuits, and thus exploit the algebraic structure of the AES block cipher.

Contributions. In this work, we show how one can use the MPCitH with preprocessing paradigm of [KKW18] for arithmetic circuits over $\mathbb{F}_{2^8}$ to define a signature scheme whose security rests on AES as opposed to LowMC. We estimate that our approach produces signatures that are on average 40% smaller than those that would be obtained using a binary circuit for AES. Our estimates are $2.5, 3.1$ and 2.8 times larger than the Picnic signatures using on LowMC for the L1, L3 and L5 security levels of the NIST project respectively. Thus our technique enables one to achieve security based on a block-cipher component which is better understood, with a better-than-expected penalty in terms of signature size.

To achieve this improvement compared to binary circuits, we make use of MPC techniques to compute the AES S-box. The problem is in computing the only non-linear component in AES is the S-Box inversion, namely the computation of $s \mapsto s^{254}$ in the finite field $\mathbb{F}_{2^8}$, which is well known to be equal to the function $s \mapsto 1/s$ when $s \neq 0$. This inversion function is easy to evaluate in MPC but different methods to capture the $s = 0$ case require more or less communication between parties, which directly translates to signature size.

Our main solution to the problem of zero S-box inputs is for the signer to select his private key k and the x component of the public key so that the computation of $y = F_k(x)$ (when F_k is the AES function) involves no zero-inputs to any S-Box. While this may seem restrictive, it in fact only reduces the key space by approximately 1.1, 2.4 and 2.9 bits for the AES-128, AES-192 and AES-256 circuits respectively. Thus, making this reduction in the key-space bypasses the problem one faces in producing an inverse of zero. This provides our most efficient solution and we obtain signature size estimates of 31.6 kB, 86.9 kB and 133.7 kB for circuits achieving L1, L3 and L5 security levels respectively.

Interestingly, we can observe that scaling up the key size of AES (and using AES-192 or AES-256) does not automatically scale up the security parameter when used in the MPCitH paradigm. The problem lies in the fact that the block-length of the circuit stays the same (128 bits in all AES versions), resulting in spurious keys if only single block input/output pairs are used in the

public key. To solve this and achieve the L3 and L5 security levels, we come up with a way to construct simple circuits with appropriate block-lengths out of the AES cipher. We also consider the use of the original Rijndael block-cipher on which AES is based and which has parametrisable block-length.

Finally, we also discuss other solutions to address the case of $s = 0$ during the inversion computation. These would enable the use of any k and x for the signature key and therefore not reduce the bit-security of the cipher, but they would require significantly more communication in the MPC protocol and thus produce longer signatures. However, these alternative solutions provide insights on the added capabilities of the MPCitH paradigm could direct the design of more appropriate MPC protocols for this purpose.

2 Preliminaries

We let κ denote the computational security parameter. We also let H denote a collision-resistant hash function with co-domain $\{0,1\}^{2\kappa}$ and let com be a non-interactive commitment scheme where x is committed to by sampling a random $r \in \{0,1\}^{\kappa}$ and computing $\gamma = \mathsf{com}(x, r)$; decommitment is done by opening x and r. (The Picnic scheme [CDG+19a] uses a hash function to instantiate com.) We use bold lower case letters for tuples, overset small arrow to indicate elements of a vector space, like $\vec{v}$, and bold upper case letters for matrices. We also write $[n] = \{1, \dots n\}$.

2.1 Efficient NIZKPoK in the MPC-in-the Head Paradigm

In 2007, Ishai et al. [IKOS07] showed how to use any MPC protocol to construct a zero-knowledge (ZK) proof for an arbitrary NP relation $\mathcal{R}$. The high level idea is the following: zero knowledge can be seen as a special function evaluation, and hence as a two-party computation between a prover $\mathcal{P}$ and a verifier $\mathcal{V}$, with common input the statement x, and $\mathcal{P}$'s private input w, which is a witness to the assertion that x belongs to a given NP language $\mathcal{L}$. The function they want to compute is then $f_x(w) = \mathcal{R}(x, w)$, which checks if w is a valid witness or not. The verifier $\mathcal{V}$ will accept the proof if $f_x(w) = \mathcal{R}(x, w) = 1$.

In the MPCitH paradigm the prover $\mathcal{P}$ simulates an n-party MPC protocol Π in "its head": $\mathcal{P}$ first samples n random values $w^{(1)}, \dots, w^{(n)}$, subject to the condition $\sum_{i\in[n]} w^{(i)} = w$, as private inputs to the parties, and then emulates the evaluation of $f(w^{(1)}, \dots, w^{(n)}) = \mathcal{R}(x, w^{(1)} + \cdots + w^{(n)})$ by choosing uniformly random coins $r^{(i)}$ for each party P_i, $i \in [n]$. Note that, once the inputs and random coins are fixed, for each round j of communication of the protocol Π and for each party P_i, the messages sent by P_i at round j are deterministically specified as a function of the internal state of P_i, i.e. P_i's private inputs $w^{(i)}$ and randomness $r^{(i)}$, and the messages $\mathsf{msg}^{(i)}$ that P_i received in previous rounds. The set with the state and all messages received by party P_i during the execution of the protocol constitutes the view of P_i, denoted as View_{P_i}.

After the evaluation, the prover $\mathcal{P}$ commits to the views of each party and sends them to $\mathcal{V}$. At this point, the verifier "corrupts" a random subset of parties, challenging the prover to open their committed views and finally verifies that the computation was done correctly from the perspective of these corrupt parties, by checking that the opened views are all consistent with each other. To obtain the desired soundness it is often necessary to iterate the above procedure many times.

Using the MPCitH paradigm, some recent works [AHIV17,CDG+17,GMO16] have constructed efficient NIZKPoK for Boolean circuits and signature schemes [CDG+17,KKW18]. In particular,

the work of Katz et al. [KKW18] constructs an honest-verifier zero-knowledge (HVZK) proof of knowledge (HVZKPoK) by instantiating the MPCitH paradigm using an MPC protocol designed in the preprocessing model. We now describe their protocol adapted for arithmetic circuits.

Arithmetic MPCitH in the Preprocessing Model. We consider some finite field $\mathbb{F}$. The HVZK protocol we introduce here provides a proof of knowledge of a witness $\boldsymbol{w} \in \mathbb{F}^\iota$ such that $C(\boldsymbol{w}) = \boldsymbol{y} \in \mathbb{F}^o$, for a given circuit $C : \mathbb{F}^\iota \to \mathbb{F}^o$ and output $\boldsymbol{y}$.

As the underlying protocol Π_C is designed in the preprocessing model (and therefore has a preprocessing phase followed by an online phase), the HVZK proof also happens in two stages. In the first stage, $\mathcal{P}$ commits to a number of preprocessing executions (which consist only of input-independent randomness). Then $\mathcal{V}$ requests that some of these are opened, and checks that they are consistent preprocessing computations. In the second stage, the prover uses the unopened preprocessing material to execute parallel and independent executions of the online phase of the MPC protocol Π_C. It then commits to these executions and, for each of them, is challenged by the verifier to open a random selection of $n-1$ parties' views. By receiving these views, $\mathcal{V}$ is able to perform checks to ensure that the prover did not falsify the executions of Π_C.

Since we will make use of an arithmetic circuit C over a field $\mathbb{F}$, C is composed of addition and multiplication gates operating on values in $\mathbb{F}$. This is a generalization of [KKW18] as that work considers circuits operating on only bits with XOR and AND gates. The online evaluation of C is done by a SPDZ-like [DPSZ12] protocol, simplified as we only require security against semi-honest adversaries (notably, we use broadcast as the only communication channel).

This protocol is executed by n parties. For every value $x \in \mathbb{F}$ in the arithmetic circuit C, the protocol uses an additive secret sharing $\langle x \rangle$ which denotes the sharing $x^{(1)}, \ldots, x^{(n)}$, such that $\sum_{i=1}^{n} x^{(i)} = x$ and every party P_i holds $x^{(i)}$.

To reconstruct, or "open", a shared value $\langle x \rangle$, each party P_i broadcasts its share $x^{(i)}$ and each party P_j can then reconstruct $x = \sum_{i=1}^{n} x^{(i)}$. With such a secret sharing, the following operations can be performed locally, i.e. without communication between parties:

- *Addition with public constant:* To compute $\langle z \rangle \leftarrow \langle x + a \rangle$ given $\langle x \rangle$ and a public value a, party P_1 sets his share to be $z^{(1)} \leftarrow x^{(1)} + a$ and every other party P_i $(i \neq 1)$ sets $z^{(i)} \coloneqq x^{(i)}$.
- *Addition of two shared values:* To compute $\langle z \rangle \leftarrow \langle x + y \rangle$ given $\langle x \rangle$ and $\langle y \rangle$, every party P_i sets his share to be $z^{(i)} \leftarrow x^{(i)} + y^{(i)}$.
- *Multiplication by public constant:* To compute $\langle z \rangle \leftarrow \langle a \cdot x \rangle$ given $\langle x \rangle$ and a, every party P_i sets his share to be $z^{(i)} \leftarrow a \cdot x^{(i)}$.

However, computing the multiplication of two shared values, i.e. $\langle z \rangle \leftarrow \langle x \cdot y \rangle$ given $\langle x \rangle$ and $\langle y \rangle$, cannot be done locally; it requires both preprocessing material and communication during the online phase. Namely, given a precomputed triple $(\langle a \rangle, \langle b \rangle, \langle c \rangle)$, such that $c = a \cdot b$, the parties can compute $\langle z \rangle$ as follows:

1. Locally compute $\langle \alpha \rangle \coloneqq \langle x - a \rangle$ and $\langle \beta \rangle \coloneqq \langle y - b \rangle$.
2. Open α and β.
3. Locally compute $\langle z \rangle = \langle c \rangle - \alpha \cdot \langle b \rangle - \beta \cdot \langle a \rangle + \alpha \cdot \beta$.

This technique is due to [Bea92] and is easily checked to be correct:

$$\begin{aligned} z &= c - \alpha \cdot b - \beta \cdot a + \alpha \cdot \beta \\ &= a \cdot b - (x - a) \cdot b - (y - b) \cdot a + (x - a) \cdot (y - b) \\ &= x \cdot y. \end{aligned}$$

Before we describe the HVZK protocol of [KKW18], we present the execution of both phases of Π_C that the prover will have to simulate; namely the preprocessing and online phases.

1) Preprocessing phase. The only preprocessing required is the generation of random multiplication triples $\{(\langle a_m \rangle, \langle b_m \rangle, \langle c_m \rangle)\}_{m \in [\mathsf{mult}]}$, where mult is the number of multiplication gates in C. As noted in [KKW18] for its preprocessing computation, for each triple, the i-th share of $\langle a \rangle$ and $\langle b \rangle$ is uniform and therefore can be generated by party P_i applying a pseudorandom generator (PRG) to a short random seed $\mathsf{seed}^{(i)}$. Each share of $\langle c \rangle$ can also be generated in that way, but then the actual generated value $\tilde{c} = \sum_{i=1}^{n} c^{(i)}$ would not equal $a \cdot b$ with very high probability. Instead, for each triple $(\langle a_m \rangle, \langle b_m \rangle, \langle c_m \rangle)$, party P_n is given a "correction value" $\Delta_m = a \cdot b - \sum_{i=1}^{n-1} c^{(i)}$ and directly sets $c^{(n)} \leftarrow \Delta_m$.

In summary, the outcome of the preprocessing is that every party P_i is given a κ-bit seed $\mathsf{seed}^{(i)} \in \{0,1\}^\kappa$ and P_n is also given mult values $\{\Delta_m\}$ denoted by $\mathsf{aux}^{(n)}$. This information is called the *state* of party P_i and is denoted by $\mathsf{state}^{(i)}$.

2) Online phase. The online computation can itself be divided into three components: *input distribution*, *computation* and *output reconstruction*.

Unlike for real MPC, the input $\boldsymbol{w} = (w_1, \ldots, w_\iota)$ is global and must be distributed by the prover, outside of the view of any $n-1$ parties. To do so, each party P_i uses $\mathsf{seed}^{(i)}$ to generate a random share $w_j^{(i)}$, for $j \in [\iota]$. Identically to the generation of multiplication triple, a correction value Λ_j must then be computed so that $w_j = \Lambda_j + \sum_{i=1}^{n} w_j^{(i)}$. The values $(w_j^{(1)}, \ldots, w_j^{(n)}, \Lambda_j)$ now constitute an n-out-of-$(n+1)$ sharing of w_j and therefore it is safe for $\mathcal{P}$ to communicate the Λ_j values to $\mathcal{V}$ in addition to $n-1$ of the seeds $\mathsf{seed}^{(i)}$. Before the computation begins, we let P_n be the one to set $w_j^{(n)} \leftarrow w_j^{(n)} + \Lambda_j$, for $j \in [\iota]$.

Next, the parties compute the intermediary values in the computation of $C(\boldsymbol{w})$. Whenever multiplication gate m is encountered, each party P_i recalls their share of the next unused triple $a_m^{(i)}, b_m^{(i)}, c_m^{(i)}$ from the preprocessing phase. This yields the triple $(\langle a_m \rangle, \langle b_m \rangle, \langle c_m \rangle)$ which can then be used to compute multiplication gate m. Whenever P_i needs to sample a random value, it does so using $\mathsf{seed}^{(i)}$.

Once the parties have computed the shared output values $\langle y_1 \rangle, \ldots, \langle y_o \rangle$, they jointly reconstruct $\boldsymbol{y} = C(\boldsymbol{w})$. To do so, they open each value y_j, for $j \in [o]$, by having each party broadcast its share $y_j^{(i)}$

As noted above during the description of the MPCitH paradigm, this online phase is entirely deterministic. In particular, the next broadcast message of party P_i at a given time in Π_C depends only on $\mathsf{state}^{(i)}$ and the messages received by P_i so far. Denoting by A the set of "corrupt" parties, this implies that when the verifier $\mathcal{V}$ checks the views of these $n-1$ parties in the execution of Π_C, they only need to be given the $n-1$ states of parties in A together with the messages broadcast by the unopened party during each multiplication gate; they can then recompute the internal values of $C(\boldsymbol{w})$ since they can infer the broadcast messages sent by the opened parties. This means that instead of sending View_{P_i}, for each $i \in A$, $\mathcal{P}$ only needs to send $\mathsf{View}_A = \{\mathsf{state}^{(i)}\}_{i \in A} \cup \{\mathsf{msg}^{(j)}\}_{j \notin A}$, where $\{\mathsf{msg}^{(j)}\}_{j \notin A}$ are the messages sent by the "honest" parties.

PARAMETERS: Let $H : \{0,1\}^* \to \{0,1\}^{2\kappa}$ and G be collision-resistant hash functions and com be a commitment scheme. Note G is only used in the non-interactive variant of the protocol.
Other parameters are T and τ, that indicate the total number of preprocessing executions emulated by $\mathcal{P}$ and the number of online executions, respectively.

INPUTS: Both parties hold a description of C over $\mathbb{F}$, the value $\boldsymbol{y} \in \mathbb{F}^o$. The prover $\mathcal{P}$ also holds $\boldsymbol{w} \in \mathbb{F}^\iota$ such that $C(\boldsymbol{w}) = \boldsymbol{y}$.

Round 1: *Protocol execution emulation.*

1. For each $t \in [T]$, $\mathcal{P}$ emulates the preprocessing phase as follows:
   (a) Sample a uniform master seed $\mathsf{seed}_t \in \{0,1\}^\kappa$ and use it to generate $\mathsf{seed}_t^{(1)}, \ldots, \mathsf{seed}_t^{(n)} \in \{0,1\}^\kappa$ and $r_t^{(1)}, \ldots, r_t^{(n)} \in \{0,1\}^\kappa$ used in the commitments.
   (b) For each multiplication gate $m \in [\mathsf{mult}]$:
      i. Use $\mathsf{seed}_j^{(i)}$ to sample $a_{t,m}^{(i)}, b_{t,m}^{(i)}$ and also $c_{t,m}^{(i)}$ for $i = 1, \ldots, n-1$.
      ii. Compute $a_{t,m} = \sum_{i=1}^{n} a_{t,m}^{(i)}$ and $b_{t,m}$ similarly.
      iii. Compute the offset $\Delta_{t,m} = a_{t,m} \cdot b_{t,m} - \sum_{i=1}^{n-1} c_{t,m}^{(i)}$.
   (c) Set $\mathsf{aux}_t^{(n)} = (\Delta_{t,m})_{m \in [\mathsf{mult}]}$.
   (d) Set $\mathsf{state}_t^{(i)} = \mathsf{seed}_t^{(i)}$ for $i \in [n-1]$ and set $\mathsf{state}_t^{(n)} = (\mathsf{seed}_t^{(n)}, \mathsf{aux}_t^{(n)})$.
   (e) For each $i \in [n]$, compute $\gamma_{\mathsf{s},t}^{(i)} = \mathsf{com}(\mathsf{state}_t^{(i)}, r_t^{(i)})$.
   (f) Compute $h_{\mathsf{s},t} = H(\gamma_{\mathsf{s},t}^{(1)}, \ldots, \gamma_{\mathsf{s},t}^{(n)})$.
2. For each $t \in [T]$, given $\boldsymbol{w} = (w_1, \ldots, w_\iota)$, $\mathcal{P}$ emulates the online phase as follows:
   (a) Generate the input offsets for each $j \in [\iota]$:
      1. Use $\mathsf{seed}_t^{(i)}$ to sample $w_{t,j}^{(i)}$ for $i \in [n]$.
      2. Compute $\tilde{w}_{t,j} = \sum_{i=1}^{n} w_{t,j}^{(i)}$.
      3. Compute $\Lambda_{t,j} = w_j - \tilde{w}_{t,j}$.
   (b) Compute C by proceeding through the gates in topological order. For each party P_i, record each broadcast message in $\mathsf{msg}_t^{(i)}$.
   (c) Compute $h_{\mathsf{m},t} = H(\{\Lambda_{t,j}\}_{j \in [\iota]}, \mathsf{msg}_t^{(1)}, \ldots, \mathsf{msg}_t^{(n)})$.
3. Compute $h_{\mathsf{s}} = H(h_{\mathsf{s},1}, \ldots, h_{\mathsf{s},T}), h_{\mathsf{m}} = H(h_{\mathsf{m},1}, \ldots, h_{\mathsf{m},T})$ and send $h^* = H(h_{\mathsf{s}}, h_{\mathsf{m}})$ to $\mathcal{V}$.

Fig. 1: 3-round HVZK proof (part 1).

The HVZKPoK Protocol. In Figures 1 and 2, we give a slightly modified version of the 3-round protocol presented by Katz et al. [KKW18] to compute their proof of knowledge, so that it can use our Π_C protocol for an arithmetic circuit C over $\mathbb{F}$. We refer the reader to the original paper [KKW18] for a more in-depth explanation of the protocol. We adopt their optimization of performing a more general cut-and-choose by having the prover run T independent preprocessing phases and the verifier choosing τ of them to be used for independent online phases. For each of these online phases, $\mathcal{V}$ challenges $\mathcal{P}$ on $n-1$ parties which it then checks, together with the $T - \tau$ preprocessing phases that were not used.

From the 3-round protocol, it is possible to obtain a non-interactive ZKPoK by applying the Fiat-Shamir [FS87] transform, as indicated in the description of **Round 2** in Figure 2. The system we obtain in this way consists of two stages, Prove and $\mathsf{VerifyProof}$.

- $\mathsf{Prove}(\boldsymbol{y}, \boldsymbol{w})$ takes as input $(\boldsymbol{y}, \boldsymbol{w})$ and consists of **Round 1, 2** and **3** of Fig. 1 and 2; i.e. $\mathcal{P}$ computes the first round message and then *locally* computes the challenge by hashing this message using a collision-resistant hash function G. The resulting proof, σ, consists of the concatenation of the first-round message and the response to the challenge.
- $\mathsf{VerifyProof}(\boldsymbol{y}, \sigma)$ does the **Verification** step (Fig. 2) and returns $b \in \{0,1\}$.

Round 2: *Challenge.*
$\mathcal{V}$ challenges $\mathcal{P}$ on the executions indexed by a set $\mathcal{T} \subset [T]$, with $|\mathcal{T}| = \tau$. For each of these, it chooses a party that remains honest, i.e. it chooses a vector $(i_\mathfrak{t})_{\mathfrak{t}\in\mathcal{T}} \in [n]^\tau$. It then sends $(\mathcal{T}, (i_\mathfrak{t})_{\mathfrak{t}\in\mathcal{T}})$ to $\mathcal{P}$.
In the NIZKPoK variant, $\mathcal{P}$ locally computes $(\mathcal{T}, (i_\mathfrak{t})_{\mathfrak{t}\in\mathcal{T}}) = G(h^*)$.

Round 3: *Opening.*
$\mathcal{P}$ sends $(\mathsf{seed}_\mathfrak{t}, h_{\mathsf{m},\mathfrak{t}})_{\mathfrak{t}\in\overline{\mathcal{T}}}$ and $((\mathsf{state}_\mathfrak{t}^{(i)}, r_\mathfrak{t}^{(i)})_{i\in[n]\setminus\{i_\mathfrak{t}\}}, \gamma_{\mathsf{s},\mathfrak{t}}^{(i_\mathfrak{t})}, \{\Lambda_{\mathfrak{t},j}\}_{j\in[\iota]}, \mathsf{msg}_\mathfrak{t}^{(i_\mathfrak{t})})_{\mathfrak{t}\in\mathcal{T}}$, to $\mathcal{V}$.

Verification:

1. For each $\mathfrak{t} \in \mathcal{T}$, use $(\mathsf{state}_\mathfrak{t}^{(i)}, r_\mathfrak{t}^{(i)})_{i\in[n]\setminus\{i_\mathfrak{t}\}}$ to reconstruct $\gamma_{\mathsf{s},\mathfrak{t}}^{(i)}$ for $i \in [n] \setminus \{i_\mathfrak{t}\}$. Then compute $h'_{\mathsf{s},\mathfrak{t}} = H(\gamma_{\mathsf{s},\mathfrak{t}}^{(1)}, \ldots, \gamma_{\mathsf{s},\mathfrak{t}}^{(n)})$ using $\gamma_{\mathsf{s},\mathfrak{t}}^{(i_\mathfrak{t})}$ sent by $\mathcal{P}$.
2. For $\mathfrak{t} \in \overline{\mathcal{T}}$, use $\mathsf{seed}_\mathfrak{t}$ to compute $h'_{\mathsf{s},\mathfrak{t}}$ as an honest $\mathcal{P}$ would.
3. Then compute $h'_\mathsf{s} = H(h'_{\mathsf{s},1}, \ldots, h'_{\mathsf{s},T})$.
4. For each $\mathfrak{t} \in \mathcal{T}$, use $\{\mathsf{state}_\mathfrak{t}^{(i)}\}_{i\in[n]\setminus\{i_\mathfrak{t}\}}$, $(\Lambda_{\mathfrak{t},j})_{j\in[\iota]}$ and $\mathsf{msg}_\mathfrak{t}^{(i_\mathfrak{t})}$ to recompute the online phase. Check that the output reconstruction yields the correct value of $\boldsymbol{y}$, and compute $h'_{\mathsf{m},\mathfrak{t}} = H(\{\Lambda_{\mathfrak{t},j}\}_{j\in[\iota]}, \mathsf{msg}_\mathfrak{t}^{(1)}, \ldots, \mathsf{msg}_\mathfrak{t}^{(n)})$.
5. Then compute $h'_\mathsf{m} = H(h'_{\mathsf{m},1}, \ldots, h'_{\mathsf{m},T})$ using $(h'_{\mathsf{m},\mathfrak{t}})_{\mathfrak{t}\in\overline{\mathcal{T}}}$ sent by $\mathcal{P}$.
6. Check that $h^* \stackrel{?}{=} H(h'_\mathsf{s}, h'_\mathsf{m})$.

Fig. 2: 3-round HVZK proof (part 2).

To obtain a formula for our proof size estimates we analyze each of the elements communicated by the prover in Rounds 1 and 3. We present and incorporate the optimizations discussed in [KKW18] and thus obtain a formula very close to theirs. At the end of Round 1, $\mathcal{P}$ sends $h^* = H(h_\mathsf{s}, h_\mathsf{m})$ to $\mathcal{V}$, which contributes 2κ bits to the proof size. In Round 3, $\mathcal{P}$ sends two sets of elements; the first corresponds to the $T - \tau$ opened preprocessing executions and the second corresponds to the τ executions of the online phase of the protocol. By generating the master seeds as the leaves of a binary tree expanding from a single root, all-but-τ of the seeds can be sent by only sending $\tau \cdot \log(T/\tau)$ elements, each of κ bits. By computing the hash h_m in a similar way, i.e. as the root of a tree where the $h_{\mathsf{m},\mathfrak{t}}$ values are the leaves, then it also suffices to send at most $\tau \cdot \log(T/\tau)$ values, each of 2κ bits. This implies that the opening of the $T - \tau$ preprocessing executions adds $\tau \cdot \log(T/\tau) \cdot 3\kappa$ to the proof size. To open the τ online executions, $\mathcal{P}$ sends, for each execution, the commitment $\gamma_{\mathsf{s},\mathfrak{t}}^{(i_\mathfrak{t})}$ of 2κ bits, the input correction values $\{\Lambda_{\mathfrak{t},j}\}_{j\in[\iota]}$ of size $|\boldsymbol{w}|$ bits, the messages of the unopened party of size $|\mathsf{msg}^{(i)}|$ and the states of the openened parties. To reduce the communication here, it is observed in [KKW18] that there is sufficient entropy contained in $\mathsf{state}_t^{(i)}$ to not require a separate randomness $r_t^{(i)}$. Combining this with a tree-like structure thus reduces the communication to only $\kappa \cdot \log n$ for the states. In the worst case when party P_n is opened, the prover also has to send the auxiliary information of size $|\mathsf{aux}^{(n)}|$. This results in the following estimate for the proof sizes.

$$2 \cdot \kappa + \tau \cdot \log(T/\tau) \cdot 3 \cdot \kappa + \tau \cdot (\kappa \cdot \log n + 2 \cdot \kappa + |\mathsf{aux}^{(n)}| + |\boldsymbol{w}| + |\mathsf{msg}^{(i)}|). \tag{1}$$

We have $\mathsf{aux}^{(n)} = \{\Delta_m\}_{m\in[\mathsf{mult}]}$ and typically, the majority of messages in $\mathsf{msg}^{(i)}$ are the openings of α and β during the multiplications. We see that when κ, n, T and τ are fixed, the final proof size is strongly correlated with the number of multiplications in the circuit.

2.2 The Picnic Signature Scheme

It is straightforward to use the NIZKPoK described in the previous section to obtain the Picnic signature scheme [CDG+17]. Given a block cipher $F_k(\boldsymbol{x}) : \mathcal{K} \times \mathcal{X} \to \mathcal{Y}$, presented as a binary circuit, the scheme is described by three algorithms:

– $\mathsf{Gen}(1^\kappa)$: Sample $\boldsymbol{x}$ in $\mathcal{X} = \{0,1\}^\kappa$, and $k \in \mathcal{K} = \{0,1\}^\kappa$ and then compute $\boldsymbol{y} = F_k(\boldsymbol{x})$. The public key pk is given by $(\boldsymbol{y}, \boldsymbol{x})$ and the secret key is $\mathsf{sk} = k$.
– $\mathsf{Sign}(\mathsf{sk}, m)$: Given a message m to be signed, compute $\sigma \leftarrow \mathsf{Prove}(\mathsf{pk}, k)$. Compute the challenge internally as $H(\tilde{\sigma}, m)$, where $\tilde{\sigma}$ denotes the message sent in **Round 1** of the proof.
– $\mathsf{Verify}(\mathsf{pk}, m, \sigma)$: Compute $\mathsf{VerifyProof}(\mathsf{pk}, \sigma)$ and the challenge as $H(\tilde{\sigma}, m)$. Return 1 if the result of $\mathsf{VerifyProof}$ is 1 and 0 otherwise.

The Picnic signature scheme is thus a NIZKPoK of k realized with the HVZKPoK protocol of [KKW18] for binary circuits. The size of the proof σ then depends on the number of AND gates required for the evaluation of the symmetric primitive F. In the submission to the NIST's Post-Quantum Cryptography project [Nat16,CDG+19b], F is instantiated with LowMC [ARS+15], an "MPC friendly" block-cipher explicitly designed to have low AND depth and lower multiplication complexity. LowMC is a very parametrizable scheme, the number s of 3-bits S-boxes per round and the number of rounds r can both be modified to favor either low round complexity or low multiplication complexity. The block-size and key-size κ does not affect the number of S-boxes, it only imposes the condition that $3s \leq \kappa$; bits of the state that are not affected by the S-box layer are left unchanged. In [CDG+19a], the Picnic submission team provides parameters (κ, s, r) for the LowMC block-cipher and parameters $(n = 64, \kappa, T, \tau)$ for the NIZKPoK. We reproduce these parameters in Table 1 together with the number of AND gates in each circuit and the estimated signature size using the formula given in (1). As per the MPC protocol for binary circuits given in [KKW18], we use $|\mathsf{aux}^{(n)}| = |\mathsf{msg}^{(i)}| = (\#\mathsf{ANDs})$ and $|\boldsymbol{w}| = \kappa$ in our estimates. We note that these estimated proof sizes consistently fall in between the maximum and averages sizes reported in the Picnic submission [CDG+19a] and we will therefore compare our estimations to these.

Scheme	κ	s	r	#ANDs	T	τ	est. size
picnic2-L1-FS	128	10	20	600	343	27	12.7 kB
picnic2-L3-FS	192	10	30	900	570	39	28.1 kB
picnic2-L5-FS	256	10	38	1140	803	50	47.9 kB

Table 1: Picnic parameters and estimated proof sizes.

Key generation security. As discussed in [CDG+17, Appendix D], the security of the key generation relies on the assumption that the block cipher $F_k(\boldsymbol{x})$ is a one-way function with respect to k. That is, for a fixed plaintext block $\boldsymbol{x} \in \mathcal{X}$, the function $f_{\boldsymbol{x}} : \mathcal{K} \to \mathcal{Y}$ defined by $f_{\boldsymbol{x}} : k \mapsto F_k(\boldsymbol{x})$ is a OWF. This is indeed the correct assumption since $\boldsymbol{x}$ and $\boldsymbol{y}$ as part of the public-key, thus fixing the function $f_{\boldsymbol{x}}$ and posing the challenge of recovering a suitable pre-image $\tilde{k} \in \mathcal{K}$ such that $f_{\boldsymbol{x}}(\tilde{k}) = F_{\tilde{k}}(\boldsymbol{x}) = \boldsymbol{y}$. This also shows that first picking $\boldsymbol{x}$ and then picking k during key generation is the natural way of first fixing the function $f_{\boldsymbol{x}}$ and then computing the challenge image $\boldsymbol{y}$.

In the full version of the original Picnic work [CDG+17], the authors go on to prove that if the block cipher $F_k(\boldsymbol{x})$ is a PRF family *with respect to* $\boldsymbol{x}$ (that is for a fixed k, the function $F_k : \mathcal{X} \to \mathcal{Y}$ is a PRF) then it is also a OWF *with respect to* k.

2.3 The Advanced Encryption Standard

AES is a 128-bit block-cipher based on a substitution-permutation network (SPN). It allows key lengths of 128, 192 or 256 bits and the corresponding number of rounds for the SPN is respectively 10, 12 or 14. The state always consists of 128 bits and can be considered as a 4×4 matrix of elements in $\mathbb{F}_{2^8}$. The cipher can thus be considered either as a Boolean circuit over $\mathbb{F}_2$ or as an arithmetic circuit over $\mathbb{F}_{2^8}$. We consider the latter. The round function is composed of four operations on the state, of which only one is non-linear: SubBytes.

AddRoundKey takes the 128-bit round key produced by the key schedule and performs an exclusive-or (XOR) operation with the current state to obtain the new state. As $\mathbb{F}_{2^8}$ has characteristic 2, the XOR operation can be computed by simply adding two elements together.

SubBytes is the only non-linear block of the round function, which transforms each of the 16 bytes of the state by means of a substitution function, known as the S-box. The AES S-box is a multiplicative inverse in the field $\mathbb{F}_{2^8}$, followed by an invertible affine transformation. In some sense, the S-box can be seen as

$$S : s \mapsto \phi^{-1}\left(\mathbf{A} \cdot \phi\left(s^{-1}\right) + \vec{b}\right) \tag{2}$$

where $\phi : \mathbb{F}_{2^8} \to (\mathbb{F}_2)^8$ is an isomorphism of vector spaces (mapping bytes in $\mathbb{F}_{2^8}$ to vectors of eight bits in $\mathbb{F}_2$) and $\mathbf{A} \in (\mathbb{F}_2)^{8\times 8}$ and $\vec{b} \in (\mathbb{F}_2)^8$ are the public parameters of the affine transformation.

ShiftRows is a permutation of the 16 state bytes, obtained by rotating row i of the state by i bytes to the left. Hence, the first row (row zero) remains the same, the second row is rotated to the left by one byte, etc.

MixColumns is the final phase and linear component. It consists of a mixing operation that is applied to each column of the state separately. Column i is transformed by a matrix multiplication with a 4×4 matrix defined over $\mathbb{F}_{2^8}$.

Key Schedule. The key schedule is mostly linear except for the application of the same AES S-box to up to four bytes of the round key.

As the S-box is the only non-linear block, Table 2 presents how many times it needs to be computed for each AES circuit.

AES-	128	192	256
# Rounds	10	12	14
# Round S-boxes	160	192	224
# Key schedule S-boxes	40	32	52
Total # S-boxes	200	224	276

Table 2: Number of S-boxes in the AES circuits.

Rijndael. The AES is the standardized version of the Rijndael cipher [DR99]. The most prominant difference between the two is that Rijndael allows both a variable key size κ and a variable block-length β. The round transformations are the same, with SubBytes performing S-box on each of the $\frac{\beta}{8}$ state bytes and MixColumns transforming each of the $\frac{\beta}{32}$ state columns. The key schedule is identical to that of AES-κ, but keep in mind that the expanded key is larger in the former case and hence the number of S-box calculations also differs. We summarize the number of S-boxes in Rijndael-κ circuits with $\kappa = \beta$ in Table 3.

Rijndael-	128	192	256
# Rounds	10	12	14
# Round S-boxes	160	288	448
# Key schedule S-boxes	40	48	112
Total # S-boxes	200	336	560

Table 3: Number of S-boxes in the Rijndael circuits.

3 Computing the S-box

To design an MPC protocol for the circuit $f_{\boldsymbol{x}}(\boldsymbol{k}) \coloneqq \mathsf{AES}_{\boldsymbol{k}}(\boldsymbol{x})$ over $\mathbb{F}_{2^8}$, we design a communication-efficient computation of the non-linear S-box in a distributed way using the advantages of MPCitH. This computation happens in two stages: first an *inversion* stage described by the map

$$s \mapsto \begin{cases} s^{-1} & \text{if } s \neq 0, \\ 0 & \text{if } s = 0, \end{cases} \quad \text{over } \mathbb{F}_{2^8}; \tag{3}$$

and second, an *affine* stage where $s^{-1} \mapsto \phi^{-1}(\mathbf{A} \cdot \phi(s^{-1}) + \vec{b})$ in $(\mathbb{F}_2)^8$ as described in Equation (2). As the second stage is in fact local, we present it first and then address the inversion stage in Section 3.1.

The affine transformation (being linear) can be applied by each party independently. Since it operates on individual bits, the parties must first derive a sharing of the bit-decomposition of s. As we only require semi-honest security for our protocol, this can be also be achieved locally. Given the isomorphism $\phi : \mathbb{F}_{2^8} \to (\mathbb{F}_2)^8$, each party can locally obtain $\vec{s}^{(i)} = \phi(s^{(i)})$ which then corresponds to a sharing $\langle \vec{s} \rangle = \phi(\langle s \rangle)$. Each party can then locally compute the affine transformation $\vec{t}^{(i)} \coloneqq \mathbf{A} \cdot \vec{s}^{(i)} + \vec{b}$; indeed, as $\mathbf{A}$ and $\vec{b}$ are both public values, the transformation can be computed as a series of multiplications by and additions with public constants during the online phase.. Finally, they can recompose $\langle t \rangle = \phi^{-1}(\langle \vec{t} \rangle)$. Thus the affine phase of the S-box can be computed entirely locally.

3.1 Computing the Inversion

To perform the Galois field inversion within the MPCitH paradigm, we use a masked inversion method. Namely, given $\langle s \rangle \in \mathbb{F}_{2^8}$ and a randomly sampled $\langle r \rangle \in \mathbb{F}_{2^8}$ (where each party samples $r^{(i)}$ at random from $\mathsf{seed}^{(i)}$), we run the following:

1. Compute $\langle s \cdot r \rangle$ (by using a preprocessed triple $(\langle a \rangle, \langle b \rangle, \langle c \rangle)$ and opening $s - a$ and $r - b$).
2. $\mathsf{Open}(s \cdot r)$.
3. Compute $(s^{-1} \cdot r^{-1})$ (done locally by each P_i).
4. Compute $\langle s^{-1} \rangle = (s^{-1} \cdot r^{-1}) \cdot \langle r \rangle$.

Performing the multiplication and the opening of $\langle s \cdot r \rangle$ results in every party broadcasting three elements of $\mathbb{F}_{2^8}$ for each inversion. This implies that we have a base communication cost of 3 bytes per party per inversion in the online phase; i.e. for each P_i,

$$|\mathsf{msg}^{(i)}| = 8 \cdot (3 \cdot \#(\text{S-boxes}) + o),$$

where the additional o bytes come from the opening of the output values as $f_{\boldsymbol{x}}$ outputs tuples in $(\mathbb{F}_{2^8})^o$.

Also considering the inclusion of the offset value $\Delta \in \mathbb{F}_{2^8}$ in the auxiliary information $\mathsf{aux}^{(n)}$, for the triple that is used for the inversion, we also have that

$$|\mathsf{aux}^{(n)}| = 8 \cdot \#(\text{S-boxes}).$$

There are however two complications with this method: when either $r = 0$ or $s = 0$. We first present how to deal with the first one, assuming that the second does not happen. We then discuss how to mitigate the second one.

Need for Non-Zero Randomness. Indeed, assuming that $s \neq 0$, we see that if $r = 0$, then $s \cdot r = 0$ as well and the inversion computation cannot proceed. However, this does not leak any information regarding s, so when the parties observe that $\mathsf{Open}(s \cdot r)$ yields 0, they can restart the computation of the inversion with a fresh $\langle r' \rangle$ and a new multiplication triple.

While this would not be a problem for real MPC executions of this protocol, here we are restricted because of the commitment to $\mathsf{aux}^{(n)}$, and therefore to the number of preprocessed triples, that the prover $\mathcal{P}$ of the HVZKPoK must make before it emulates the online phase of the protocol. The prover does emulate the online phase for every preprocessing phase, and it would therefore be possible for them to observe exactly how many triples are required for an execution. However, this would ultimately be dependent on the input.

In order to commit to the preprocessing, the prover would have to indicate how many triples were generated and using which bits of randomness, given that those bits produced from $\mathsf{seed}^{(i)}$ would be mixed with bits used to produce other randomness, such as the input sharings. We do not discuss here the design of a protocol which accounts for this flexible verification and the evaluation of whether this would leak information regarding the input to the verifier.

Instead, we design our protocol to generate a fixed number m of additional triples for every execution. In the rare cases that this additional number is not enough, we say that the prover aborts the proof and restarts. Table 4 shows the probability that a proof aborts for each of the AES circuits depending on the number of additional triples provided by $\mathcal{P}$. If we want the probability of needing to abort a proof to be less than $10^{-8} \approx 2^{-20}$, then adding $m = 9$ additional triples is sufficient.

The addition of these m additional triples then implies that, for each execution, the auxiliary information contains $(\#(\text{S-boxes}) + m)$ elements in $\mathbb{F}_{2^8}$ and we therefore have

$$|\mathsf{aux}^{(n)}| = 8 \cdot (\#(\text{S-boxes}) + m).$$

m	0	1	2	3	4	5	6	7	8	9	10
AES-128	$2^{-0.9}$	$2^{-2.4}$	$2^{-4.5}$	$2^{-6.9}$	$2^{-9.7}$	$2^{-12.7}$	$2^{-15.9}$	$2^{-19.3}$	$2^{-22.9}$	$2^{-26.6}$	$2^{-30.5}$
AES-192	$2^{-0.8}$	$2^{-2.2}$	$2^{-4.1}$	$2^{-6.4}$	$2^{-9.0}$	$2^{-11.8}$	$2^{-14.8}$	$2^{-18.1}$	$2^{-21.5}$	$2^{-25.1}$	$2^{-28.8}$
AES-256	$2^{-0.6}$	$2^{-1.8}$	$2^{-3.4}$	$2^{-5.4}$	$2^{-7.7}$	$2^{-10.2}$	$2^{-13.0}$	$2^{-15.9}$	$2^{-19.0}$	$2^{-22.3}$	$2^{-25.7}$

Table 4: Probability of abort of each circuit (assuming $\Pr[r = 0] = \frac{1}{256}$).

Similarly, since parties are susceptible to repeat the computation of up to m inversions during the protocol and therefore broadcast more than 3 bytes per inversion, we have that for each P_i

$$|\mathsf{msg}^{(i)}| \leq 8 \cdot (3 \cdot (\#(\text{S-boxes}) + m) + o).$$

One advantage of restricting the number of triples produced in this way is that it guarantees a maximum proof size, independently of the randomness used.

3.2 Need for Non-Zero Input

In the previous section we assumed that $s \neq 0$ to say that if $s \cdot r = 0$, then it must be that $r = 0$. However, in general, it is possible for the input s to the AES S-box to be 0. If the inversion was computed as above, with this possibility, then opening $s \cdot r = 0$ would in fact reveal information about s since r is not allowed to be zero in a correct inversion.

To avoid this leakage, we restrict our protocol to only prove knowledge of values of $\boldsymbol{k}$ for which there is no zero as input to any S-box in the computation of $f_{\boldsymbol{x}}(\boldsymbol{k})$. We write $\mathcal{K}_{\boldsymbol{x}} \subset \mathcal{K}$ to denote the subset of such keys. While this reduces the number of applications of this protocol as a general HVZKPoK for AES keys, this is not so significant in the context of Picnic signatures. Indeed, the values $\boldsymbol{x}$ and $\boldsymbol{k}$ for which the proof must be given are fixed during key generation and used for all signatures. It is therefore feasible to restrict the Keygen algorithm to first select a random $\boldsymbol{x}$ and then sample values of $\boldsymbol{k}$ until one from $\mathcal{K}_{\boldsymbol{x}}$ is found.

We note that a malicious prover could intentionally generate a key for which some zeroes do appear in the computation of the circuit. However, in the context of PKI, users typically present a signature of their certificate under their public-key as a proof of possession. Check such a signature would immediately reveal whether a key was malformed and thus prevent verifiers from accepting further signatures.

Loss of Security. This particular design of the Keygen procedure naturally affects the assumption that $f_{\boldsymbol{x}}(\boldsymbol{k}) \coloneqq \mathrm{AES}_{\boldsymbol{k}}(\boldsymbol{x})$ is still a OWF family as required for Picnic signatures. While the random selection of $\boldsymbol{x}$ still ensures that a random function $f_{\boldsymbol{x}}$ is selected from the family, the restricted selection of $\boldsymbol{k} \in \mathcal{K}_{\boldsymbol{x}}$ reduces the image space of $f_{\boldsymbol{x}}$ and also the pre-image space for a given $\boldsymbol{y}$.

We can first estimate this security loss by assuming that, for a given block $\boldsymbol{x}$ and a randomly sampled key $\boldsymbol{k} \in \mathcal{K}$, the S-boxes are independent from one another and each has a probability of $\frac{1}{256}$ of receiving 0 as input. Table 5 presents the proportion of keys that would also belong to $\mathcal{K}_{\boldsymbol{x}}$ for the three different AES circuits, *i.e.* $(255/256)^{\#\text{ S-boxes}}$. For AES-256, we see that approximately $\frac{1}{3}$ of keys possess this property, which, from an attacker's perspective, reduces by only $\log_2(3) \approx 1.4$ the bit-security of the problem of inverting the OWF $f_{\boldsymbol{x}}$ and recovering $\boldsymbol{k} \in \mathcal{K}_{\boldsymbol{x}}$. The assumption that the S-boxes are independent is justified by the PRF-behaviour or AES. Moreover, our experiments

Circuit	AES-128	AES-192	AES-256
Number of S-boxes	200	224	276
No-zeroes probability	45.7%	41.6%	34.0%

Table 5: Probabilities of no zero-inputs to S-boxes.

confirmed that, for a fixed $\boldsymbol{x}$, the same proportions of keys as shown in Table 5 yielded no zero-inputs.

This also shows that it is not very computationally expensive for Keygen to re-sample $\boldsymbol{k}$ as it will only have to do so three times on average. To emphasize the low cost of this, we also recall that during Keygen, the signer samples and verifies that $\boldsymbol{k} \in \mathcal{K}_{\boldsymbol{x}}$ for a given $\boldsymbol{x}$. While they do this, they do not have to execute the AES circuit using the MPC protocol, the S-box inputs can be verified using an un-shared execution.

A more formal analysis. In [CDG+17, Appendix D], the authors show that if an adversary $\mathcal{A}$, on input $\boldsymbol{y}$ has a high probability of inverting the OWF $f_{\boldsymbol{x}}(\boldsymbol{k}) = \boldsymbol{y}$, then one can build a distinguisher $\mathcal{D}$ which has a high probability of distinguishing $F_{\boldsymbol{k}}$ from a random function. Recall that in our case, $F_{\boldsymbol{k}}$ is the usual $\text{AES}_{\boldsymbol{k}}$ PRF family and that $f_{\boldsymbol{x}}$ is the AES function family considered on $\boldsymbol{k}$ indexed by $\boldsymbol{x}$.

Let us now assume that the adversary $\mathcal{A}$ has a high probability p of inverting $f_{\boldsymbol{x}}(\boldsymbol{k})$ only if $\boldsymbol{k} \in \mathcal{K}_{\boldsymbol{x}}$. In the reduction proof, $\mathcal{D}$ queries the PRF oracle $\mathcal{O}$ on input $\boldsymbol{x}$ to receive $\boldsymbol{y}$ as its image either under the PRF $F_{\boldsymbol{k}}$ for a random $\boldsymbol{k} \in \mathcal{K}$ or under a random function. It then queries $\mathcal{A}$ on input $\boldsymbol{y}$ in the hope of receiving the correct key $\boldsymbol{k}$. The only difference between our situation and theirs is that the key space $\mathcal{K}_{\boldsymbol{x}}$ for which $\mathcal{A}$ is capable of returning the correct key is smaller than the whole of $\mathcal{K}$. In the case where $\mathcal{O}$ is indeed computing the PRF, there is then a probability of $|\mathcal{K}_{\boldsymbol{x}}|/|\mathcal{K}|$ that the sampled hidden key will belong to the reduced key space and therefore $\mathcal{A}$ will return the correct key with smaller probability $p \cdot |\mathcal{K}_{\boldsymbol{x}}|/|\mathcal{K}|$. All other aspects of the proof are the same as in [CDG+17] and therefore, under the assumption that AES, over any key $\boldsymbol{k} \in \mathcal{K}$, is a PRF with respect to $\boldsymbol{x}$, it holds that it also is a OWF with respect to $\boldsymbol{k} \in \mathcal{K}_{\boldsymbol{x}}$ with a tightness gap proportional to $|\mathcal{K}_{\boldsymbol{x}}|/|\mathcal{K}|$ for a given $\boldsymbol{x}$.

We can therefore say that if our AES circuit with restricted keys would yield a weak OWF, then there would be a similar proportion of keys for which it would be easy to distinguish the AES function from a random one, thus contradicting the assumed and observed PRF behaviour of the AES. Furthermore our experiments confirm that this proportion of keys is non-negligible, further increasing confidence in our OWF construction.

3.3 Application to Picnic Signatures

We now estimate the signature size if our arithmetic circuit $C_{\boldsymbol{x}}(\boldsymbol{k}) = \text{AES}_{\boldsymbol{k}}(\boldsymbol{x})$ were to be used instead of the LowMC binary circuit in Picnic signatures and also present the interesting conclusion that a naive application of AES-192 or AES-256 is not sufficient to achieve stronger levels of security.

AES-128: Table 6 presents our estimates using the AES-128 circuit instead of the LowMC circuit with parameters $(k = 128, s = 10, r = 20)$, specified by the Picnic submission [CDG+19a] as achieving AES-128-like security. For the picnic2-AES-128-bin circuit, we used the state-of-the-art

Level	Circuit	κ	T	τ	$\lvert\mathsf{aux}^{(n)}\rvert$	$\lvert w\rvert$	$\lvert\mathsf{msg}^{(i)}\rvert$	Est. size
L1	picnic2-AES-128-bin	128	343	27	6400	128	6400	51.9 kB
	picnic2-AES-128-inv				1664	128	5120	**31.6 kB**

Table 6: Estimates for Picnic signatures with AES-128 circuit.

figure of 32 AND gates per AES S-box reported in the inversion circuit of [BMP13], thus yielding $\lvert\mathsf{aux}^{(n)}\rvert = \lvert\mathsf{msg}^{(i)}\rvert = 200 \cdot 32 = 6400$. We note that this differs from the figure of 5440 AND gates stated in [CDG+17, Section 6.1]. While we cite the same work [BMP13], a figure of 5440 would make sense if the AES-128 circuit contained only 170 S-boxes, or 160 S-boxes with 34 AND gates per S-box. Our Table 2 shows that if the full circuit is considered, including the key schedule, then 200 S-boxes are necessary. Moreover, removing the key schedule is not possible as a fraudulent prover would then be able to create a forgery by selecting a final round key which would make the second-to-last state agree with the public key value $\boldsymbol{y}$. Finally, for our picnic2-AES-128-inv circuit, we used our costs of $\lvert\mathsf{aux}^{(n)}\rvert = 8 \cdot (200 + 8) = 1664$ and $\lvert\mathsf{msg}^{(i)}\rvert = 8 \cdot (3 \cdot (200 + 8) + 16) = 5120$ accounting for $m = 8$ additional inversion operations and $o = 16$ bytes of communication per party for output reconstruction.

We see that implementing AES-128 using the best current binary circuit would increase the signature size by a factor of 4.07 compared to the estimated signature size for picnic2-L1-FS given in Table 1. However, our inversion technique yields an increase only by a factor of 2.48, thus a reduction of 39% over the binary circuit. We note that this improvement comes with the caveat that our technique of Section 3.2 reduces the key-space for $\boldsymbol{k}$ by $\lvert\log_2(0.457)\rvert = 1.13$ bits from the 128 bit-security level.

AES-192 and AES-256: Such a direct comparison as the one presented above is not possible for the AES-192 and AES-256 circuits. While these two algorithms are believed to provide respectively 192 and 256 classical bit-security when used as block-ciphers, this does not hold in this paradigm due to the requirement for a one-way function for the key generation of Picnic-like signatures.

Indeed, as remarked in [CDG+17, Appendix D], the security of the key generation requires the block-size and the key-size of the underlying block-cipher to be equal – in part to prevent quantum attacks which benefit from a square-root speedup over the block-size. This is not the case for AES-192 and AES-256 and therefore, interestingly, the shift to longer keys, without an accompanying shift to longer blocks, does not translate into increased security against forgery attacks for the signature scheme. Here, the standardized 128 bit block-length of the AES cipher becomes an obstacle to easily achieving both stronger security and optimal efficiency. For AES-192, for example, a single block of encryption would result in (an expected) 2^{64} spurious keys per single block $\boldsymbol{x}$, $\boldsymbol{y}$ pair. Thus the probability of guessing a valid key, with a single block pair $\boldsymbol{x}$, $\boldsymbol{y}$ will be 2^{-128} and not 2^{-192} as desired, and in addition applying Grover's search for a valid key would only be slightly more complex than in the AES-128 case.

This fixed block-length was actually not a part of the original Rijndael design [DR99]. Hence, to obtain L3 (resp. L5) constructions, one could alternatively use the Rijndael cipher with 192-bit (resp. 256-bit) blocks and key. Since these are not standardized, we first discuss some constructions using AES.

To design a circuit suitable for 192 (resp. 256) bit-security level Picnic signatures, we therefore combine two copies of an AES-192 (resp. AES-256) circuit, keyed with the same key. To realize

such a cipher with longer block- and key-length, we propose to use AES in ECB mode. For our L3 construction using AES-192, we only require $\boldsymbol{x}$ and $\boldsymbol{y}$ to be 192 bits long. We therefore pad $\boldsymbol{x}$ with 64 0s to reach two full blocks, and then truncate the resulting ECB-mode encryption to its first 192 bits (one and a half blocks) to produce the output $\boldsymbol{y}$. For our L5 construction using AES-256, we use an $\boldsymbol{x}$ value of 256 bits, encrypted as two blocks in ECB mode, to produce a two-block value $\boldsymbol{y}$ of 256 bits.

One side-effect of using ECB is that we have the problem of malleable public keys, for example the public key $(\boldsymbol{x}_0\|\boldsymbol{x}_1, \boldsymbol{y}_0\|\boldsymbol{y}_1)$ would be equivalent to $(\boldsymbol{x}_1\|\boldsymbol{x}_0, \boldsymbol{y}_1\|\boldsymbol{y}_0)$. Whilst this does not break the security of the signature scheme in the standard security game, this could be a problem in practice. However, we can tie signatures to a specific public key by including it in the hash used to generate the challenge in the NIZKPoK.

The security of key generation now depends on the ECB construction which is clearly not a PRF family, but we can instead rely on the assumption that ECB mode is OW-CPA and we provide a similar argument as in [CDG+17] to formalise this assumption.

Claim. If $F_k(x)$ is OW-CPA then $f_x(k)$ is a OWF.

Proof (sketch). The proof follows similary to the proof in [CDG+17, Appendix D]. Indeed, given an adversary $\mathcal{A}$ that can return k on input $y = f_x(k)$, we can build a adversary $\mathcal{B}$ against the OW-CPA challenger for F_k as follows. First, $\mathcal{B}$ submits x to the encryption oracle and receives $y = F_k(x)$ for random $k \in \mathcal{K}$ chosen by the oracle. Then $\mathcal{B}$ runs $\mathcal{A}$ on input y to obtain k with high probability. Finally, $\mathcal{B}$ requests the challenge ciphertext y^* and uses k to compute $x^* = F_k^{-1}(y^*)$ and return it as its answer to the OW-CPA challenge. With high probability, $\mathcal{B}$ returns the correct pre-image.

Both of the ECB-mode circuits result in executing two AES circuits but this drawback is inherent to the fixed block-length of the AES block-cipher and therefore applies equally to our inversion approach and the naive binary approach. We estimate the proof sizes obtained with these designs in Table 7.

For the binary circuits, we use the same figure of 32 AND gates per S-box [BMP13]. In the case of the picnic2-AES-192-bin and picnic2-AES-256-bin circuits, we double the number of round S-boxes presented in Table 2 to obtain figures for $|\mathsf{aux}^{(n)}|$ and $|\mathsf{msg}^{(i)}|$. We keep the same number of key-schedule S-boxes as both circuits use the same key $\boldsymbol{k}$ and therefore the key-schedule needs to be computed only once. For the inversion circuits, we also double the number of round S-boxes in our formulæ for $|\mathsf{aux}^{(n)}|$ and $|\mathsf{msg}^{(i)}|$ but we must also change number m of additional triples due to the increased risk of sampling zero randomness during the inversion computation. A quick calculation similar to the one used for Table 4 shows that our doubled-up circuits for AES-192 and AES-256 require $m = 11$ and $m = 12$ respectively to have a probability of abort below 2^{-20}.

We see that implementing our doubled-up AES-192 and AES-256 using the best current binary circuit would increase the signature sizes by a factor of 5.30 and 4.87 respectively compared to the estimated signature size for picnic2-L3-FS and picnic2-L5-FS given in Table 1. However, our inversion technique yields increases only by a factor of 3.09 and 2.79 respectively, thus reductions of 41.7% and 42.8% over the binary circuits. We note that this improvement comes with the caveat that our technique of Section 3.2 reduces the key-space for $\boldsymbol{k}$ by $|\log_2(0.196)| = 2.35$ bits and $|\log_2(0.141)| = 2.83$ bits from the 192 and 256 bit-security levels respectively.

Rijndael-192 and -256. To avoid the reliance on ECB mode and on a slightly different security assumption, we also provide estimates of proof sizes that make use of the original Rijndael-192

Level	Circuit	κ	T	τ	$\lvert\mathsf{aux}^{(n)}\rvert$	$\lvert w\rvert$	$\lvert\mathsf{msg}^{(i)}\rvert$	Est. size
L3	picnic2-AES-192-bin	192	570	39	13312	192	13312	149.1 kB
	picnic2-Rijndael-192-bin				10752	192	10752	124.2 kB
	picnic2-AES-192-inv				3416	192	10440	**86.9 kB**
	picnic2-Rijndael-192-inv				2768	192	8496	**74.2 kB**
L5	picnic2-AES-256-bin	256	803	50	16000	256	16000	233.7 kB
	picnic-Rijndael-256-bin				17920	256	17920	257.7 kB
	picnic2-AES-256-inv				4096	256	12544	**133.7 kB**
	picnic2-Rijndael-256-inv				4576	256	13984	**149.7 kB**

Table 7: Estimates for Picnic signatures with AES-192, AES-256 and Rijndael circuits.

and -256 circuits which use an equal block and key size with the same S-box as AES. The number of rounds in Rijndael-192 (resp. Rijndael-256) is the same as in AES-192 (resp. AES-256), i.e. 12 (resp. 14)[3].

We use the figures of Table 3 to obtain figures for $\lvert\mathsf{aux}^{(n)}\rvert$ and $\lvert\mathsf{msg}^{(i)}\rvert$, presented in Table 7. As above, we calculate that values of $m = 10$ and $m = 12$ are required by the Rijndael-192 and -256 circuits respectively to have a probability of abort below 2^{-20}.

We see that implementing the Rijndael-192 and -256 circuits using the best current binary circuit would increase the signature sizes by a factor of 4.42 and 5.38 respectively compared to the estimated signature size for picnic2-L3-FS and picnic2-L5-FS given in Table 1. However, our inversion technique yields increases only by a factor of 2.64 and 3.12 respectively, thus reductions of 40.2% and 41.8% over the binary circuits. We note that this improvement comes with the caveat that our technique of Section 3.2 reduces the key-space for $\boldsymbol{k}$ by $\lvert\log_2(0.268)\rvert = 1.90$ bits and $\lvert\log_2(0.112)\rvert = 3.16$ bits from the 192 and 256 bit-security levels respectively.

It is not surprising that the Rijndael-192 results are overall better those with AES-192, since the ECB construction with extra padding performs more work than it should. On the other hand, the AES-256 constructon is more efficient than the Rijndael-256 design since the latter has a larger key schedule.

4 Alternative Computations of the AES S-box

Representing the AES S-box as the map $s \mapsto s^{-1}$ (with $0 \mapsto 0$) over $\mathbb{F}_{2^8}$ is one of three methods described by Damgård and Keller [DK10] for this task in standard MPC. In fact, it is the most efficient method of that work to compute the AES arithmetic circuit in real-world MPC. However, their treatment of the $s = 0$ difficulty differs from ours, and they also present two alternative methods to compute the inversion operation. We discuss these methods and their cost here, along with other methods, to study the possibilities for computing the AES circuit in the MPCitH paradigm.

Square-and-multiply. The first method of [DK10] is not to compute $s \mapsto s^{-1}$ but rather $s \mapsto s^{254}$. As $\mathrm{ord}(\mathbb{F}_{2^8}{}^*) - 1 = 255 - 1 = 254$, this achieves the same result with the additional advantage of not

[3] See https://csrc.nist.gov/csrc/media/projects/cryptographic-standards-and-guidelines/documents/aes-development/rijndael-ammended.pdf.

requiring a special case for $s = 0$ which maps to 0 naturally. However, s^{254} requires a square-and-multiply chain to be computed. The shortest such chain given in [DK10] requires 11 multiplications. Combining the cost of the openings and the auxiliary information required for triple generation, this method would cost $11 \cdot 3 = 33$ bytes per S-box, compared to only 4 bytes per S-box for our method in Section 3.

Masked Exponentiation. This second method of [DK10] computes the same map $s \mapsto s^{254}$ but using the fact that $(a+b)^{2^i} = a^{2^i} + b^{2^i}$ in fields of characteristic 2. Here, they are able to take advantage of the preprocessing model; they first pre-compute the squares $\langle r \rangle, \langle r^2 \rangle, \langle r^4 \rangle, \ldots, \langle r^{128} \rangle$ for a random value $r \in \mathbb{F}_{2^8}$. Then, to invert $\langle s \rangle$, they additively mask $\langle s \rangle + \langle r \rangle$, open $(s+r)$ and square locally to obtain $(s+r)^2, (s+r)^4, \ldots, (s+r)^{128}$. Finally, they unmask each power-of-two with the corresponding shared power-of-two of r, $(s+r)^{2^i} + \langle r^{2^i} \rangle = \langle s^{2^i} \rangle$ and then perform an online multiplication chain to obtain $\prod_{i=1}^{7} \langle s^{2^i} \rangle = \langle s^{254} \rangle$.

Unlike in [DK10], the requirement on the MPC protocol for only semi-honest security means that the successive shared powers of $\langle r \rangle$ can be computed locally. Indeed, in characteristic 2, $r^2 = (\sum_{i=1}^{n} r^{(i)})^2 = \sum_{i=1}^{n} (r^{(i)})^2$, so each party can compute their shares locally. Therefore the cost of this method in our paradigm is the opening of $(s+r)$ and the six multiplications required for the computation of $\langle s^{128} \rangle$. This would still amount to $1 + 6 \cdot 3 = 19$ bytes per S-box (broadcast and auxiliary information put together).

Masked Inversion. In [DK10], Damgård and Keller also conclude that the masked inversion method that we presented in Section 3 is the most efficient. Indeed, at its core it only requires 3 bytes in the online phase and 1 triple in the preprocessing phase. However, the leakage that occurs when $s = 0$ must be prevented. While it is possible in our application to Picnic signatures to generate the public-/private-key pairs so that this leakage does not take place, this is not possible for generic computations of the AES circuit.

The solution given in [DK10] is to have the parties compute a shared "zero-indicator" value $\langle \delta(s) \rangle$ defined as

$$\delta(s) = \begin{cases} 1 & \text{if } s = 0, \\ 0 & \text{otherwise.} \end{cases}$$

It is then easily verified that the inversion mapping of (3) is equivalent to

$$s \mapsto (s + \delta(s))^{-1} + \delta(s)$$

When $\delta(s)$ is computed in shared form, then the parties can perform the inversion computation with $\langle s + \delta(s) \rangle$ instead of $\langle s \rangle$. Once $\langle (s + \delta(s))^{-1} \rangle$ is obtained, the parties can locally compute $\langle (s + \delta(s))^{-1} \rangle + \langle \delta(s) \rangle$. This ensures that if $s = 0$ then $s + \delta(s) = 1$ is instead inverted to $(s + \delta(s))^{-1} = 1$ and that the second addition of $\delta(s)$ returns the value to 0, as $\mathbb{F}_{2^8}$ has characteristic 2.

This technique could be applied to our MPCitH circuit for AES to avoid the loss of generality and security that come with the efficient method presented in the previous section. We present here two methods to compute the zero-indicator $\delta(s)$ in secret-shared form.

Method 1. In [DK10], the value $\langle \delta(s) \rangle$ is computed by first bit-decomposing $\langle s \rangle$, then NOT-ing each bit and finally computing the joint AND of the eight bits of s. This joint AND requires 7 bit-wise

AND gates to output the final value. If these gates are performed so that each party eventually holds $\delta(s)^{(i)} \in \mathbb{F}_2$, then each party can embed its share into $\mathbb{F}_{2^8}$ to obtain its share of $\langle \delta(s) \rangle$.

As before, bit-decomposition is trivially obtained by each party locally bit-decomposing its share into $\vec{s}^{(i)} = \phi(s^{(i)})$. The NOT gates are then also performed locally by each party. For the AND gates, we make use of the same technique of preprocessed multiplication triples, but over $\mathbb{F}_2$ instead of $\mathbb{F}_{2^8}$. For each AND gate, 1 bit then needs to be added to $\mathsf{aux}^{(n)}$ to correct the triple and 2 bits need to be opened in the online phase to compute the multiplication.

This method therefore results in $7 \cdot 3 = 21$ bits per S-box in addition to the 4 bytes required for the inversion of $\langle s + \delta(s) \rangle$. This results in approximately 6.6 bytes per S-box which is only 17.5% less than implementing the best binary circuit for AES in the model of [KKW18].

Method 2. In the world of MPC, the computation of the shared zero-indicator $\langle \delta(s) \rangle$ can be seen as an n-party functionality which takes $\langle s \rangle$ as input and distributes $\langle \delta(s) \rangle$ to all the parties. As we are in the MPCitH setting, the Prover can observe the value of s and act as this functionality by providing each party with their share of $\delta(s)$. To further optimize this, we can assume that P_i, for $i = 1 \to n-1$, samples their share of $\delta(s)$ pseudo-randomly and that the Prover only give P_n a correction value which makes the reconstructed value equal to $\delta(s)$.

This approach is the first here that deviates from a normal MPC scenario and exploits the MPCitH paradigm to reduce the cost. However, it comes with the caveat that a malicious Prover is able to select an arbitrary correction value for P_n. This implies that they have the freedom to arbitrarily modify the computation of the circuit. A partial fix for this is to specify that the shares of $\delta(s)$ must be either 0 or 1 within $\mathbb{F}_{2^8}$. This is only natural, as $\delta(s)$ can only take either of these values and the sum of such values is again 0 or 1. In this way, the prover can at most substitute a 1 for a 0 or vice-versa thus greatly limiting their ability to cheat arbitrarily. They can only inject an arbitrary $\mathbb{F}_{2^8}$ value into P_n and not get caught if P_n is the party that is not revealed to the Verifier, which corresponds exactly to their usual cheating capability in Picnic. Furthermore, this approach means that the additional auxiliary input required for P_n is only one bit of information, as opposed to one byte.

The remaining drawback is then that a malicious Prover can arbitrarily change the value of $\delta(s)$ from 0 to 1 in the computation. Note that this 1-bit fault can only be injected in the least significant bit of the input of the AES inversion, and if injected, the same fault is automatically applied to the inversion output as well, corresponding to the second XOR with $\delta(s)$. Unfortunately, it remains as an open question to compute what the soundness would be of a proof which used this technique for the computation of the AES circuit. This kind of fault injections have not been studied in the side-channel analysis literature, since a fault attacker typically injects faults in order to obtain an exploitable (non-zero) output difference in the ciphertext (*i.e.* Differential Fault Analysis [BS97]). A malicious prover in MPCitH on the other hand would be trying to modify an execution yielding a wrong ciphertext into an apparently correct one.

In the normal MPCitH scenario, a malicious prover who does not have knowledge of the secret key $\boldsymbol{k}$, has a probability $2^{-\kappa}$ to produce a valid signature by guessing the key, where $\kappa \in \{128, 192, 256\}$ is the key and block size. In this adapted scenario with very specific 1-bit faults, the size of the prover's search space increases to $2^{\kappa} \cdot 2^{\#\text{ S-boxes}}$, since they can guess both a key and a fault configuration by selecting which of the S-box computations to infect. Let $\tilde{\boldsymbol{k}}$ be the key guess and $\tilde{F}$ be the circuits of Section 3 with injected faults; then the prover succeeds if the correct ciphertext is output at the end, *i.e.* for a public key $(\boldsymbol{x}, \boldsymbol{y})$, $\tilde{F}_{\tilde{\boldsymbol{k}}}(\boldsymbol{x}) = \boldsymbol{y}$. We again assume that all S-box calculations are independent and treat AES as a PRF, hence we assume that each

of the 2^κ possible outputs are equiprobable. We thus estimate that the number of correct choices is

$$\#\text{ correct choices} \approx \frac{\text{total }\#\text{ choices}}{\#\text{ outputs}} = \frac{2^{\kappa+\#\text{S-boxes}}}{2^\kappa} = 2^{\#\text{S-boxes}}.$$

This results in the following success probability for a malicious prover:

$$\Pr[\tilde{F}_{\tilde{\boldsymbol{k}}}(\boldsymbol{x}) = \boldsymbol{y}] \approx \frac{\#\text{ correct choices}}{\text{total }\#\text{ choices}} = \frac{2^{\#\text{S-boxes}}}{2^{\kappa+\#\text{S-boxes}}} = 2^{-\kappa}$$

which corresponds exactly to the success probability in the MPCitH scenario without fault injections.

This methods originates from using an n-party functionality within the MPCitH paradigm, which is normally not permitted as a malicious Prover is then able to cheat arbitrarily. However our analysis shows that, for this particular application, this additional freedom does not seem to give the Prover further advantages to forge a signature. Therefore, while n-party functionalities may not be permitted in general, this shows that case-by-case analysis may reveal that they do not affect the soundness for specific proofs. This then raises the question of what other proofs may be realized by taking advantage of such functionalities, which are in fact "free", in terms of communication between parties, as they can be computed outside of the MPC protocol by the Prover themselves. We leave such a study for further work.

Acknowledgments

The authors would like to thank the referees for a number of comments which improved the quality of the paper. This work has been supported in part by ERC Advanced Grant ERC-2015-AdG-IMPaCT, by the Defense Advanced Research Projects Agency (DARPA) and Space and Naval Warfare Systems Center, Pacific (SSC Pacific) under contract No. N66001-15-C-4070, and by the Fund for Scientific Research Flanders (FWO) under an Odysseus project GOH9718N. Lauren De Meyer is funded by a PhD fellowship of the FWO. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the ERC, United States Air Force, DARPA or FWO.

References

AHIV17. Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In Thuraisingham et al. [TEMX17], pages 2087–2104.

ARS+15. Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 430–454, Sofia, Bulgaria, April 26–30, 2015. Springer, Heidelberg, Germany.

BDOZ11. Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In Kenneth G. Paterson, editor, *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 169–188, Tallinn, Estonia, May 15–19, 2011. Springer, Heidelberg, Germany.

Bea92. Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 420–432, Santa Barbara, CA, USA, August 11–15, 1992. Springer, Heidelberg, Germany.

BMP13. Joan Boyar, Philip Matthews, and René Peralta. Logic minimization techniques with applications to cryptology. *Journal of Cryptology*, 26(2):280–312, April 2013.

BS97. Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In Burton S. Kaliski Jr., editor, *CRYPTO'97*, volume 1294 of *LNCS*, pages 513–525, Santa Barbara, CA, USA, August 17–21, 1997. Springer, Heidelberg, Germany.

CDG+17. Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. Post-quantum zero-knowledge and signatures from symmetric-key primitives. In Thuraisingham et al. [TEMX17], pages 1825–1842.

CDG+19a. Melissa Chase, David Derler, Steven Goldfeder, Jonathan Katz, Vladimir Kolesnikov, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, Xiao Wang, and Greg Zaverucha. The Picnic Signature Algorithm: Specification version 2.1, 2019. last retrieved Fri May 3, 2019.

CDG+19b. Melissa Chase, David Derler, Steven Goldfeder, Jonathan Katz, Vladimir Kolesnikov, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. The Picnic Signature Scheme, 2019. Submission to NIST Post-Quantum Cryptography project.

DGKN09. Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. Asynchronous multiparty computation: Theory and implementation. In Stanislaw Jarecki and Gene Tsudik, editors, *PKC 2009*, volume 5443 of *LNCS*, pages 160–179, Irvine, CA, USA, March 18–20, 2009. Springer, Heidelberg, Germany.

DK10. Ivan Damgård and Marcel Keller. Secure multiparty AES. In Radu Sion, editor, *FC 2010*, volume 6052 of *LNCS*, pages 367–374, Tenerife, Canary Islands, Spain, January 25–28, 2010. Springer, Heidelberg, Germany.

DKL+12. Ivan Damgård, Marcel Keller, Enrique Larraia, Christian Miles, and Nigel P. Smart. Implementing AES via an actively/covertly secure dishonest-majority MPC protocol. In Ivan Visconti and Roberto De Prisco, editors, *SCN 12*, volume 7485 of *LNCS*, pages 241–263, Amalfi, Italy, September 5–7, 2012. Springer, Heidelberg, Germany.

DPSZ12. Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662, Santa Barbara, CA, USA, August 19–23, 2012. Springer, Heidelberg, Germany.

DR99. Joan Daemen and Vincent Rijmen. AES Proposal: Rijndael, 1999. last retrieved Fri May 3, 2019.

FS87. Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 186–194, Santa Barbara, CA, USA, August 1987. Springer, Heidelberg, Germany.

GMO16. Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. ZKBoo: Faster zero-knowledge for boolean circuits. In Thorsten Holz and Stefan Savage, editors, *USENIX Security 2016*, pages 1069–1083, Austin, TX, USA, August 10–12, 2016. USENIX Association.

IKOS07. Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In David S. Johnson and Uriel Feige, editors, *39th ACM STOC*, pages 21–30, San Diego, CA, USA, June 11–13, 2007. ACM Press.

KKW18. Jonathan Katz, Vladimir Kolesnikov, and Xiao Wang. Improved non-interactive zero knowledge with applications to post-quantum signatures. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 525–537, Toronto, ON, Canada, October 15–19, 2018. ACM Press.

KOR+17. Marcel Keller, Emmanuela Orsini, Dragos Rotaru, Peter Scholl, Eduardo Soria-Vazquez, and Srinivas Vivek. Faster secure multi-party computation of AES and DES using lookup tables. In Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi, editors, *ACNS 17*, volume 10355 of *LNCS*, pages 229–249, Kanazawa, Japan, July 10–12, 2017. Springer, Heidelberg, Germany.

Nat16. National Insitute of Standards and Technology. Post-Quantum Cryptography project, 2016. last retrieved Fri May 3, 2019.

TEMX17. Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors. *ACM CCS 2017*, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press.