The Picnic Signature Scheme

Design Document

Melissa Chase, David Derler, Steven Goldfeder,
Jonathan Katz, Vladimir Kolesnikov, Claudio Orlandi,
Sebastian Ramacher, Christian Rechberger,
Daniel Slamanig, Xiao Wang, Greg Zaverucha

March 30, 2019
Version 2.0

Contents

1 Introduction

Picnic is a signature scheme that is designed to provide security against attacks by quantum computers, in addition to attacks by classical computers. The scheme uses a zero-knowledge proof system and is based on symmetric key primitives like hash functions and block ciphers with conjectured post-quantum security. In particular, Picnic does not rely on number-theoretic or algebraic hardness assumptions.

In this document we present the building blocks of the Picnic signature scheme in Section 2. In Section 3 we present several variants of the Picnic signature scheme and various optimizations to the building blocks. We specify the parameters for some of the building blocks in Section 4 (and leave complete details of the parameters to the specification document). In Section 5 we include the formal security proofs for the proposed instantiations of Picnic. Section 7 presents an analysis of the algorithm with respect to known attacks and Section 8 provides a thorough description of the expected security strength. Finally, Section 9 discuss advantages and limitations, Section 10 discusses additional security properties, and Section 11 presents results on efficiency and memory usage of the Picnic scheme.

Source Materials Parts of this document are taken or adapted from research papers [CDG+17, KKW18, DKP+19] by the authors and the Picnic Specification document [Tea19a]. The Picnic website [Tea19b] lists these sources and related talks, with links to the papers themselves.

1.1 The Picnic Design Team

Picnic was designed collaboratively by the following group of people.

Melissa Chase, Microsoft
David Derler, DFINITY
Steven Goldfeder, Princeton
Jonathan Katz, University of Maryland
Vladimir Kolesnikov, Georgia Tech
Claudio Orlandi, Aarhus University
Sebastian Ramacher, Graz University of Technology
Christian Rechberger, Graz University of Technology & DTU
Daniel Slamanig, AIT Austrian Institute of Technology
Xiao Wang, Northwestern University
Greg Zaverucha, Microsoft

1.2 Acknowledgments

We are grateful for help from Christian Paquin (integration of Picnic in OQS and OpenSSL), Larry Joy (HSM demo), Alexander Grass and Angela Promitzer (contributions to the optimized implementation), Daniel Kales (contributions to the optimized implementation and running benchmarks).

2 Background

In this section we review some background material relevant to the Picnic design.

2.1 Commitments

Definition 2.1 (Commitment Scheme)**.** A (non-interactive) commitment scheme consists of algorithms Com and Open with the following properties:

$\mathsf{Com}(M)$: On input a message $M \in \{0, 1\}$, the commitment algorithm outputs $(C, D) \leftarrow \mathsf{Com}(M; R)$, where R is a random value used when forming the commitment. C is the commitment string, while D is the decommitment string which is kept secret until opening time.

$\mathsf{Open}(C, D)$: On input C, D, the verification algorithm either outputs a message M or $\perp$.

Computationally secure commitments must satisfy the following properties

Correctness. If $\mathsf{Com}(M)$ outputs (C, D) then $\mathsf{Open}(C, D) = M$.

Hiding. For every message pair M, M' the probability ensembles $\{C : (C, D) \leftarrow \mathsf{Com}(M)\}_{\kappa\in\mathbb{N}}$ and $\{C : (C, D) \leftarrow \mathsf{Com}(M')\}_{\kappa\in\mathbb{N}}$ are computationally indistinguishable for security parameter κ.

Binding. We say that an adversary $\mathcal{A}$ wins if it outputs C, D, D' such that $\mathsf{Open}(C, D) = M$, $\mathsf{Open}(C, D') = M'$ and $M \neq M'$. We require that for all efficient algorithms $\mathcal{A}$ (running in time polynomial in κ), the probability that $\mathcal{A}$ wins is a negligible function of κ.

Our implementation uses hash-based commitments, which requires modeling the hash function as a random oracle in our security analysis. Let H be a cryptographic hash function. The commitment scheme works as follows:

$\mathsf{Com}(M):$ Sample $R \overset{R}{\leftarrow} \{0,1\}^\kappa$ and set $C \leftarrow H(R,M)$ and return $(C,(R,M))$;

$\mathsf{Open}(C,D):$ Parse D as (R,M) and return M if $H(R,M)=C$, and return $\perp$ otherwise.

2.2 Zero-Knowledge Proofs and Σ-Protocols

A sigma protocol is a three-flow protocol between a prover and verifier, used to prove knowledge of a secret. A well-known class of sigma protocols are the so-called generalized Schnorr proofs, which allow the prover to prove knowledge of a discrete logarithm, and that it satisfies certain properties. In the present work we use a sigma protocol that allows one to prove knowledge of an input to an arbitrary binary circuit. Sigma protocols are usually zero-knowledge proofs, which informally means that the proof protocol does not reveal any information about the secret. We describe interactive protocols, but will show later how to make them non-interactive (so that signatures are non-interactive). Let L be an **NP**-language with associated witness relation R so that $L = \{x \mid \exists w : R(x,w)=1\}$. A Σ-protocol for language L is defined as follows.

Definition 2.2 (Σ-Protocol)**.** A Σ-protocol for relation R is an interactive three-move protocol between a PPT prover $\mathsf{P} = (\mathsf{Commit}, \mathsf{Prove})$ and a PPT verifier $\mathsf{V} = (\mathsf{Challenge}, \mathsf{Verify})$, where P makes the first move and transcripts are of the form $(\mathsf{a}, \mathsf{e}, \mathsf{z}) \in \mathsf{A} \times \mathsf{E} \times \mathsf{Z}$, where a is output by Commit, e is output by $\mathsf{Challenge}$ and z is output by Prove. Additionally, Σ protocols satisfy the following properties

Completeness. A Σ-protocol for language L is complete, if for all security parameters κ, and for all $(x,w) \in R$, it holds that

$$\Pr[\langle \mathsf{P}(1^\kappa, x, w), \mathsf{V}(1^\kappa, x)\rangle = 1] = 1.$$

***s*-Special Soundness.** A Σ-protocol for language L is s-special sound, if there exists a PPT extractor E so that for all x, and for all sets of accepting transcripts $\{(\mathsf{a}, \mathsf{e}_i, \mathsf{z}_i)\}_{i \in [s]}$ with respect to x where $\forall i, j \in$

$[s], i \neq j : \mathsf{e}_i \neq \mathsf{e}_j$, generated by any algorithm with polynomial runtime in κ, it holds that

$$\Pr\left[\ w \leftarrow \mathsf{E}(1^\kappa, x, \{(\mathsf{a}, \mathsf{e}_i, \mathsf{z}_i)\}_{i\in[s]}) \quad : \quad (x, w) \in R\ \right] \geq 1 - \epsilon(\kappa).$$

We can also consider a computational variant which says that if $\mathcal{A}$ is a PPT algorithm then the probability that it can produce an accepting transcript from which E fails to extract a valid witness is negligible.

Special Honest-Verifier Zero-Knowledge. A Σ-protocol is special honest-verifier zero-knowledge, if there exists a PPT simulator S so that for every $x \in L$ and every challenge e from the challenge space, it holds that a transcript $(\mathsf{a}, \mathsf{e}, \mathsf{z})$, where $(\mathsf{a}, \mathsf{z}) \leftarrow \mathsf{S}(1^\kappa, x, \mathsf{e})$ is computationally indistinguishable from a transcript resulting from an honest execution of the protocol.

The s-special soundness property gives an immediate bound for the soundness of the protocol: if no witness exists then (ignoring a negligible error) the prover can successfully answer at most $(s-1)/t$ of the possible challenges, where $t = |\mathsf{E}|$ is the size of the challenge space. If this value is too large, it is possible to reduce the soundness error using parallel repetition (see [Dam10, CDS94] for details).

Lemma 2.3. *Let Π be a Σ-protocol with s-special soundness. Then the ℓ-fold parallel repetition of Π, denoted Π^ℓ, is also a Σ-protocol with s^ℓ-special soundness.*

2.3 Non-Interactive Zero-Knowledge Proofs of Knowledge

For our signatures, we use non-interactive zero-knowledge proofs of knowledge, in which the proof is a single message. Here we define zero-knowledge and the necessary notion of simulation-extractability. We present the definition the random oracle model against classical adversaries and the definition in the quantum random oracle model against quantum adversaries.[1]

Zero-knowledge says that there is a simulator which can produce proofs that are indistinguishable from those produced by P without knowing the

[1] These definition roughly combine the ROM definitions of [BPW12] and the QROM definitions of [Unr15].

witnesses to an adversary who is given access to a simulated version of the random oracle.

Definition 2.4 (Zero-Knowledge)**.** A protocol P, V for relation R is zero-knowledge against (quantum) adversaries in the (quantum) random oracle model if there exist PPT algorithms $\mathsf{S_R}, \mathsf{Sim}$ such that for all (quantum) PPT adversaries $\mathcal{A}$

$$|\Pr[b \leftarrow \mathcal{A}^{R(\cdot),P(\cdot,\cdot)}(1^\lambda) : b = 1] - \Pr[b \leftarrow \mathcal{A}^{\mathsf{S_R}(\cdot),\mathsf{Sim}_P(\cdot,\cdot)}(1^\lambda) : b = 1]|$$

is negligible in λ, where R is a random function to which $\mathcal{A}$ can provide (quantum) inputs, Sim_P takes a pair $(x, w) \in R$ and calls $\mathsf{Sim}(x)$, and $\mathsf{Sim}, \mathsf{S_R}$ share state.

Simulation-extractability says that an adversary cannot produce a new proof for a statement for which he does not know the witness, even if he is allowed to see proofs produced by someone else for statements of his choice. More formally, we say that even when an adversary is given access to the zero-knowledge simulator to obtain proofs for (true or false) statements of its choice, whenever it produces a new proof (not produced by the simulator) that is accepted by the verifier, there is an extractor that can look at the implementation of the adversary and extract a valid witness for that statement.

Definition 2.5 (Simulation extractability)**.** A protocol P, V for relation R satisfies simulation extractability against (quantum) adversaries in the (quantum) random oracle model if there exist PPT algorithms $\mathsf{S_R}, \mathsf{Sim}$ satisfying the zero-knowledge definition and a PPT (quantum) extractor such that for all (quantum) PPT adversaries $\mathcal{A}$

$$\Pr[(x, \pi) \leftarrow \mathcal{A}^{\mathsf{S_R}(\cdot),\mathsf{Sim}(\cdot)}(1^\lambda);\ w \leftarrow E(\mathcal{A}, x, \pi) : \\ V^{\mathsf{S_R}}(x, \pi) = 1 \wedge (x, \pi) \notin \mathcal{Q} \wedge (x, w) \notin R]$$

is negligible in λ, where $\mathsf{S_R}, \mathsf{Sim}$, and E share state, $\mathcal{Q}$ is the list of $\mathcal{A}$'s queries to Sim and the resulting responses, and passing $\mathcal{A}$ as input to E means E is given access to a (quantum) implementation of $\mathcal{A}$.

2.4 Signature Schemes

In the following we recall a standard definition of signature schemes along with two widely used security notions.

Definition 2.6 (Signature Scheme)**.** A signature scheme Σ is a triple $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ of PPT algorithms, which are defined as follows:

$\mathsf{Gen}(1^\kappa)$: This algorithm takes a security parameter κ as input and outputs a secret (signing) key sk and a public (verification) key pk with associated message space $\mathcal{M}$ (we may omit to make the message space $\mathcal{M}$ explicit).

$\mathsf{Sign}(\mathsf{sk}, m)$: This algorithm takes a secret key sk and a message $m \in \mathcal{M}$ as input and outputs a signature σ.

$\mathsf{Verify}(\mathsf{pk}, m, \sigma)$: This algorithm takes a public key pk, a message $m \in \mathcal{M}$ and a signature σ as input and outputs a bit $b \in \{0, 1\}$.

Besides the usual correctness property, Σ needs to provide some unforgeability notion. We consider two notions, namely existential unforgeability under adaptively chosen message attacks ($\mathsf{EUF\text{-}CMA}$ security) and its strong variant ($\mathsf{sEUF\text{-}CMA}$ security), which we define below.

Definition 2.7 ($\mathsf{EUF\text{-}CMA}$)**.** A signature scheme Σ is $\mathsf{EUF\text{-}CMA}$ secure, if for all PPT adversaries $\mathcal{A}$ there is a negligible function $\varepsilon(\cdot)$ such that

$$\Pr\Big[\ (\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^\kappa),\ (m^*, \sigma^*) \leftarrow \mathcal{A}^{\mathsf{Sign}(\mathsf{sk}, \cdot)}(\mathsf{pk}) \ : \\ \mathsf{Verify}(\mathsf{pk}, m^*, \sigma^*) = 1 \ \wedge \ (m^*, \cdot) \notin \mathcal{Q}^{\mathsf{Sign}}\ \Big] \leq \varepsilon(\kappa),$$

where the environment keeps track of the queries and responses to and from the signing oracle via $\mathcal{Q}^{\mathsf{Sign}}$.

Definition 2.8 ($\mathsf{sEUF\text{-}CMA}$)**.** A signature scheme Σ is strongly $\mathsf{EUF\text{-}CMA}$ ($\mathsf{sEUF\text{-}CMA}$) secure, if for all PPT adversaries $\mathcal{A}$ there is a negligible function $\varepsilon(\cdot)$ such that

$$\Pr\Big[\ (\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^\kappa),\ (m^*, \sigma^*) \leftarrow \mathcal{A}^{\mathsf{Sign}(\mathsf{sk}, \cdot)}(\mathsf{pk}) \ : \\ \mathsf{Verify}(\mathsf{pk}, m^*, \sigma^*) = 1 \ \wedge \ (m^*, \sigma^*) \notin \mathcal{Q}^{\mathsf{Sign}}\ \Big] \leq \varepsilon(\kappa),$$

where the environment keeps track of the queries and responses to and from the signing oracle via $\mathcal{Q}^{\mathsf{Sign}}$.

2.5 Fiat-Shamir Transform

The Fiat-Shamir (FS) transform [FS86] is an elegant way to construct signature schemes from Σ-protocols. The basic idea is similar to constructing NIZK proofs from Σ-protocols, but the challenge e is generated by hashing the prover's first message a and the message m to be signed, i.e., define a modified challenge algorithm $\mathsf{Challenge}'$ that outputs $e \leftarrow H(\mathsf{a}, m)$. Then, the prover can locally obtain the challenge after computing the initial message. Starting a verifier $\mathsf{V}' = (\mathsf{Challenge}', \mathsf{Verify})$ on the same initial message would yield the same challenge. The prover outputs (a, z) as the challenge.

More formally, by using the hash function $H : \mathsf{A} \times \mathsf{X} \rightarrow \mathsf{E}$, which we model as a random oracle, we obtain the non-interactive PPT algorithms $(\mathtt{Prove}_H, \mathtt{Verify}_H)$, defined as follows:

$\mathtt{Prove}_H(1^\kappa, (x, m), w)$: Start P on input $(1^\kappa, x, w)$, obtain the first message a, answer with $\mathsf{e} \leftarrow H(\mathsf{a}, m)$, and finally obtain z. Return $\pi \leftarrow (\mathsf{a}, \mathsf{z})$.

$\mathtt{Verify}_H(1^\kappa, (x, m), \pi)$: Parse π as (a, z). Start V' on $(1^\kappa, x)$, send (a, m) as the first message to the verifier. When V' outputs e, reply with z and output 1 if V' accepts and 0 otherwise.

2.6 Unruh Transform

Similar to the Fiat-Shamir transform, Unruh's transform [Unr12, Unr15, Unr16] allows one to construct NIZK proofs and signature schemes from Σ-protocols. In contrast to the FS transform, Unruh's transform can be proven secure in the QROM (quantum random oracle model), strengthening the security guarantee against quantum adversaries.

At a high level, Unruh's transform works as follows: Given a 2-special-sound Σ-protocol, integers t and M, a statement x and a random permutation G, the prover will repeat the first phase of the Σ-protocol t times. Then, for each of the t runs, it produces proofs to M different randomly selected challenges. The prover applies G to each of the so-obtained responses. The prover then selects the responses to publish for each round of the Σ-protocol by querying the random oracle on the message to be signed, all first rounds of the Σ-protocol and the outputs of G on all responses.

For a more formal treatment, we keep t as above and let $M \in [2, |\mathsf{E}|]$. Let $H : \{0,1\}^* \rightarrow [M]^t$ be a hash function we model as a random oracle.

We obtain the non-interactive PPT algorithms ($\texttt{Prove}_H, \texttt{Verify}_H$) defined as follows:

$\texttt{Prove}_H(1^\kappa, (x, m), w)$:

1. For $i \in [t]$:
   (a) Start P on $(1^\kappa, x, w)$ and obtain first message a_i.
   (b) For $j \in [M]$, set $\mathsf{e}_{i,j} \overset{R}{\leftarrow} \mathsf{E} \backslash \{\mathsf{e}_{i,1}, \dots, \mathsf{e}_{i,j-1}\}$ and obtain response $\mathsf{z}_{i,j}$ for challenge $\mathsf{e}_{i,j}$.
2. For $i, j \in [t] \times [M]$, set $g_{i,j} \leftarrow G(\mathsf{z}_{i,j})$.
3. Let $(J_1, \dots, J_t) \leftarrow H(m, (\mathsf{a}_i)_{i \in [t]}, (\mathsf{e}_{i,j})_{(i,j) \in [t] \times [M]}, (g_{i,j})_{(i,j) \in [t] \times [M]})$
4. Return $\pi \leftarrow ((\mathsf{a}_i)_{i \in [t]}, (\mathsf{e}_{i,j})_{(i,j) \in [t] \times [M]}, (g_{i,j})_{(i,j) \in [t] \times [M]}, (\mathsf{z}_{i,J_i})_{i \in [t]})$

$\texttt{Verify}_H(1^\kappa, (x, m), \pi)$: Parse π as $((\mathsf{a}_i)_{i \in [t]}, (\mathsf{e}_{i,j})_{(i,j) \in [t] \times [M]}, (g_{i,j})_{(i,j) \in [t] \times [M]}, (\mathsf{z}_i)_{i \in [t]})$.

1. Let $(J_1, \dots, J_t) \leftarrow H(m, (\mathsf{a}_i)_{i \in [t]}, (\mathsf{e}_{i,j})_{(i,j) \in [t] \times [M]}, (g_{i,j})_{(i,j) \in [t] \times [M]})$
2. For $i \in [t]$ check that all $\mathsf{e}_{i,1}, \dots, \mathsf{e}_{i,M}$ are pairwise distinct.
3. For $i \in [t]$ check whether V accepts the proof with respect to x, first message a_i, challenge e_{i,J_i} and response z_i.
4. For $i \in [t]$ check $g_{i,J_i} = G(\mathsf{z}_i)$.
5. Output 1 if all checks succeeded and 0 otherwise.

We discuss a specialization of Unruh's transform to our Σ-protocol in Section 3.1.

2.7 (2,3)-Decomposition of Circuits

A circuit decomposition is a protocol for jointly computing a circuit, similar to an MPC protocol, but with greater efficiency. In a (2,3)-decomposition there are three players and the protocol has 2-privacy, i.e., it remains secure even if two of the three players are corrupted.

Definition 2.9 ((2,3)-decomposition)**.** Let $f(\cdot)$ be a function that is computed by an n-gate circuit ϕ such that $f(x) = \phi(x) = y$. Let k_1, k_2, and k_3 be tapes of length κ chosen uniformly at random from $\{0,1\}^\kappa$ corresponding to players P_1, P_2 and P_3, respectively. Consider the following set of functions, $\mathcal{D}$:

$$\begin{aligned}
(\mathsf{view}_1^{(0)},\ \mathsf{view}_2^{(0)},\ \mathsf{view}_3^{(0)}) &\leftarrow \mathtt{Share}(x, k_1, k_2, k_3) \\
\mathsf{view}_i^{(j+1)} &\leftarrow \mathtt{Update}(\mathsf{view}_i^{(j)}, \mathsf{view}_{i+1}^{(j)}, k_i, k_{i+1}) \\
y_i &\leftarrow \mathtt{Output}(\mathsf{View}_i) \\
y &\leftarrow \mathtt{Reconstruct}(y_1, y_2, y_3)
\end{aligned}$$

such that `Share` is a potentially randomized function that takes x as input and outputs the initial view for each player containing the secret share of x_i of x - i.e. $\mathsf{view}_i^{(0)} = x_i$. The function `Update` computes the wire values for the next gate and updates the view accordingly. The function $\mathtt{Output}_i$ takes as input the final view, $\mathsf{View}_i \equiv \mathsf{view}_i^{(n)}$ after all gates have been computed and outputs player P_i's *output share*, denoted y_i.

Correctness requires that reconstructing a (2,3)-decomposed evaluation of a circuit ϕ yields the same value as directly evaluating ϕ on the input value. The 2-privacy property requires that revealing the values from two shares reveals nothing about the input value. More formally, these two properties are defined as follows: We define the experiment $\mathsf{EXP}_{\mathsf{decomp}}^{(\phi,\mathsf{x})}$ in Experiment 1, which runs the decomposition over a circuit ϕ on input x: We say that $\mathcal{D}$ is a $(2,3)$-*decomposition* of ϕ if the following two properties hold when running $\mathsf{EXP}_{\mathsf{decomp}}^{(\phi,\mathsf{x})}$:

$\mathsf{EXP}_{\mathsf{decomp}}^{(\phi,\mathsf{x})}$:

1. First run the Share function on x: $\mathsf{view}_1^{(0)}, \mathsf{view}_2^{(0)}, \mathsf{view}_3^{(0)} \leftarrow \mathtt{Share}(x, k_1, k_2, k_3)$
2. For each of the three views, call the update function successively for every gate in the circuit: $\mathsf{view}_i^{(j)} = \mathtt{Update}(\mathsf{view}_i^{(j-1)}, \mathsf{view}_{i+1}^{(j-1)}, k_i, k_{i+1})$ for $i \in [1,3]$, $j \in [1,n]$
3. From the final views, compute the output share of each view: $y_i \leftarrow \mathtt{output}(\mathsf{View}_i)$

Experiment 1: Decomposition Experiment

Correctness. For all circuits ϕ, for all inputs x and for the y_i's produced by $\mathsf{EXP}^{(\phi,\mathsf{x})}_{\mathsf{decomp}}$,

$$\Pr[\phi(x) = \mathtt{Reconstruct}(y_1, y_2, y_3)] = 1.$$

2-Privacy. Let $\mathcal{D}$ be correct. Then for all $e \in \{1, 2, 3\}$ there exists a PPT simulator $\mathcal{S}_e$ such that for any probabilistic polynomial-time (PPT) algorithm $\mathcal{A}$, for all circuits ϕ, for all inputs x, and for the distribution of views and k_i's produced by $\mathsf{EXP}^{(\phi,\mathsf{x})}_{\mathsf{decomp}}$ we have that

$$\left| \Pr[\mathcal{A}(x, y, k_e, \mathsf{View}_e, k_{e+1}, \mathsf{View}_{e+1}, y_{e+2}) = 1] - \Pr[\mathcal{A}(x, y, \mathcal{S}_e(\phi, y)) = 1] \right|$$

is negligible.

We now discuss the (2,3)-decomposition used by ZKB++. Let R be an arbitrary finite ring and ϕ a function such that $\phi : R^m \to R^\ell$ can be expressed by an n-gate arithmetic circuit over the ring using addition by constant, multiplication by constant, addition and multiplication gates. A $(2, 3)-$decomposition of ϕ is given by the following functions. In the notation below, arithmetic operations are done in R^s where the operands are elements of R^s):

- $(x_1, x_2, x_3) \leftarrow \mathtt{Share}(x, k_1, k_2, k_3)$ samples random $x_1, x_2, x_3 \in R^m$ such that $x_1 + x_2 + x_3 = x$.
- $y_i \leftarrow \mathtt{Output}_i(\mathsf{view}_i^{(n)})$ selects the ℓ output wires of the circuit as stored in the view $\mathsf{view}_i^{(n)}$.
- $y \leftarrow \mathtt{Reconstruct}(y_1, y_2, y_3) = y_1 + y_2 + y_3$
- $\mathsf{view}_i^{(j+1)} \leftarrow \mathtt{Update}_i^{(j)}(\mathsf{view}_i^{(j)}, \mathsf{view}_{i+1}^{(j)}, k_i, k_{i+1})$ computes P_i's view of the output wire of gate g_j and appends it to the view. Notice that it takes as input the views and random tapes of both party P_i as well as party P_{i+1}. We use w_k to refer to the k-th wire, and we use $w_k^{(i)}$ to refer to the value of w_k in party P_i's view. The update operation depends on the type of gate g_j.

The gate-specific operations are defined as follows.

Addition by Constant ($w_b = w_a + k$):

$$w_b^{(i)} = \begin{cases} w_a^{(i)} + k & \text{if } i = 1, \\ w_a^{(i)} & \text{otherwise.} \end{cases}$$

Multiplication by Constant ($w_b = w_a \cdot k$):

$$w_b^{(i)} = k \cdot w_a^{(i)}$$

Binary Addition ($w_c = w_a + w_b$):

$$w_c^{(i)} = w_a^{(i)} + w_b^{(i)}$$

Binary Multiplication ($w_c = w_a \cdot w_b$):

$$\begin{aligned} w_c^{(i)} \;=\; & w_a^{(i)} \cdot w_b^{(i)} \quad + w_a^{(i+1)} \cdot w_b^{(i)} \quad + \\ & w_a^{(i)} \cdot w_b^{(i+1)} + R_i(c) - R_{i+1}(c), \end{aligned}$$

where $R_i(c)$ is the c-th output of a pseudorandom generator seeded with k_i.

Note that with the exception of the constant addition gate, the gates are symmetric for all players. Also note that P_i can compute all gate types locally with the exception of binary multiplication gates as this requires inputs from P_{i+1}. In other words, for every operation except binary multiplication, the `Update` function does not use the inputs from the second party, i.e., $\mathsf{view}_{i+1}^{(j)}$ and k_{i+1}.

While we do not give the details here, [GMO16a] shows that this decomposition meets the correctness and 2-privacy requirements of Definition 2.9. In other words, for every operation except binary multiplication, the `Update` function does not use the inputs from the second party, i.e., $\mathsf{view}_{i+1}^{(j)}$ and R_{i+1}.

2.8 ZKB++

ZKB++, an optimized version of ZKBoo [GMO16a], is a proof system for zero-knowledge proofs on arbitrary circuits. ZKBoo and ZKB++ build on the MPC-in-the-head paradigm of Ishai *et al.* [IKOS09], that we describe only informally here. The multiparty computation protocol (MPC) will implement the relation, and the input is the witness. For example, the MPC could compute $y = \text{SHA-256}(x)$ where players each have a share of x and y is public. The idea is to have the prover simulate a multiparty computation protocol "in their head", commit to the state and transcripts of all players, then have the verifier "corrupt" a random subset of the simulated players by seeing their complete state. The verifier then checks that the corrupted players performed the correct computation, and if so, he has some assurance that the output is correct. Iterating this for many rounds then gives the verifier high assurance.

ZKBoo generalizes the idea of [IKOS09] by replacing MPC with circuit decompositions. In Scheme 1 and Scheme 2 we present the prover and the verifier of the ZKB++ Σ-protocol.

```
P.Commit :  1. For each iteration $i \in [t]$: Sample random seeds $k_1^{(i)}, k_2^{(i)}, k_3^{(i)}$
               obtain view $\mathsf{View}_j^{(i)}$ and output share $y_j^{(i)}$. For each player $P_j$
               compute
               (a) $(x_1^{(i)}, x_2^{(i)}, x_3^{(i)}) \leftarrow \mathtt{Share}(x, k_1^{(i)}, k_2^{(i)}, k_3^{(i)})$
               (b) $\mathsf{View}_j^{(i)} \leftarrow \mathtt{Update}(\ldots \mathtt{Update}(x_j^{(i)}, x_{j+1}^{(i)}, k_j^{(i)}, k_{j+1}^{(i)}) \ldots)$
               (c) $y_j^{(i)} \leftarrow \mathtt{Output}(\mathsf{View}_j^{(i)})$
               (d) Commit $C_j^{(i)} \leftarrow \mathsf{Com}(k_j^{(i)}, x_j^{(i)}, \mathsf{View}_j^{(i)}, y_j^{(i)})$, and let $\mathsf{a}^{(i)} \leftarrow$
                   $(y_1^{(i)}, y_2^{(i)}, y_3^{(i)}, C_1^{(i)}, C_2^{(i)}, C_3^{(i)})$.
            2. Return $(\mathsf{a}^{(i)})_{i \in [t]}$.

P.Prove :  On input of a challenge $(\mathsf{e}^{(i)})_{i \in [t]}$, set for each iteration $i \in [1, t]$

   $$\mathsf{z}^{(i)} \leftarrow \begin{cases} (\mathsf{View}_2^{(i)}, k_1^{(i)}, k_2^{(i)}) & \text{if } \mathsf{e}^{(i)} = 1, \\ (\mathsf{View}_3^{(i)}, k_2^{(i)}, k_3^{(i)}, x_3^{(i)}) & \text{if } \mathsf{e}^{(i)} = 2, \\ (\mathsf{View}_1^{(i)}, k_3^{(i)}, k_1^{(i)}, x_3^{(i)}) & \text{if } \mathsf{e}^{(i)} = 3. \end{cases}$$

   and return $(\mathsf{z}^{(i)})_{i \in [t]}$.
```

Scheme 1: The prover of the ZKB++ Σ-protocol.

We now discuss various instantiation aspects of ZKB++ that make the

$\mathsf{V.Challenge}$: Store $(\mathsf{a}^{(i)})_{i\in[t]}$ and return $(\mathsf{e}^{(i)})_{i\in[t]} \xleftarrow{R} \mathsf{E}^t$.

$\mathsf{V.Verify}$:

1. For each iteration $i \in [t]$ reconstruct the views, input and output shares that were not explicitly given as part of the proof response $\mathsf{z}^{(i)}$:

   (a) Set

$$x^{(i)}_{\mathsf{e}^{(i)}} \leftarrow \begin{cases} R_{\mathsf{e}^{(i)}}(0) & \text{if } \mathsf{e}^{(i)} \neq 3, \\ x^{(i)}_3 \text{ given as part of } \mathsf{z}^{(i)} & \text{if } \mathsf{e}^{(i)} = 3. \end{cases}$$

$$x^{(i)}_{\mathsf{e}^{(i)}+1} \leftarrow \begin{cases} R_{\mathsf{e}^{(i)}+1}(0) & \text{if } \mathsf{e}^{(i)} \neq 2, \\ x^{(i)}_3 \text{ given as part of } \mathsf{z}^{(i)} & \text{if } \mathsf{e}^{(i)} = 2. \end{cases}$$

   (b) Obtain $\mathsf{View}^{(i)}_{\mathsf{e}^{(i)}+1}$ from $\mathsf{z}^{(i)}$.

   (c) $\mathsf{View}^{(i)}_{e} \leftarrow \texttt{Update}(\ldots\texttt{Update}(x^{(i)}_{\mathsf{e}^{(i)}}, x^{(i)}_{e+1}, k^{(i)}_e, k^{(i)}_{e+1})\ldots)$

   (d) $y^{(i)}_{\mathsf{e}^{(i)}} \leftarrow \texttt{Output}(\mathsf{View}^{(i)}_{\mathsf{e}^{(i)}})$

   (e) $y^{(i)}_{\mathsf{e}^{(i)}+1} \leftarrow \texttt{Output}(\mathsf{View}^{(i)}_{\mathsf{e}^{(i)}+1})$

   (f) $y^{(i)}_{\mathsf{e}^{(i)}+2} \leftarrow y - y^{(i)}_{\mathsf{e}^{(i)}} - y^{(i)}_{\mathsf{e}^{(i)}+1}$

2. Re-compute the commitments for views $\mathsf{View}^{(i)}_{\mathsf{e}^{(i)}}$ and $\mathsf{View}^{(i)}_{\mathsf{e}^{(i)}}$. For $j \in \{\mathsf{e}^{(i)}, \mathsf{e}^{(i)}+1\}$:

$$C^{(i)}_j \leftarrow \mathsf{Com}(k^{(i)}_j, x^{(i)}_j, \mathsf{View}^{(i)}_j, y^{(i)}_j)$$

3. Set $\mathsf{a}'^{(i)} \leftarrow (y^{(i)}_1, y^{(i)}_2, y^{(i)}_3, C^{(i)}_1, C^{(i)}_2, C^{(i)}_3)$ taking $C^{(i)}_{\mathsf{e}^{(i)}+2}$ from $\mathsf{a}^{(i)}$.

4. If $\mathsf{a}'^{(i)} = \mathsf{a}^{(i)}$ for all $i \in [t]$, output Accept, otherwise Reject.

Scheme 2: The verifier of the ZKB++ Σ-protocol.

optimizations with respect to ZKBoo possible. To highlight the difference, we also present the Fiat-Shamir transformed ZKBoo proof system in Scheme 3 and the Fiat-Shamir transformed ZKB++ in Scheme 4.

The Share Function. We make the `Share` function sample the shares pseudorandomly as:

$$\begin{aligned}(x_1, x_2, x_3) \leftarrow \texttt{Share}(x, k_1, k_2, k_3) &:= \\ &x_1 = R_1(0),\ x_2 = R_2(0),\ x_3 = x - x_1 - x_2.\end{aligned}$$

R_i is a pseudorandom generator seeded with k_i. We specify the `Share` function in this manner as it will lead to more compact proofs. Moving now to the ZKBoo protocol, for each round, the prover is required to "open" two views. In order to verify the proof, the verifier must be given both the random tape and the input share for each opened view. If these values are generated independently of one another, then the prover will have to explicitly include both of them in the proof. However, with our sampling method, in View_1 and View_2, the prover only needs to include k_i, as x_i can be deterministically computed by the verifier.

The exact savings depends on which views the prover must open, and thus depends on the challenge.

Not Including Input Shares. Since the input shares are generated pseudorandomly using the seed k_i, we do not need to include them in the view when $\mathsf{e} = 1$. However, if $\mathsf{e} = 2$ or $\mathsf{e} = 3$, we still need to send one input share for the third view for which the input share cannot be derived from the seed. Thus we explicitly specify the input share when required and do not include it in $\mathsf{View}_i^{(j)}$.

No Additional Randomness for Commitments. Since the first input to the commitment is the seed value k_i for the random tape, the protocol input to the commitment doubles as a randomization value, ensuring that commitments are hiding. To simplify security analysis, we in fact choose two different random oracles H', H''. We use $H'(k_i)$ as the seed to generate the random tape used to generate the input shares and views, and we use $H''(k_i)$ as input to the commitment. In the random oracle model then, this produces two independent random values; as $H''(k_i)$ for the unopened view only appears as input to the commitment, this effectively replaces the randomness needed for the commitment scheme in the RO model. (Since one already needs the RO model to make the proofs non-interactive, there is no extra

$\mathsf{Prove}_H(1^\kappa, y, x)$:

1. For each iteration $i \in [1,t]$: Sample random tapes $k_1^{(i)}, k_2^{(i)}, k_3^{(i)}$ and run the decomposition to get an output view $\mathsf{View}_j^{(i)}$ and output share $y_j^{(i)}$. In particular, for each player P_j:
   (a) $(x_1^{(i)}, x_2^{(i)}, x_3^{(i)}) \leftarrow \mathtt{Share}(x, k_1^{(i)}, k_2^{(i)}, k_3^{(i)})$
   (b) $\mathsf{View}_j^{(i)} \leftarrow \mathtt{Update}(\ldots \mathtt{Update}(x_j^{(i)}, x_{j+1}^{(i)}, k_j^{(i)}, k_{j+1}^{(i)}) \ldots)$
   (c) $y_j^{(i)} \leftarrow \mathtt{Output}(\mathsf{View}_j^{(i)})$
   (d) Commit $C_j^{(i)} \leftarrow \mathsf{Com}(k_j^{(i)}, \mathsf{View}_j^{(i)})$, and let $\mathsf{a}^{(i)} \leftarrow (y_1^{(i)}, y_2^{(i)}, y_3^{(i)}, C_1^{(i)}, C_2^{(i)}, C_3^{(i)})$.
2. Compute the challenge: $\mathsf{e} \leftarrow H(\mathsf{a}^{(1)}, \ldots, \mathsf{a}^{(t)})$. Interpret the challenge such that for $i \in [1,t]$, $\mathsf{e}^{(i)} \in \{1,2,3\}$
3. For each iteration $i \in [1,t]$, let $\mathsf{z}^{(i)} = (D_{\mathsf{e}^{(i)}}^{(i)}, D_{\mathsf{e}^{(i)}+1}^{(i)})$.
4. Output $\pi \leftarrow [(\mathsf{a}^{(1)}, \mathsf{z}^{(1)}), (\mathsf{a}^{(2)}, \mathsf{z}^{(2)}), \cdots, (\mathsf{a}^{(t)}, \mathsf{z}^{(t)})]$

$\mathsf{Verify}_H(1^\kappa, y, \pi)$:

1. Parse π as $[(\mathsf{a}^{(1)}, \mathsf{z}^{(1)}), (\mathsf{a}^{(2)}, \mathsf{z}^{(2)}), \cdots, (\mathsf{a}^{(t)}, \mathsf{z}^{(t)})]$.
2. Compute the challenge: $\mathsf{e}' \leftarrow H(\mathsf{a}^{(1)}, \cdots, \mathsf{a}^{(t)})$. Interpret the challenge such that for $i \in [1,t]$, $\mathsf{e}'^{(i)} \in \{1,2,3\}$.
3. For each iteration $i \in [1,t]$: If there exists $j \in \{\mathsf{e}'^{(i)}, \mathsf{e}'^{(i)}+1\}$ such that $\mathsf{Open}(C_j^{(i)}, D_j^{(i)}) = \perp$, output Reject. Otherwise, for all $j \in \{\mathsf{e}'^{(i)}, \mathsf{e}'^{(i)}+1\}$, set $\{k_j^{(i)}, \mathsf{View}_j^{(i)}\} \leftarrow \mathsf{Open}(C_j^{(i)}, D_j^{(i)})$.
4. For each iteration $i \in [1,t]$: If $\mathtt{Reconstruct}(y_1^{(i)}, y_2^{(i)}, y_3^{(i)}) \neq y$, output Reject. If there exists $j \in \{\mathsf{e}'^{(i)}, \mathsf{e}'^{(i)}+1\}$ such that $y_j^{(i)} \neq \mathtt{Output}(\mathsf{View}_j^{(i)})$, output Reject. For each wire value $w_j^{(\mathsf{e})} \in \mathsf{View}_e$, if $w_j^{(\mathsf{e})} \neq \mathtt{Update}(\mathsf{view}_{\mathsf{e}}^{(j-1)}, \mathsf{view}_{\mathsf{e}+1}^{(j-1)}, k_{\mathsf{e}}, k_{\mathsf{e}+1})$ output Reject.
5. Output Accept.

Scheme 3: The ZKBoo non-interactive proof system.

$\mathsf{Prove}_H(1^\kappa, y, x)$:

1. For each iteration $i \in [t]$: Sample random tapes $k_1^{(i)}, k_2^{(i)}, k_3^{(i)}$ and obtain output view $\mathsf{View}_j^{(i)}$ and output share $y_j^{(i)}$. For each player P_j compute
   (a) $(x_1^{(i)}, x_2^{(i)}, x_3^{(i)}) \leftarrow \mathtt{Share}(x, k_1^{(i)}, k_2^{(i)}, k_3^{(i)})$
   (b) $\mathsf{View}_j^{(i)} \leftarrow \mathtt{Update}(\ldots \mathtt{Update}(x_j^{(i)}, x_{j+1}^{(i)}, k_j^{(i)}, k_{j+1}^{(i)}) \ldots)$
   (c) $y_j^{(i)} \leftarrow \mathtt{Output}(\mathsf{View}_j^{(i)})$
   (d) Commit $C_j^{(i)} \leftarrow \mathsf{Com}(k_j^{(i)}, x_j^{(i)}, \mathsf{View}_j^{(i)}, y_j^{(i)})$, and let $\mathsf{a}^{(i)} \leftarrow (y_1^{(i)}, y_2^{(i)}, y_3^{(i)}, C_1^{(i)}, C_2^{(i)}, C_3^{(i)})$.
2. Compute the challenge: $\mathsf{e} \leftarrow H(\mathsf{a}^{(1)}, \ldots, \mathsf{a}^{(t)})$.
3. For each iteration $i \in [1, t]$ set: $\mathsf{z}^{(i)} \leftarrow (\mathsf{View}_2^{(i)}, k_1^{(i)}, k_2^{(i)})$ if $\mathsf{e}^{(i)} = 1$ and $\mathsf{z}^{(i)} \leftarrow (\mathsf{View}_{\mathsf{e}^{(i)}+1}^{(i)}, k_{\mathsf{e}^{(i)}}^{(i)}, k_{\mathsf{e}^{(i)}+1}^{(i)}, x_3^{(i)})$ otherwise, and return $\pi \leftarrow (\mathsf{e}, \mathsf{z}_{i \in [t]}^{(i)})$.

$\mathsf{Verify}_H(1^\kappa, y, \pi)$: Parse π as $(\mathsf{e}, \mathsf{z}_{i \in [t]}^{(i)})$.

1. For each iteration $i \in [t]$ reconstruct the views, input and output shares that were not explicitly given as part of the proof $\mathsf{z}^{(i)}$:
   (a) Set $x_{\mathsf{e}^{(i)}}^{(i)} \leftarrow R_{\mathsf{e}^{(i)}}(0)$ if $\mathsf{e}^{(i)} \neq 3$, otherwise obtain $x_3^{(i)}$ from $\mathsf{z}^{(i)}$. Set $x_{\mathsf{e}^{(i)}+1}^{(i)} \leftarrow R_{\mathsf{e}^{(i)}+1}(0)$ if $\mathsf{e}^{(i)} \neq 2$, otherwise obtain $x_3^{(i)}$ from $\mathsf{z}^{(i)}$.
   (b) Obtain $\mathsf{View}_{\mathsf{e}^{(i)}+1}^{(i)}$ from $\mathsf{z}^{(i)}$.
   (c) $\mathsf{View}_e^{(i)} \leftarrow \mathtt{Update}(\ldots \mathtt{Update}(x_{\mathsf{e}^{(i)}}^{(i)}, x_{e+1}^{(i)}, k_e^{(i)}, k_{e+1}^{(i)}) \ldots)$
   (d) $y_{\mathsf{e}^{(i)}}^{(i)} \leftarrow \mathtt{Output}(\mathsf{View}_{\mathsf{e}^{(i)}}^{(i)})$, $y_{\mathsf{e}^{(i)}+1}^{(i)} \leftarrow \mathtt{Output}(\mathsf{View}_{\mathsf{e}^{(i)}+1}^{(i)})$
   (e) $y_{\mathsf{e}^{(i)}+2}^{(i)} \leftarrow y + y_{\mathsf{e}^{(i)}}^{(i)} + y_{\mathsf{e}^{(i)}+1}^{(i)}$
2. Re-compute the commitments for views $\mathsf{View}_{\mathsf{e}^{(i)}}^{(i)}$ and $\mathsf{View}_{\mathsf{e}^{(i)}}^{(i)}$. For $j \in \{\mathsf{e}^{(i)}, \mathsf{e}^{(i)} + 1\}$: $C_j^{(i)} \leftarrow \mathsf{Com}(k_j^{(i)}, x_j^{(i)}, \mathsf{View}_j^{(i)}, y_j^{(i)})$.
3. Set $\mathsf{a}'^{(i)} \leftarrow (y_1^{(i)}, y_2^{(i)}, y_3^{(i)}, C_1^{(i)}, C_2^{(i)}, C_3^{(i)})$ taking $C_{\mathsf{e}^{(i)}+2}^{(i)}$ from $\mathsf{a}^{(i)}$.
4. Re-compute the challenge: $\mathsf{e}' \leftarrow H(\mathsf{a}'^{(1)}, \ldots, \mathsf{a}'^{(t)})$. If $\mathsf{e} = \mathsf{e}'$ output Accept, otherwise Reject.

Scheme 4: The Fiat-Shamir transformed ZKB++ protocol.

assumption here.) Hence we will use the following hash-based commitment scheme:

$\mathsf{Com}(M):$ Set $C \leftarrow H(M)$ and return (C, M);

$\mathsf{Open}(C, D):$ Return M if $H(D) = C$, and return $\perp$ otherwise.

We will prove in Section 5.1 that our proof system is secure when using this scheme.

Not Including the Output Shares. The output shares y_i are included in the proof as part of a. Moreover, for the two views that are opened, those output shares are included a second time. First, we do not need to send two of the output shares twice. We actually do not need to send any output shares at all as they can be deterministically computed from the rest of the proof as follows:

For the two views that are given as part of the proof, the output share can be recomputed from the remaining parts of the view. Essentially, the output share is just the value on the output wires. Given the random tapes and the communicated bits from the binary multiplication gates, all wires for both views can be recomputed.

For the third view, recall that the `Reconstruct` function simply adds the three output shares to obtain y. But the verifier is given y, and can thus instead recompute the third output share. In particular, given y_i, y_{i+1} and y, the verifier can compute: $y_{i+2} = y - y_i - y_{i+1}$. Thus we explicitly specify the output share when required and do not include it in $\mathsf{View}_i^{(j)}$.

When applying the Fiat-Shamir and Unruh transforms to ZKB++ to obtain a signature scheme, we can also perform the following modifications.

Not Including Commitments. It is unnecessary to send all three commitments to the verifier. Since for the two views that are opened, the verifier can recompute the commitment. Only for the third view that the verifier is not given the commitment needs to be explicitly sent.

Security. One can observe that all optimizations except *"No Additional Randomness for Commitments"* are equivalence transformations, and, therefore, do not impact the security of the overall ZKB++ proof system. In Section 5.1, we formally confirm that using no additional randomness for the commitments does not impact the security of the ZKB++ proof system.

2.9 KKW

KKW [KKW18] is another variant of ZKBoo, focusing on minimizing the signature size. It is very similar to ZKBoo and ZKB++ but uses an MPC protocol with more than three parties.

2.9.1 The MPC Protocol

Recall that the prover needs to simulate a set of parties, $S_1, \ldots, S_n$. In KKW, all simulated parties run an n-party protocol Π in the preprocessing model, secure against semi-honest corruption of all-but-one of the parties.

The protocol Π maintains the invariant that, for each wire in the circuit, the parties hold an n-out-of-n secret sharing of a random mask along with the (public) masked value of the wire. Specifically, if we let z_α denote the value of wire α in the circuit C when evaluated on input w, then the parties will hold $[\lambda_\alpha]$ (for uniform $\lambda_\alpha \in \{0,1\}$) along with the value $\hat{z}_\alpha \stackrel{\text{def}}{=} z_\alpha \oplus \lambda_\alpha$.

Preprocessing phase. In the preprocessing phase, shares are set up among the parties as follows. For each wire α that is either an input wire of the circuit or the output wire of an AND gate, the parties are given $[\lambda_\alpha]$, where $\lambda_\alpha \in \{0,1\}$ is uniform. For an XOR gate with input wires α, β and output wire γ, define $\lambda_\gamma \stackrel{\text{def}}{=} \lambda_\alpha \oplus \lambda_\beta$; note the parties can compute $[\lambda_\gamma]$ locally. Finally, for each AND gate with input wires α, β, the parties are given $[\lambda_{\alpha,\beta}]$, where $\lambda_{\alpha,\beta} \stackrel{\text{def}}{=} \lambda_\alpha \cdot \lambda_\beta$.

One observation is that the shares of the $\{\lambda_\alpha\}$ are uniform, and so can be generated by having each party S_i apply a pseudorandom generator to a short, random seed seed_i given to that party, and then (implicitly) defining the $\{\lambda_\alpha\}$ based on the resulting shares. All-but-one of the shares of the $\{\lambda_{\alpha,\beta}\}$ can also be generated in this way, but the final share is constrained by the values of $\lambda_\alpha, \lambda_\beta$. To ensure that the shares of the $\{\lambda_{\alpha,\beta}\}$ are correct, S_n can be given an additional $|C|$ "correction bits" that determine its share of $\lambda_{\alpha,\beta}$ for each AND gate with input wires α, β.

To summarize: each S_i is given a κ-bit seed $\mathsf{seed}_i \in \{0,1\}^\kappa$, and S_n is additionally given $|C|$ bits denoted by aux_n. We refer to this information as the *state* of the parties, and denote the state of S_i by state_i. In the online phase of the protocol, each party S_i uses seed_i to generate its shares of the $\{\lambda_\alpha\}$; for $1 \leq i \leq n-1$, party S_i also uses seed_i to generate its shares of the $\{\lambda_{\alpha,\beta}\}$. Party S_n uses aux_n as its shares of the $\{\lambda_{\alpha,\beta}\}$.

Protocol execution. Note that in this setting, where all parties are semi-honest, we can perform public reconstruction of a shared value $[x]$ by simply having each party broadcast its share.

We assume the parties begin the protocol holding a masked value $\hat{z}_\alpha$ for each input wire α. (In this context these masked values will be provided to the parties by the prover who is simulating execution of the protocol.) These masked values, along with the corresponding $\{\lambda_\alpha\}$, define an effective input to the protocol. During the online phase of the protocol, the parties inductively compute $\hat{z}_\alpha$ for all wires in the circuit. Specifically, for each gate of the circuit with input wires α, β and output wire γ, where the parties already hold $\hat{z}_\alpha, \hat{z}_\beta$, the parties do:

- If the gate is an XOR gate, then the parties locally compute
$$\hat{z}_\gamma := \hat{z}_\alpha \oplus \hat{z}_\beta \ .$$
- If the gate is an AND gate, the parties locally compute
$$[s] := \hat{z}_\alpha[\lambda_\beta] \oplus \hat{z}_\beta[\lambda_\alpha] \oplus [\lambda_{\alpha,\beta}] \oplus [\lambda_\gamma],$$
and then publicly reconstruct s. Finally, they compute $\hat{z}_\gamma := s \oplus \hat{z}_\alpha \hat{z}_\beta$. One can verify that $\hat{z}_\gamma = z_\gamma \oplus \lambda_\gamma$.

Once the parties have computed $\hat{z}_\alpha$ for the output wire α, the output value z_α is computed by publicly reconstructing λ_α and then setting $z_\alpha := \hat{z}_\alpha \oplus \lambda_\alpha$.

We remark that the online phase of this protocol is deterministic. Also observe that all communication is due to share reconstruction: for a circuit with $|C|$ AND gates, at most $|C| + 1$ share reconstructions are needed.

2.9.2 The Proof Protocol

The high-level idea is to have the prover run M emulations of the preprocessing phase and their online phases, and commit to all emulations. The verifier selects $M - \tau$ of them and checks the preprocessing phase; the online phase of the remaining τ executions will be checked by revealing the view of all-but-one parties. The protocol is shown in Figure 1 and Figure 2. In the following, we will discuss more details of KKW.

Checking the preprocessing phase. Recall from the previous section that, following the preprocessing phase, the state of party S_i for $1 \leq i \leq n-1$ is a seed seed_i, while the state of party S_n is a seed seed_n along with a $|C|$-bit string aux_n. We improve the communication complexity in several ways:

KKW Prover Algorithm

Inputs: Both parties have a circuit C; the prover also holds w with $C(w) = 1$. Values M, n, τ are parameters of the protocol.

Round 1 For each $j \in [M]$, the prover does:

1. Choose uniform $\mathsf{seed}_j^* \in \{0,1\}^\kappa$ and use it to generate values $\mathsf{seed}_{j,1}$, $r_{j,1}$, $\ldots$, $\mathsf{seed}_{j,n}, r_{j,n}$. Also compute $\mathsf{aux}_j \in \{0,1\}^{|C|}$ as described in the text. For $i = 1, \ldots, n-1$, let $\mathsf{state}_{j,i} := \mathsf{seed}_{j,i}$; let $\mathsf{state}_{j,n} := \mathsf{seed}_{j,n} \| \mathsf{aux}_j$.
2. For $i \in [n]$, compute $\mathsf{com}_{j,i} := \mathsf{Com}(\mathsf{state}_{j,i}; r_{j,i})$.
3. The prover simulates the online phase of the n-party protocol Π (as described in the text) using $\{\mathsf{state}_{j,i}\}_i$, beginning by computing the masked inputs $\{\hat{z}_{j,\alpha}\}$ (based on w and the $\{\lambda_{j,\alpha}\}$ defined by the preprocessing). Let $\mathsf{msgs}_{j,i}$ denote the messages broadcast by S_i in this protocol execution.
4. Let $h_j := H(\mathsf{com}_{j,1}, \ldots, \mathsf{com}_{j,n})$ and $h'_j := H(\{\hat{z}_{j,\alpha}\}, \mathsf{msgs}_{j,1}, \ldots, \mathsf{msgs}_{j,n})$.

Compute $h := H(h_1, \ldots, h_M)$ and $h' := H(h'_1, \ldots, h'_M)$ and send $h^* := H(h, h')$ to the verifier.

Round 2 The verifier chooses a uniform τ-sized set $\mathcal{C} \subset [M]$ and $\mathcal{P} = \{p_j\}_{j \in [\tau]}$ where each $p_j \in [n]$ is uniform. Send $(\mathcal{C}, \mathcal{P})$ to the prover.

Round 3 For each $j \in [M] \setminus \mathcal{C}$, the prover sends seed_j^*, h'_j to the verifier.

For each $j \in \mathcal{C}$, the prover sends $\{\mathsf{state}_{j,i}, r_{j,i}\}_{i \neq p_j}$, com_{j,p_j}, $\{\hat{z}_{j,\alpha}\}$, and msgs_{j,p_j} to the verifier.

Figure 1: Prover algorithm of the KKW proof protocol.

1. The prover computes $H(\mathsf{com}_1, \ldots, \mathsf{com}_n)$, and then sends the hash of the results from all m executions; thus, it sends just a single hash value to the verifier.

KKW Verifier Algorithm

Verification The verifier accepts iff all the following checks succeed:

1. For every $j \in \mathcal{C}$, $i \neq p_j$, the verifier uses $\mathsf{state}_{j,i}$ and $r_{j,i}$ to compute $\mathsf{com}_{j,i}$. It then computes $h_j := H(\mathsf{com}_{j,1}, \ldots, \mathsf{com}_{j,n})$.
2. For $j \in [M] \backslash \mathcal{C}$, the verifier uses seed_j^* to compute h_j as an honest prover would. It then computes $h := H(h_1, \ldots, h_M)$.
3. For each $j \in \mathcal{C}$, the verifier simulates an execution of Π among the $\{S_i\}_{i \neq p_j}$ using $\{\mathsf{state}_{j,i}\}_{i \neq p_j}$, masked input-wire values $\{\hat{z}_\alpha\}$, and msgs_{j,p_j}. This yields $\{\mathsf{msgs}_i\}_{i \neq p_j}$ and an output bit b. The verifier checks that $b \overset{?}{=} 1$ and computes $h'_j := H(\{\hat{z}_{j,\alpha}\}, \mathsf{msgs}_{j,1}, \ldots, \mathsf{msgs}_{j,n})$ as well as $h' := H(h'_1, \ldots, h'_m)$.
4. The verifier checks that $H(h, h') \overset{?}{=} h^*$.

Figure 2: Verifier algorithm of the KKW proof protocol.

2. When opening a challenged execution, it is unnecessary for the prover to send aux_n since the correct value of aux_n can be computed from $\mathsf{seed}_1, \ldots, \mathsf{seed}_n$.
3. By generating the $\{\mathsf{seed}_i\}$ and the $\{r_i\}$ from a "root" seed $\mathsf{seed}^* \in \{0,1\}^\kappa$, the prover can open a challenged execution of the preprocessing phase by simply sending seed^*.

Checking the online execution. An execution of the protocol proceeds gate-by-gate, with the processing of each AND gate requiring reconstruction of one shared value. Although the communication complexity of share reconstruction in the protocol is n bits (one bit per party), for the purposes of checking, we do not need the prover to send n bits per gate in order to prove consistent execution. This is because the verifier only needs to obtain the protocol messages sent by the (single) *unopened* party in order to check the execution of the $n-1$ opened parties. Thus, it suffices for the prover to send just a single bit per AND gate.

In addition to the protocol messages sent by the unopened party, the prover also needs to reveal the state (from the preprocessing phase) of every opened party. For each opened party S_i, $i \neq n$, this involves just $O(\kappa)$ bits; if S_n is opened then this requires $|C| + O(\kappa)$ bits due to aux_n. In either case the marginal communication complexity per AND gate is *independent* of the number of parties n.

Reducing the number of random seeds. In the cth emulation of the preprocessing phase, the prover generates n seeds $\mathsf{seed}_{c,1}$, $\ldots$, $\mathsf{seed}_{c,n}$ from a root seed seed_c^*, commits to the n generated seeds, and then sends $n-1$ of those seeds to the verifier. The second step requires $(n-1) \cdot \kappa$ bits of communication.

Motivated by the NNL scheme for stateless revocation [NNL01], we observe that we can reduce the communication by generating the seeds in a more structured way. Namely, imagine labeling the root of a binary tree of depth $\log n$ with seed_c^*, and then inductively labeling the children of each node with the output of a pseudorandom generator applied to the node's label. The $\{\mathsf{seed}_{c,i}\}_{i\in[n]}$ will be the labels of the n leaves of the tree. To reveal $\{\mathsf{seed}_{c,i}\}_{i\neq p}$, it suffices to reveal the labels on the siblings of the path from the root of the tree to leaf p. Those labels allow the verifier to reconstruct $\{\mathsf{seed}_{c,i}\}_{i\neq p}$ while still hiding $\mathsf{seed}_{c,p}$. Applying this optimization reduces the communication complexity to $O(\kappa \cdot \log n)$ for revealing the seeds used by the $n-1$ opened parties.

We can, in fact, apply the same idea to the root seeds $\{\mathsf{seed}_j^*\}_{j=1}^m$ used for the different emulations of the preprocessing phase; this reduces the communication required to reveal all-but-one of those seeds in Round 3 from $(m-1) \cdot \kappa$ bits to $O(\kappa \cdot \log m)$ bits. Further, we are not limited to revealing all-but-one of the leaf labels; more generally, the scheme just described supports revealing all-but-τ of the leaf labels using communication at most $O(\kappa \cdot \tau \log \frac{m}{\tau})$ bits (cf. [NNL01]).

Reducing the size of commitments with Merkle trees. $M - \tau$ of the commitments $h'_1, \ldots, h'_M$ are sent as part of the proof for the instances where $j \notin C$. These are necessary for verification, when recomputing the challenge the prover provides $M - \tau$ of the commitments, and the verifier recomputes the other τ. We can reduce the proof size by having the prover commit to $h'_1, \ldots, h'_M$ using a Merkle tree and hash the root when computing the challenge. Then, the verifier is given the root and enough information to confirm that the τ commitments he recomputes are in fact committed to by the root. In this way, only $O(\tau \log(M/\tau))$ hash values are are communicated, which is much less than $M - \tau$ because τ is much smaller than M in our parameter sets.

Reducing the size of commitment openings. As in ZKB++, we can reduce the size of openings for commitments by not using additional randomness when forming the commitments $\mathsf{com}_{i,j}$ and h'_j. Intuitively, since other values in the commitment have sufficient entropy from an attacker's perspective, having the commitment does not provide additional useful information. This is analyzed more formally in Section 6.2, where our security proof for the signature scheme uses commitments without additional randomness.

2.10 LowMC

LowMC [ARS+15, ARS+16] is a very parameterizable symmetric encryption scheme design enabling instantiation with low AND depth and low multiplicative complexity. Given any blocksize, a choice for the number of S-boxes per round, and security expectations in terms of time and data complexity, instantiations can be created minimizing the AND depth, the number of ANDs, or the number of ANDs per encrypted bit. Table 1 lists the choices for the parameters for security levels L1, L3, L5.

The description of LowMC is possible independently of the choice of parameters using a partial specification of the S-box and arithmetic in vector spaces over $\mathbb{F}_2$. In particular, let n be the blocksize, m be the number of S-boxes, k the key size, and r the number of rounds, we choose round constants $C_i \xleftarrow{R} \mathbb{F}_2^n$ for $i \in [1, r]$, full rank matrices $K_i \xleftarrow{R} \mathbb{F}_2^{n \times k}$ and regular matrices $L_i \xleftarrow{R} \mathbb{F}_2^{n \times n}$ independently during the instance generation and keep them fixed. Keys for LowMC are generated by sampling from $\mathbb{F}_2^k$ uniformly at random.

LowMC encryption starts with key whitening which is followed by several rounds of encryption. A single round of LowMC is composed of an S-box layer, a linear layer, addition with constants and addition of the round key, i.e.

$$\begin{aligned}\textsc{LowMCRound}(i) = {} & \textsc{KeyAddition}(i) \\ & \circ \textsc{ConstantAddition}(i) \\ & \circ \textsc{LinearLayer}(i) \circ \textsc{SboxLayer}.\end{aligned}$$

SboxLayer is an m-fold parallel application of the same 3-bit S-box on the first $3 \cdot m$ bits of the state. The S-box is defined as $S(a, b, c) = (a \oplus bc, a \oplus b \oplus ac, a \oplus b \oplus c \oplus ab)$.

The other layers only consist of $\mathbb{F}_2$-vector space arithmetic. LinearLayer(i) multiplies the state with the linear layer matrix L_i, ConstantAdditon(i) adds the round constant C_i to the state, and KeyAddition(i) adds the round key to the state, where the round key is generated by multiplying the master key with the key matrix K_i.

Algorithm 1 gives a full description of the encryption algorithm.

Algorithm 1 LowMC encryption for key matrices $K_i \in \mathbb{F}_2^{n \times k}$ for $i \in [0, r]$, linear layer matrices $L_i \in \mathbb{F}_2^{n \times n}$ and round constants $C_i \in \mathbb{F}_2^n$ for $i \in [1, r]$.

```
Require: plaintext $p \in \mathbb{F}_2^n$ and key $y \in \mathbb{F}_2^k$
  $s \leftarrow K_0 \cdot y + p$
  for $i \in [1, r]$ do
      $s \leftarrow Sbox(s)$
      $s \leftarrow L_i \cdot s$
      $s \leftarrow C_i + s$
      $s \leftarrow K_i \cdot y + s$
  end for
  return $s$
```

LowMC is very flexible in the choice of parameters: the block size n, the key size k, the number of 3-bit S-boxes m in the substitution layer and the allowed data complexity d of attacks can independently be chosen. To reduce the multiplicative complexity, the number of S-boxes applied in parallel can be reduced, leaving part of the substitution layer as the identity mapping. The number of rounds r needed to achieve the goals is then determined as a function of all these parameters. We discuss concrete choices of the parameters in Section 4.2.

3 The Picnic Signature Schemes

We consider several schemes, all obtained by transforming an interactive zero-knowledge protocol into a (non-interactive) signature. Different signature schemes are obtained by varying the zero-knowledge protocol used and the transformation that is applied. Picnic-FS uses ZKB++ with the Fiat-Shamir transform, Picnic-UR uses ZKB++ with the Unruh transform, and Picnic2 uses the proof protocol of [KKW18] with the Fiat-Shamir transform. In Scheme 5 we provide a general description of our schemes.

For all schemes, the public key contains values u, y, and the signer proves knowledge of a pre-image x of y with respect to a one-way function f_u. In all cases, we instantiate the one way function with LowMC; specifically, if F denotes the LowMC block cipher then we choose a uniform domain element u and define the one-way function f_u via $f_u(x) = F_x(u)$. (See Section 2.10 for an overview of LowMC and Section 4.2 for a discussion of our parameter choices.) One-wayness of f_u follows from the assumption that LowMC is a secure block cipher (i.e., pseudorandom permutation); Section 7 discusses the difficulty of inverting f_u using known attacks.

$\underline{\mathsf{Gen}(1^\kappa) :}$ Choose $u \xleftarrow{R} \mathsf{K}_\kappa$, $x \xleftarrow{R} \mathsf{D}_\kappa$, compute $y \leftarrow f_u(x)$, set $\mathsf{pk} \leftarrow (y, u)$ and $\mathsf{sk} \leftarrow (\mathsf{pk}, x)$ and return $(\mathsf{sk}, \mathsf{pk})$.

$\underline{\mathsf{Sign}(\mathsf{sk}, m) :}$ Parse sk as (pk, x), compute $p = (r, s) \leftarrow \mathtt{Prove}_H((y, u), x)$ and return $\sigma \leftarrow p$, where internally the challenge is computed as $c \leftarrow H(r, \mathsf{pk}||m)$.

$\underline{\mathsf{Verify}(\mathsf{pk}, m, \sigma) :}$ Parse pk as (y, u), and σ as $p = (r, s)$. Return 1 if the following holds, and 0 otherwise:

$$\mathtt{Verify}_H((y, u), p) = 1,$$

where internally the challenge is computed as $c \leftarrow H(r, \mathsf{pk}||m)$.

Scheme 5: Generic description the Picnic-FS and Picnic2-FS signature schemes. Scheme Picnic-UR is similar, except `Prove` and `Verify` are different.

3.1 Efficient Instantiation of Unruh's Transform

Although Unruh's transformation was only designed for Σ-protocols with 2-special soundness, we can easily modify it for ZKB++ (which has 3-special soundness).

Since ZKB++ has 3-special soundness, we would need at least three responses for each iteration. Moreover, since there are only three possible challenges in ZKB++, we run Unruh's transform with $\mathsf{E} = \{1, 2, 3\}$ and $M = 3$—i.e., every possible challenge and response. If we apply this naively, we obtain the following protocol:

$\mathtt{Prove}_H(1^\kappa, (x, m), w)$: 1. For $i \in [t]$:

(a) Start P on $(1^\kappa, x, w)$ and obtain first message a_i.

(b) For all $\mathsf{e}_{i,j} = j \in \mathsf{E}$, obtain response $\mathsf{z}_{i,j}$ for challenge $\mathsf{e}_{i,j}$.

2. For $(i, j) \in [t] \times \mathsf{E}$, set $g_{i,j} \leftarrow G(\mathsf{z}_{i,j})$.
3. Let $(J_1, \ldots, J_t) \leftarrow H(m, (\mathsf{a}_i)_{i\in[t]}, (g_{i,1}, \ldots, g_{i,3})_{i\in[t]})$
4. Return $\pi \leftarrow ((\mathsf{a}_i)_{i\in[t]}, (g_{i,1}, \ldots, g_{i,3})_{(i,j)\in[t]}, (\mathsf{z}_{i,J_i})_{i\in[t]})$

As we no longer randomly select the challenges, we can omit them as input to the hash function and do not need to include them in the proof.

To instantiate the function G in the protocol, Unruh shows that one does not need a random oracle that is actually a permutation. Instead, as long as the domain and codomain of G are the same size (and large enough), it can be used, since it is indistinguishable from a random permutation. So let $G : \{0, 1\}^{|\mathsf{z}_{i,j}|} \to \{0, 1\}^{|\mathsf{z}_{i,j}|}$ be a hash function modeled as a random oracle. The size of the response changes depending on what the challenge is. If the challenge is 0, the response is slightly smaller as it does not need to include the extra input share. So more precisely, this is actually two hash functions, G_0 used for the 0-challenge response and $G_{1,2}$ used for the other two challenges. In our specification document we define G precisely.

Optimization 1: Making Use of Overlapping Responses. We can make use of the structure of the ZKB++ proofs to achieve a very significant reduction in the proof size. Although we refer to three separate challenges, in the case of the ZKB++ protocol, there is a large overlap between the contents of the responses corresponding to these challenges. In particular, there are only three distinct views in the ZKB++ protocol, two of which are opened for a given challenge.

Instead of computing a permutation of each *response*, $\mathsf{z}_{i,j}$, we can compute a permutation of each *view*, $v_{i,j}$. For each $i \in \{1, \ldots, t\}$, and for each $j \in \mathsf{E}$, the prover computes $g_{i,j} = G(v_{i,j})$.

The verifier checks the permuted value for each of the two views in the response. In particular, for challenge $j \in \{1, 2, 3\}$, the verifier will need to check that $g_{i,j} = G(v_{i,j})$ and $g_{i,j+1} = G(v_{i,j+1})$.

Optimization 2: Omit Re-Computable Values. Moreover, since G is a public function, we do not need to include $G(v_{i,j})$ in the transcript if we have included $v_{i,j}$ in the response. Thus for the two views (corresponding to a single challenge) that the prover sends as part of the proof, we do not need to include the permutations of those views. We only need to include $G(v_{i,(j+2)})$, where $v_{i,(j+2)}$ is the view that the prover does not open for the given challenge.

3.2 Seed Generation

We generate seeds for the random tapes using SHAKE with the private key, message, and public key as input and requesting the required number of output bytes from the XOF. Complete details are given in the specification document. Our current implementation and specification use deterministic signatures as a default to facilitate testing, however the specification shows how to randomize signatures by including additional entropy in the derivation process. The specification recommends randomizing signatures, especially when side-channel attacks are a concern.

3.3 Random Tapes

We generate random tapes using the SHAKE XOF as a KDF to expand the seed to the required number of output bytes. Complete details are given in the specification document.

3.4 Challenge Generation

For both the FS and Unruh transform the challenge is computed with a hash function. For Picnic-FS and Picnic-UR the function is $H : \{0,1\}^* \to \{0,1,2\}^t$ (implemented using SHAKE) and rejection sampling: we split the output bits in pairs of two bits and reject all pairs with both bits set. For Picnic2 the range of the function is a set and a list of integers whose size depends on the parameter set. The same rejection sampling method is used

3.5 Function G

As explained in Section 3.1, the function G used in Picnic-UR may be implemented with a hash function with the same domain and range. We implement $G(x)$ with SHAKE, where the requested number of output bits is $|x|$.

4 Choice of Parameters

In this section we explain our parameter selection and rationale.

4.1 Choice of LowMC and SHAKE

The signature size depends on constants that are close to the security expectation. The only exceptions are the number of binary multiplication gates, and the size of the ring, which both depend on the choice of the primitive. In this section we compare LowMC to existing standardized primitives and to other primitives with a low number of multiplications.

Standardized Primitives. The smallest known Boolean circuit for AES-128 needs 5440 AND gates, AES-192 needs 6528 AND gates, and AES-256 needs 7616 AND gates [BMP13]. An AES circuit in $\mathbb{F}_{2^4}$ might be more efficient in our setting, as in this case the number of multiplications is lower than 1000 [CGP+12]. This results in an impact on the signature size that is equivalent to 4000 AND gates. Even though collision resistance is often not required, hash functions like SHA-256 are a popular choice for proof-of-concept implementations. The number of AND gates of a single call to the SHA-256 compression function is about 25000 and a single call to the permutation underlying SHA-3 is 38400.

Lightweight Ciphers. Most early designs in this domain focused on small area when implemented in hardware where an XOR gate is by a small factor larger than an AND or NAND gate. Notable designs with a low number of AND gates at the 128-bit security level are the block ciphers Noekeon [DPVAR00] (2048 ANDs) and Fantomas [GLSV14] (2112 ANDs). Furthermore, one should mention Prince [BCG+12] (1920 ANDs), or the stream cipher Trivium [DP08] (1536 AND gates to compute 128 output bits, with 80-bit security).

Custom Ciphers with a Low Number of Multiplications. Motivated by applications in SHE/FHE schemes, MPC protocols and SNARKs, recently a trend to design symmetric encryption primitives with a low number of multiplications or a low multiplicative depth started to evolve. This is a trend we can take advantage of.

We start with the LowMC [ARS+15] block cipher family. In the most recent version of the design [ARS+16], the number of AND gates can be below 500 for 80-bit security, below 800 for 128-bit security, and below 1400 for 256-bit security. The stream cipher Kreyvium [CCF+16] (similarly to Trivium) needs 1536 AND gates to compute 128 output bits, but offers a higher security level of 128 bits. Even though FLIP [MJSC16] was designed to have especially low depth, it needs hundreds of AND gates per bit and is hence not competitive in our setting.

Last but not least there are the block ciphers and hash functions around MiMC [AGR+16] which need less than $2 \cdot s$ multiplications for s-bit security in a field of size close to 2^s. Note that MiMC is the only design in this category which aims at minimizing multiplications in a field larger than $\mathbb{F}_2$. However, since the size of the signature depends on both the number of multiplications and the size of the field, this leads to a factor $2s^2$ which, for all arguably secure instantiations of MiMC, is already larger than the number of AND gates in the AES circuit.

LowMC has two important advantages over other designs: It has the lowest number of AND gates for every security level: The closest competitor Kreyvium needs about twice as many AND gates and only exists for the 128-bit security level. The fact that it allows for an easy parameterization of the security level is another advantage. We hence use LowMC for our concrete proposal.

Hashing with SHAKE. Keccak is a family of cryptographic primitives including hash functions and extensible output functions (XOF). It was selected as the successor to SHA-2 and was standardized as SHA3 [NIS15]. SHAKE is an XOF constructed from SHA3. Since both SHA3 and SHAKE have a large number of AND gates (as described above), we do not use them for Picnic key generation.

However, other parts of the ZKB++ protocol require a hash function. We will use SHAKE in two modes

1. as a hash function with a fixed output length

2. as a key derivation function, where we expand a fixed length seed into a larger pseudorandom value, by requesting larger, variable sized SHAKE outputs.

For the Picnic2 parameter sets, in our current implementation hashing with SHAKE accounts for roughly 30% of signing runtime (at security level L1). If in the future, a faster hash function becomes widely viewed as providing sufficient security (and example candidate is Kangaroo12 [BDP+18]), it may be worth considering using it with Picnic2.

Security level	Blocksize n	S-boxes m	Keysize k	Rounds r
L1	128	10	128	20
L3	192	10	192	30
L5	256	10	256	38

Table 1: Parameters for LowMC targeting security levels L1, L3 and L5. All parameters are computed for data complexity $d = 1$.

4.2 LowMC Parameters

To minimize the number of AND gates for a given key length k and data complexity d, we want to minimize $r \cdot m$ (where r is the number of rounds and m is the number of sboxes). One strategy would be to set m to 1, and to look for an n that minimizes r. Examples of such an approach are already given in the document describing version 2 of the LowMC design [ARS+16]. In our setting, this approach may not lead to the best results, as it ignores the impact of the large amount of XOR operations it requires. While Picnic signatures defined with these parameters have minimal length, the large number of XOR gates make singing and verification slow. To find the most suitable parameters, we thus explore a larger range of values for m, looking to balance signing and verification cost with signature size.

Whenever we want to instantiate our signature scheme with LowMC with κ-bit quantum security, we set $k = n = 2 \cdot \kappa$. This matches AES, and the security levels in the NIST call for proposals.

Furthermore, we observe that for a given key the adversary only ever sees a single plaintext-ciphertext pair (namely, the public key in the Picnic scheme). This is why we can set the data complexity $d = 1$[2].

4.3 Number of Parallel Repetitions

Parameters for ZKB++. A single repetition of ZKB++ has a soundness error of $2/3$, which means that we need to perform parallel repetitions to achieve the desired soundness error. Hence we need 219 parallel repetitions for 128-bit classical security ($(3/2)^{219} \geq 2^{128}$). For 128-bit PQ security, we

[2] d is given in units of $\log_2(n)$, where n is the number of pairs. Thus setting $d = 1$ corresponds to 2-pairs, which is exactly what we need for our signature schemes.

must set our repetition count to $t \coloneqq 438$. This is double the repetition count required for classical security due to Grover's algorithm [Gro96]. The required number of repetitions for the L1, L3 and L5 security levels are given in Table 2.

level	# parallel repetitions
L1	219
L3	329
L5	418

Table 2: Number of parallel repetitions required at each security level.

Parameters for KKW. In the KKW protocol, there are multiple parameters to be chosen, including the number of emulated parties n, the number of total executions M, and the number of online executions checked by the verifier τ. The soundness error ϵ, depends on the choice of (M, n, τ), and is computed with the following equation.

$$\epsilon(M, n, \tau) = \max_{M-\tau \leq k \leq M} \left\{ \frac{\binom{k}{M-\tau}}{\binom{M}{M-\tau} \cdot n^{k-M+\tau}} \right\}.$$

In Table 3, we list some example parameter choices that minimize signature size. Each choice of (M, n, τ) represents different trade offs in computation and signature size. Our specification fixes $n = 64$, since generally larger n gives smaller signatures, and it allows allows implementations to pack the bit shares of each party into a 64-bit word, and operate on them with machine word operations.

Our specified choices of M and τ move away from the smallest possible signature size (e.g., at level L1 (M, τ) is $(343, 27)$ compared to $(631, 20)$ in Table 3), in order to reduce M. We found that for a small increase in size, M is significantly reduced, which significantly reduces the CPU cost of signing and verification.

4.4 Alternative Parameters

In this section we list some possible alternatives to the parameters we chose, and discuss the tradeoffs. These apply to the Picnic2 parameters, since the KKW protocol has more parameters than the ZKB++ protocol. Some

	n	4	8	16	32	64	128
L1	M	218	252	352	462	631	916
	τ	65	44	33	27	23	20
	n	4	8	16	32	64	128
L3	M	319	376	563	740	1017	1677
	τ	98	66	49	40	34	29
	n	4	8	16	32	64	128
L5	M	456	533	781	1024	1662	2540
	τ	129	87	65	53	44	38

Table 3: Example combinations of parameters (M, n, τ) for KKW at each security level.

alternative parameters for LowMC, that apply to all parameter sets, were discussed in Section 4.2 and [CDG+17].

Faster (but larger) signatures The main computational cost in our current implementation of the Picnic2 parameter sets is the online simulation step of the M MPC instances. We can choose a smaller M, provided we increase τ to maintain the soundness error ϵ. Larger τ increases the signature size. We searched the parameter space and chose one alternative parameter set and benchmarked the **Optimized-C** implementation on our benchmark Platform C (these are described in Section 11). Table 4 compares the possible "fast" parameters to the parameters we chose.

Parameter set	(M, τ)	Size	Sign	Verify
`Picnic2-L1-FS`	$(343, 27)$	12K	155	87
L1 fast	$(133, 60)$	21K	60	42
`Picnic2-L3-FS`	$(570, 39)$	27K	421	189
L3 fast	$(198, 91)$	48K	146	95
`Picnic2-L5-FS`	$(803, 50)$	46K	848	328
L5 fast	$(264, 119)$	82K	279	172

Table 4: Alternative Picnic2 parameter sets that trade off size for CPU cost. The timings are in milliseconds, the sizes are in bytes, rounded up and averaged from 10 runs.

An alternative size/speed tradeoff can be made by changing n, the number of parties. Using larger n increases the computational cost, but can let us decrease τ to have shorter signatures. We chose $n = 64$ because it matches the word size on 64-bit processors, and is therefore convenient for packing binary shares of all parties in a single word. Also note that small changes in n have limited effect on ϵ, but do have appreciable CPU costs. On SIMD implementations with 128-bit and larger registers it may be worth considering $n = 128$, and similarly on constrained platforms setting n to the word size may be advantageous.

Using a 5-round protocol The KKW interactive proof protocol can naturally be done as a five round protocol (see [KKW18, Figure 1]). In the first round the prover commits to the offline portion of many MPC instances, the verifier selects a subset, then the prover commits to the online simulations (using the selected subset), and the verifier selects one party from each instance to remain unopened. The potential benefit of this approach is that the prover needs to perform the online MPC simulation for a much smaller number of instances (roughly τ instead of M). This is significant, since in our current implementation the online part of the MPC simulation is much slower than the offline part (roughly 5x slower).

However, when making this non-interactive, the soundness error is given by a different formula (since both challenges must be sufficiently large), and we must use different values of M and τ. If we directly instantiate a five round protocol, M must be extremely large, making performance must worse than a three round protocol (even though there are fewer online simulations, the offline simulations are not completely free). We are investigating a variant of the five round version where the prover does more than τ online simulations but fewer than M, this a promising approach to reducing computational costs, and this investigation is ongoing.

5 Formal Security Analysis

This section contains formal security analyses of the Picnic-FS and Picnic-UR schemes (Picnic2 is analyzed in §6). We begin in Section 5.1 by analyzing the (interactive) ZKB++ protocol. We then separately analyze Picnic-FS and Picnic-UR, which differ in the transform applied to ZKB++, in the sense of unforgeability. As we discuss in Section 5.4, the scheme can actually be shown to satisfy *strong* unforgeability.

5.1 Security Analysis of ZKB++

First, we observe that not including output shares and commitments are what we call equivalence transformations: there is a transformation (which anyone can compute) which takes a ZKBoo proof and removes output shares and commitments, or which takes a proof without the output shares and commitments and produces a proof which does again include these values. Thus removing these values does not reveal any more information or make it any easier to forge proofs.

The other modification we make is to generate the initial shares pseudorandomly. Note that this cannot make it easier to forge proofs, because we are only reducing the options the prover has in choosing the shares. On the other hand, we note that the 2-privacy simulator for the decomposition works even if the initial random shares are generated pseudorandomly, so the zero-knowledge proof still goes through. Again, after this step, removing the input shares is an equivalence transformation that has no effect on security.

Thus, we only have to show that including no additional randomness in the commitments preserves completeness, 3-special soundness, and special honest-verifier zero-knowledge of ZKB++.

First, we observe that completeness is clearly not impacted and the corollary below follows from this and [GMO16b, Proof of Proposition 2].

Corollary 5.1. *The modified version of* ZKBoo*—where the commitments no longer contain additional randomness—is complete, i.e.,* ZKB++ *is complete.*

Second, under the observation that removing the randomness in the commitments does not impact the binding property of the commitments we can derive the following corollary from [GMO16b, Proof of Proposition 2].

Corollary 5.2. *The modified version of* ZKBoo*—where the commitments no longer contain additional randomness—is 3-special sound, i.e.,* ZKB++ *is 3-special sound.*

What remains is to prove the following theorem.

Theorem 5.3. *The modified version of* ZKBoo*—where the commitments no longer contain additional randomness—is special honest-verifier zero-knowledge in the random oracle model, i.e.,* ZKB++ *is special honest-verifier zero-knowledge in the (quantum) random oracle model.*

Before we prove the theorem, we recall that—as the challenge can be determined a priori in the proof for special honest-verifier zero-knowledge—we can use the 2-privacy simulator of the (2,3)-decomposition underlying ZKB++ (cf. Section 2.7 for details) to produce satisfying transcripts for the two views which need to be opened according to the challenge. Now, in the original proof [GMO16b, Proof of Proposition 2], the hiding property of the commitment which is not required to be opened ensures that the simulation works out. We have to argue that this still holds when no additional randomness is included in the commitments. Since we already use the random oracle heuristic for our signature scheme, we also rely on the random oracle heuristic for the subsequent proof.

Proof (Sketch). Recall the modification discussed in Section 2.8. We consider the random oracle model, in which H', H'' are independent random oracles. Then, consider the seed k_i for each the unopened view: this k_i is only used as input to H', H''. So the initial prover is distributed identically to another prover which replaces $H'(k_i), H''(k_i)$ for the unopened views with random values Z, Z'. Now, Z' is a random value which is only used as input to the commitment, so we can apply the same HVZK argument as ZKBoo. □

5.2 Security Analysis of Picnic-FS

If we view ZKB++ as a canonical identification scheme that is secure against passive adversaries one just needs to keep in mind that most definitions are tailored to (2-)special soundness, and the 3-special soundness of ZKB++ requires an additional rewind. In particular, an adapted version of the proof of [Kat10, Theorem 8.2], which considers this additional rewind, attests the security of Picnic-FS. We obtain the following:

Corollary 5.4. *Picnic-FS instantiated with* ZKB++ *and a secure one-way function yields an* EUF-CMA *secure signature scheme in the ROM.*

However, we actually aim for a stronger result, i.e., sEUF-CMA, which also excludes malleability of the signatures. To show that Picnic-FS also satisfies this property, we need to view ZKB++ as a Σ-protocol which is transformed to its non-interactive counterpart via the FS transform and show that this protocol is actually simulation extractable. We base our argumentation upon the argumentation of [FKMV12] to confirm simulation extractability. What we have to do is to show that the FS transformed ZKB++ is zero-knowledge and provides quasi-unique responses. We do so by proving two lemmas. Combining those lemmas with [FKMV12, Theorem 2 and Theorem 3] then yields simulation extractability as a corollary.

Lemma 5.5. *Let Q_H be the number of queries to the random oracle H, Q_S be the overall queries to the simulator, and let the commitments be instantiated via a RO H' with output space $\{0,1\}^\rho$ and the committed values having min entropy ν. Then the probability $\epsilon(\kappa)$ for all PPT adversaries $\mathcal{A}$ to break zero-knowledge of κ parallel executions of the FS transformed* ZKB++ *is bounded by $\epsilon(\kappa) \leq {}^{s}/_{2^\nu} + {}^{(Q_S \cdot Q_H)}/_{2^{3 \cdot \rho}}$.*

The subsequent proof is similar to the general results for Σ-protocols from [FKMV12], yet we have to account for the additional challenge that the simulator only outputs transcripts which are statistically close to original transcripts (which is in contrast to the identically distributed transcripts in [FKMV12]). Furthermore, we also provide concrete bounds.

Proof. We bound the probability of any PPT adversary $\mathcal{A}$ to win the zero-knowledge game by showing that the simulation of the proof oracle is statistically close to the real proof oracle. For our proof let the environment maintain a list $\mathtt{H}$ where all entries are initially set to $\perp$.

Game 0: The zero-knowledge game where the proofs are honestly computed, and the ROs are simulated honestly.

Game 1: As Game 0, but whenever the adversary requests a proof for some tuple (x, w) we choose $\mathsf{e} \xleftarrow{R} \{0,1,2\}^\kappa$ before computing a and z. If $\mathtt{H}[(\mathsf{a}, x)] \neq \perp$ we abort and call that event E. Otherwise, we set $\mathtt{H}[(\mathsf{a}, x)] \leftarrow \mathsf{e}$.

Transition - Game 0 → Game 1: Game 0 and Game 1 proceed identically unless E happens. The message a includes 3 RO commitments with respect to H', i.e., a lower bound for the min-entropy is $3 \cdot \rho$. We have that $|\Pr[S_0] - \Pr[S_1]| \leq {}^{(Q_S \cdot Q_H)}/_{2^{3 \cdot \rho}}$.

Game 2: As Game 1, but we compute the commitments in a so that the commitments which will never be opened according to e contain random values.

Transition - Game 1 → Game 2: The statistical difference between Game 1 and Game 2 can be upper bounded by $|\Pr[S_1] - \Pr[S_2]| \leq \kappa \cdot {}^{1}/_{2^\nu}$ (for compactness we collapsed the s game changes into a single game).

Game 3: As Game 2, but we use the HVZK simulator to obtain $(\mathsf{a}, \mathsf{e}, \mathsf{z})$.

Transition - Game 2 → Game 3: This change is conceptual, i.e., $\Pr[S_2] = \Pr[S_3]$.

In Game 0, we sample from the first distribution of the zero-knowledge game, whereas we sample from the second one in Game 3; the distinguishing bounds shown above conclude the proof. □

Lemma 5.6. *Let the commitments be instantiated via a RO H' with output space $\{0,1\}^\rho$ and let $Q_{H'}$ be the number of queries to H', then the probability to break quasi-unique responses is bounded by $Q_{H'}^2/2^\rho$.*

Proof. To break quasi-unique responses, the adversary would need to come up with two valid proofs $(\mathsf{a}, \mathsf{e}, \mathsf{z})$ and $(\mathsf{a}, \mathsf{e}, \mathsf{z}')$. The last message z (resp z') only contains openings to commitments, meaning that breaking quasi unique responses implies finding a collision for at least one of the commitments. The probability for this to happen is upper bounded by $Q_{H'}^2/2^\rho$ which concludes the proof. □

Combining Lemma 5.5 and Lemma 5.6 with [FKMV12, Theorem 2 and Theorem 3] yields the following corollary.

Corollary 5.7. *The FS transformed* ZKB++ *protocol is simulation extractable.*

5.3 Security Analysis of Picnic-UR

Here we prove that the proof system we get by applying our modified Unruh transform to ZKB++ is both zero knowledge and simulation extractable in the quantum random oracle model.

Before we begin, we note that the quantum random oracle model is highly non-trivial, and a lot of the techniques used in standard random oracle proofs do not apply. The adversary is a quantum algorithm that may query the oracle on quantum inputs which are a superposition of states and receive superposition of outputs. If we try to measure those states, we change the outcome, so we do not for example have the same ability to view the adversary's input and program the responses that we would in the standard ROM.

Here we rely on lemmas from Unruh's work on quantum-secure Fiat-Shamir like proofs [Unr15]. We follow his proof strategy as closely as possible, modifying it to account for the optimizations we made and the fact that we have only 3-special soundness in our underlying Σ-protocol.

Zero-Knowledge. This proof very closely follows the proof from [Unr15]. The main difference is that we also use the random oracle to form our commitments, which is addressed in the transition from game 2 to game 3 below.

Consider the simulator described in Figure 3. From this point on we assume for simplicity of notation that View_3 includes x_3.

We proceed via a series of games.

Game 1: This is the real game in the quantum random oracle model. Let H_{com} be the random oracle used for forming the commitments, H_{chal} be the random oracle used for forming the challenge, and G be the additional random permutation.

Game 2: We change the prover so that it first chooses random $e* = e*^{(1)}, \ldots, e*^{(t)}$, and then on step 2, it programs $H_{chal}(a^{(1)}, \ldots, a^{(t)}, h^{(1)}, \ldots, h^{(t)}) = e*$.

Note that each the $a^{(1)}, \ldots, a^{(t)}, h^{(1)}, \ldots, h^{(t)}$ has sufficient collision-entropy, since it includes $\{h^{(i)} = (g_1^{(i)}, g_2^{(i)}, g_3^{(i)})\}$, the output of a permutation on input whose first k bits are chosen at random (the $k_j^{(i)}$), so we can apply Corollary 11 from [Unr15] (using a hybrid argument) to argue that Game 1 and Game 2 are indistinguishable.

Game 3: We replace the output of each $H_{com}(k_{e*^{(i)}}, \mathsf{View}_{e*^{(i)}})$ and $G(k_{e*^{(i)}}, \mathsf{View}_{e*^{(i)}})$ with a pair of random values.

First, note that H_{com} and G are always called (by the honest party) on the same inputs, so we will consider them as a single random oracle with a longer output space, which we refer to as H for this proof.

Now, to show that Games 2 and 3 are indistinguishable, we proceed via a series of hybrids, where the i-th hybrid replaces the first i such outputs with random values.

To show that the i-th and $i+1$-st hybrid are indistinguishable, we rely on Lemma 9 from [Unr15]. This lemma says the following: For any quantum A_0, A_1 which make q_0, q_1 queries to H respectively and classical A_C, all three of which may share state, let P_C be the probability

$p \leftarrow \texttt{Sim}(x)$: In the simulator, we follow Unruh, and replace the initial state (before programming) of the random oracles with random polynomials of degree $2q-1$ where q is an upper bound on the number of queries the adversary makes.

1. For $i \in [1,t]$, choose random $e^{(i)} \leftarrow \{1,2,3\}$. Let e be the corresponding binary string.
2. For each iteration $r_i, i \in [1,t]$: Sample random seeds $k^{(i)}_{e^{(i)}}, k^{(i)}_{e^{(i)}+1}$ and run the circuit decomposition simulator to generate $\mathsf{View}^{(i)}_{e^{(i)}}$, $\mathsf{View}^{(i)}_{e^{(i)}+1}$, output shares $y^{(i)}_1$, $y^{(i)}_2$, $y^{(i)}_3$, and if $e^{(i)} = 1$ $x^{(i)}_3$.

   For $j = e^{(i)}, e^{(i)} + 1$ commit $[C^{(i)}_j, D^{(i)}_j] \leftarrow [H(k^{(i)}_j, \mathsf{View}^{(i)}_j), k^{(i)}_j || \mathsf{View}^{(i)}_j]$, and compute $g^{(i)}_j = G(k^{(i)}_j, \mathsf{View}^{(i)}_j)$.

   Choose random $C_{e^{(i)}+2}, g^{(i)}_{e^{(i)}+2}$

   Let $a^{(i)} = (y^{(i)}_1, y^{(i)}_2, y^{(i)}_3, C^{(i)}_1, C^{(i)}_2, C^{(i)}_3)$. And $h^{(i)} = g^{(i)}_1, g^{(i)}_2, g^{(i)}_3$.
2. Set the challenge: program $H(a^{(1)}, \ldots, a^{(t)}) := e$.
3. For each iteration r_i, $i \in [1,t]$: let $b^{(i)} = (y^{(i)}_{e^{(i)}+2}, C^{(i)}_{e^{(i)}+2})$ and set

$$z^{(i)} \leftarrow \begin{cases} (\mathsf{View}^{(i)}_2, k^{(i)}_1, k^{(i)}_2) & \text{if } e^{(i)} = 1, \\ (\mathsf{View}^{(i)}_3, k^{(i)}_2, k^{(i)}_3, x^{(i)}_3) & \text{if } e^{(i)} = 2, \\ (\mathsf{View}^{(i)}_1, k^{(i)}_3, k^{(i)}_1, x^{(i)}_3) & \text{if } e^{(i)} = 3. \end{cases}$$

4. Output $p \leftarrow [e, (b^{(1)}, z^{(1)}), (b^{(2)}, z^{(2)}), \cdots, (b^{(t)}, z^{(t)})]$.

Figure 3: The zero knowledge simulator

if we choose a random function H and a random output B, then run A^H_0 followed by A_C to generate x, and then run $A^H_1(x, B)$, that for a random j, the j-th query A^H_1 makes is measured as $x' = x$. Then as long as the output of A_C has collision-entropy at least k, the advantage with which A_1^H, when run after A_0, A_C as described, distinguishes (x, B) from $(x, H(x))$ is at most $(4+\sqrt{2})\sqrt{q_0}2^{-k/4} + 2q_1\sqrt{P_C}$.

In other words, if we can divide our game into three such algorithms and argue that the A_1 queries H on something that collapses to x with only negligible probability, then we can conclude that the two games are indistinguishable. Let A_0 run the game up until just before the i th iteration in the proof generation. Let A_C be the process which chooses $k_1^{(i)}, k_2^{(i)}, k_3^{(i)}$ and generates $\mathsf{View}_1^{(i)}, \mathsf{View}_2^{(i)}, \mathsf{View}_3^{(i)}$, and outputs $x = k_{e*^{(i)}}, \mathsf{View}_{e*^{(i)}}$. (Note that this has collision entropy $|k_{e*^{(i)}}|$ which is sufficient.) Let A_1 be the process which runs the rest of the proof, and then runs the adversary on the response.

Now we just have to argue that the probability that we make a measurement of A_1's j-th query to H and get x is negligible. To do this, we reduce to the security of the PRG used to generate the random tapes (and hence the views). Note that besides the one RO query, $k_{e*^{(i)}}$ is only used as input to the PRG. So, suppose there exists a quantum adversary A for which the resulting A_1 has non-negligible probability of making an H-query that collapses to x. Then we can construct a quantum attacker for the PRG: we run the above A_0, A_C, but instead of choosing $k_{e*^{(i)}}$ we use the PRG challenge as the resulting random tape, and return a random value as the RO output. Then we run A_1, which continues the proof (which should query $k_{e*^{(i)}}$ only with negligible probability since ks are chosen at random), and then runs the adversary. We pick a random j, and on the adversary's j-th RO query, we make a measurement and if it gives us a seed consistent with our challenge tape, we output 1, otherwise a random bit. If P_C is non-negligible then we will obtain the correct seed and distinguish with non-negligible probability.

Game 4: For each i instead of choosing random $k_{e*^{(i)}}$ and expanding it via the PRG to get the random tape used to compute the views, we choose those tapes directly at random.

Note that in Game 3, $k_{e*^{(i)}}$ are now only used as seeds for the PRG, so this follow from pseudo-randomness via a hybrid argument.

Game 5: We use the simulator to generate the views that will be opened, i.e. $j \neq e*^{(i)}$ for each i. We note that now the simulator no longer uses the witness.

This is identical by perfect privacy of the circuit decomposition.

Game 6: To allow for extraction in the simulation-extractability game we replace the random oracles with random polynomials whose degree is larger than the number of queries the adversary makes. The argument here identical to that in [Unr15].

Online Extractability. Before we prove online simulation-extractability, we define some notation to simplify the presentation:

For any proof $\pi = e, \{b^{(i)}, g^{(i)}, z^{(i)}\}_{i=1\ldots t}$, let $\mathsf{hash\text{-}input}(\pi) = \{a^{(i)}, h^{(i)} = (g_1^{(i)}, g_2^{(i)}, g_3^{(i)})\}$ be the values that the verifier uses as input to H_{chal} in the verification of π as described in Figure 4.

For a proof $\pi = (e, \{b^{(i)}, g^{(i)}, z^{(i)}\}_{i=1\ldots t})$, let $\mathsf{open}_0(z^{(i)}), \mathsf{open}_1(z^{(i)})$ denote the values derived from $z^{(i)}$ and used to compute $C_{e_i}^{(i)}$ and $C_{e_i+1}^{(i)}$ respectively in the description of Ver in Figure 4.

We say a tuple $(a, j, (o_1, o_2))$ is valid if $a = (y_1, y_2, y_3, C_1, C_2, C_3)$, $C_j = H_{\mathsf{com}}(o_1)$, $C_{j+1} = H_{\mathsf{com}}(o_2)$ and o_1, o_2 consist of k, View pairs for player $j, j+1$ that are consistent according to the circuit decomposition. We say $(a, j, (O_1, O_2))$ is set-valid if there exists $o_1 \in O_1$ and $o_2 \in O_2$ such that $(a, j, (o_1, o_2))$ is valid and set-invalid if not.

We first restate lemma 16 from [Unr15] tailored to our application, in particular the fact that our proofs do not explicitly contain the commitment but rather the information the verifier needs to recompute it.

Lemma 5.8. *Let q_G be the number of queries to G made by the adversary A and the simulator S in the simulation-extractability game, and let n be the number of proofs generated by S. Then the probability that A produces $x, \pi^* \notin$ simproofs where x, π^* is accepted by Ver^H, and $\mathsf{hash\text{-}input}(\pi^*) = \mathsf{hash\text{-}input}(\pi')$ for a previous proof π' produced by the simulator, is at most $n(n+1)/2(2^{-\kappa})^{3t} + O((q_G+1)^3 2^{-\kappa})$ (Call this event $\mathsf{MallSim}$.)*

Proof. This proof follows almost exactly as in [Unr15].

First, we argue that G is indistinguishable from a random function exactly as in [Unr15].

Then, observe that there are only two ways $\mathsf{MallSim}$ can occur:

Let e' be the hash value in π'. Then either S reprograms H sometime after π' is generated so that $H(\mathsf{hash\text{-}input}(\pi'))$ is no longer e', or π^* also contains the same e as π, i.e. $e = e'$. S only reprograms H if it chooses the same $\mathsf{hash\text{-}input}$ in a later proof - and $\mathsf{hash\text{-}input}$ includes $g_j^{(i)}$, i.e. a random function applied to an input which includes a randomly chosen seed. Thus, the probability that S chooses the same hash-input twice is at most $n(n+1)/2(2^{-\kappa})^{3t}+O((q_G+1)^3 2^{-\kappa}$, where the first term is the probability that two proofs use all the same seeds, and the second term is the probability that two different seeds result in a collision in G, where the latter follows from Theorem 8 in [Unr15].

The other possibility is that $\mathsf{hash\text{-}input}(\pi^*) = \mathsf{hash\text{-}input}(\pi')$, and $e = e'$, but $b^{(i)}, g^{(i)}, z^{(i)} \neq b'^{(i)}, g'^{(i)}, z'^{(i)}$ for some i. First note, that if $e = e'$ and $\mathsf{hash\text{-}input}(\pi^*) = \mathsf{hash\text{-}input}(\pi')$, then $g^{(i)} = g'^{(i)}$ and $b^{(i)} = b'^{(i)}$ for all i, by definition of hash-input. Thus, the only remaining possibility is that $z^{(i)} \neq z'^{(i)}$ for some i. But since $h^{(i)} = h'^{(i)}$ for all i, this implies a collision in G, which again by Theorem 8 in [Unr15] occurs with probability at most $O((q_G+1)^3 2^{-\kappa}$.

We conclude that MallSim occurs with probability at most

$$n(n+1)/2(2^{-\kappa})^{3t} + O((q_G+1)^3 2^{-\kappa}. \qquad \square$$

Here, next we present our variant of lemma 17 from [Unr15]. Note that this is quoted almost directly from Unruh with two modifications to account for the fact that our proofs do not explicitly contain the commitment but rather the information the verifier needs to recompute it, and the fact that our underlying Σ-protocol has only 3 challenges and satisfies 3-special soundness. H_0 in this lemma will correspond in our final proof to the initial state of H_{chal}, before any reprogramming.

Lemma 5.9. *Let G, H_{com} be arbitrarily distributed functions, and let $H_0 : \{0,1\}^{\leq \ell} \to \{0,1\}^{2t}$ be uniformly random (and independent of G). , Then, it is hard to find x and $\pi = e, \{b^{(i)}, g^{(i)}, z^{(i)}\}_{i=1\ldots t}$ such that for $\{a^{(i)}, (g_1^{(i)}, g_2^{(i)}, g_3^{(i)})\} = \mathsf{hash\text{-}input}(\pi)$ and $J_1 || \ldots || J_t := H_0(\mathsf{hash\text{-}input}(\pi))$*

(i) $g_{J_i}^{(i)} = G(\mathsf{open}_0(z^{(i)}))$ and $g_{J_i+1}^{(i)} = G(\mathsf{open}_1(z^{(i)}))$ for all i.

(ii) $(a^{(i)}, J_i, (\mathsf{open}_0(z^{(i)}), \mathsf{open}_1(z^{(i)})))$ is valid for all i.

(iii) For every i, there exists a j such that $(a^{(i)}, j, G^{-1}(g_{i,j}), G^{-1}(g_{i,j+1})))$ is set-invalid.

More precisely, if A^{G,H_0} makes at most q_H queries to H_0, it outputs (x, π) with these properties with probability at most $2(q_H+1)(\frac{2}{3})^{t/2}$

Proof. Without loss of generality, we can assume that G, H_{com} are fixed functions which A knows, so for this lemma we only treat H_0 as a random oracle.

For any given value of H_0, we call a tuple $c = (x, \{a^{(i)}\}_i, \{g_j^{(i)}\}_{i,j})$ a candidate iff: for each i, among the three transcripts, $(a^{(i)}, 1, G^{-1}(g_1)^{(i)}, G^{-1}(g_2^{(i)}))$, $(a^{(i)}, 2, G^{-1}(g_2^{(i)}),$
$G^{-1}(g_3^{(i)}))$, and $(a^{(i)}, 3, G^{-1}(g_3^{(i)}), G^{-1}(g_1^{(i)}))$ at least one is set-valid, and at least one is set-invalid. Let $n_{\mathsf{twovalid}}(c)$ be the number of i's for which there are 2 set-valid transcripts. Let $\mathsf{E}_{\mathsf{valid}}(c)$ be the set of challenge tuples which correspond to only set-valid conversations. (Note that $|\mathsf{E}_{\mathsf{valid}}(c)| = 2^{n_{\mathsf{twovalid}}(c)}$.) We call a candidate an H_0-*solution* if the challenge produced by H_0 only opens set-valid conversations, i.e. in lies in $\mathsf{E}_{\mathsf{valid}}(c)$. We now aim to prove that A^H outputs an H_0 solution with negligible probability.

For any given candidate c, for uniformly random H_0, the probability that c is an H_0-*solution* is $\leq (\frac{2}{3})^t$. In particular, for candidate c the probability is $(\frac{2}{3})^t * 2^{n_{\mathsf{twovalid}}(c)-t}$.

Let Cand be the set of all candidates. Let $F : \mathsf{Cand} \to \{0, 1\}$ be a random function such that for each c $F(c)$ is i.i.d. with $Pr[F(c) = 1] = (2/3)^t$.

Given F, we construct $H_F : \{0, 1\}^* \to \mathbb{Z}_3^t$ as follows:

- For each $c \notin \mathsf{Cand}$, $H_F(c)$ is set to a uniformly random $y \in \mathbb{Z}_3^t$.
- For each $c \in \mathsf{Cand}$ such that $F(c) = 0$, $H_F(c)$ is set to a uniformly random $y \in \mathbb{Z}_3^t \setminus \mathsf{E}_{\mathsf{valid}}(c)$.
- For each $c \in \mathsf{Cand}$ with $F(c) = 1$, with probability $2^{n_{\mathsf{twovalid}}-t}$, choose a random challenge tuple e from $\mathsf{E}_{\mathsf{valid}}(c)$, and set $H_F(c) := e$. Otherwise $H_F(c)$ is set to a uniformly random $y \in \mathbb{Z}_3^t \setminus \mathsf{E}_{\mathsf{valid}}(c)$.

Note that for each c, and e the probability of $H(c)$ being set to e is 3^{-t}. Suppose A_0^H outputs an H_0-solution with probability μ, then since H_F has the same distribution as H_0, $A^{H_F}()$ outputs an H_F solution c with probability μ. By our definition of H_F, if c is an H_F solution, then $F(c) = 1$. Thus, $A^{H_F}()$ outputs c such that $F(c) = 1$ with probability at least μ.

As in [Unr15], we can simulate $A^{H_F}()$ with another algorithm which generates H_F on the fly, and thus makes at most the same number of queries to F that A makes to H_F. Thus by applying Lemma 7 from [Unr15], we get

$$\mu \leq 2(q_H + 1)(\frac{2}{3})^{t/2}.$$

□

Finally, as the sigma protocol underlying our proofs is only computationally sound (because we use H_{com} for our commitment scheme), we need to argue that an extractor can extract from 3 valid transcripts with all but negligible probability.

Lemma 5.10. *There exists an extractor E_Σ such that for any PPT quantum adversary A, the probability that A can produce $(a, \{(\nu_{1,j}, \nu_{2,j})\}_{j=1,2,3})$ such that $(a, j, (\nu_{1,j}, \nu_{2,j}))$ is a valid transcript for $j = 1, 2, 3$, but $E_\Sigma(a, \{(\nu_{1,j}, \nu_{2,j})\}_{j=1,2,3})$ fails to extract a proof, is negligible.*

Proof. Recall that $a = (y_1, y_2, y_3, C_1, C_2, C_3)$, and if all three transcripts are valid, $C_j = H_{\mathsf{com}}(\nu_{1,j}) = H_{\mathsf{com}}(\nu_{2,j-1})$ for $j = 1, 2, 3$. Thus, either we have $\nu_{1,j} = \nu_{2,j-1}$ for all j or $\mathcal{A}$ has found a collision in H_{com}. But, Theorem 8 in [Unr15] tells us that the probability of finding a collision in a random function with k-bit output using at most q queries is at most $O((q+1)^3 2^{-k})$, which is negligible. If $\nu_{1,j} = \nu_{2,j-1}$ for all j, then we have 3 $k_j || \mathsf{View}_j$ values, all of which are pairwise consistent, so we conclude by the correctness of the circuit decomposition, and the fact that $(x = y, w) \in R$ iff $\phi(w) = y$ that if we sum the input share in $\mathsf{View}_1, \mathsf{View}_2, \mathsf{View}_3$, we get a witness such that $(x, w) \in R$. □

Theorem 5.11. *Our version of the Unruh protocol satisfies simulation-extractability against a quantum adversary.*

Proof. We define the following extractor:

1. On input π, compute $\mathsf{hash\text{-}input}(\pi) = \{a^{(i)}, h^{(i)} = (g_1^{(i)}, g_2^{(i)}, g_3^{(i)})\}$
2. For $i \in 1, \ldots, t$: For $j \in 1, 2, 3$, check whether there exists $\nu_{1,j} \in G^{-1}(g_j^{(i)}), \nu_{2,j} \in G^{-1}(g_{j+1}^{(i)})$ such that $(a^{(i)}, j, (\nu_{1,j}, \nu_{2,j}))$ is a valid transcript. If there is a valid transcript for all j, output $E_\Sigma(a^{(i)}, \{(\nu_{1,j}, \nu_{2,j})\}_{j=1,2,3})$ as defined by Lemma 5.10 and halt.
3. If no solution is found, output $\perp$.

First we define some notation, again borrowed heavily from [Unr15]:

Let $\mathsf{Ev}_i, \mathsf{Ev}_{ii}, \mathsf{Ev}_{iii}$ be events denoting that A in the simulation-extractability game produces a proof satisfying conditions *(i)*, *(ii)*, and *(iii)* from Lemma 5.9 respectively.

Let $\mathsf{SigExtFail}$ be the event that the extractor finds a successful $(a, \{(\nu_{1,j}, \nu_{2,j})\}_{j=1,2,3})$, but E_Σ fails to produce a valid witness.

Let $\mathsf{ShouldExt}$ denote the event that A produces x, π such that Ver^H accepts and $(x, \pi) \notin$ *simproofs*.

Then our goal is to prove that the w produced by the extractor is such that $(x, w) \in R$. I.e., we want to prove that the following probability is negligible.

$$\begin{aligned}
&\Pr[\mathsf{ShouldExt} \wedge (x,w) \notin R] \\
&\quad \leq \Pr[\mathsf{ShouldExt} \wedge (x,w) \notin R \wedge \neg\mathsf{MallSim}] + \Pr[\mathsf{MallSim}] \\
&\quad = \Pr[\mathsf{ShouldExt} \wedge (x,w) \notin R \wedge \neg\mathsf{MallSim} \wedge \neg\mathsf{Ev}_{iii}] \\
&\quad\qquad + \Pr[\mathsf{ShouldExt} \wedge (x,w) \notin R \wedge \neg\mathsf{MallSim} \wedge \mathsf{Ev}_{iii}] + \Pr[\mathsf{MallSim}] \\
&\quad \leq \Pr[(x,w) \notin R \wedge \neg\mathsf{Ev}_{iii}] \\
&\quad\qquad + \Pr[\mathsf{ShouldExt} \wedge (x,w) \notin R \wedge \neg\mathsf{MallSim} \wedge \mathsf{Ev}_{iii}] + \Pr[\mathsf{MallSim}] \\
&\quad = \Pr[\mathsf{SigExtFail}] \\
&\quad\qquad + \Pr[\mathsf{ShouldExt} \wedge (x,w) \notin R \wedge \neg\mathsf{MallSim} \wedge \mathsf{Ev}_{iii}] + \Pr[\mathsf{MallSim}] \\
&\quad = \Pr[\mathsf{SigExtFail}] \\
&\quad\qquad + \Pr[\mathsf{ShouldExt} \wedge (x,w) \notin R \wedge \neg\mathsf{MallSim} \wedge \mathsf{Ev}_{i} \wedge \mathsf{Ev}_{ii} \wedge \mathsf{Ev}_{iii}] \\
&\quad\qquad + \Pr[\mathsf{MallSim}] \\
&\quad \leq \Pr[\mathsf{SigExtFail}] + \Pr[\mathsf{Ev}_{i} \wedge \mathsf{Ev}_{ii} \wedge \mathsf{Ev}_{iii}] + \Pr[\mathsf{MallSim}]
\end{aligned}$$

Here, the second equality follows from the definition of $\mathsf{SigExtFail}$ and Ev_{iii}, and the description of the extractor. The third equality follows from the fact that $\neg\mathsf{MallSim}$ means that the hash function on $\mathsf{hash\text{-}input}(\pi)$ has not been reprogrammed, and the fact that $\mathsf{ShouldExt}$ means verification succeeds, which means that conditions *(i)* and *(ii)* are satisfied.

Finally, by Lemmas 5.10, 5.9, and 5.8, we conclude that this probability is negligible.

5.4 Strong Unforgeability of Picnic-FS and Picnic-UR

We have shown that Picnic-FS and Picnic-UR are a simulation-extractable NIZK proof systems in the classical (resp. quantum) random oracle model against classical (resp. quantum) adversaries. Strong unforgeability (sEUF-CMA security) follows directly: this is a well known result in the classical model, and shown in [Unr15] in the quantum setting. For completeness, we briefly sketch this result:

Suppose there exist an adversary $\mathcal{A}$ who can break the strong unforgeability property (cf. Definition 2.8). Then we can construct an adversary against the ZK property of the NIZK, the simulation-extractability property of the NIZK, or the one-wayness of the one-way function. We proceed through a series of games. In the first transition, we switch the signature algorithm to use the ZK simulator rather than the prover. This is indistinguishable by ZK, so the adversary will still produce forgeries with high probability, or we have a distinguisher which breaks the ZK property. Then, when the adversary produces a valid forgery, we run the extractor to produce a preimage of the one-way function. If this extractor does not succeed with high probability whenever the adversary produces a forgery, we break simulation-extractability. Note here, that our extractor is guaranteed to work as soon as either the statement or the proof is different from what the simulator produced, so we will be able to extract from new signatures on previously signed messages as required in strong unforgeability. Otherwise, we have produced a pre-image given only the output of the one-way function (recall that we use the simulator to sign, so we do not need the pre-image there), so we break the one-wayness property.

6 Formal Security Analysis of Picnic2

In this section we provide a formal analysis of the Picnic2 signature schemes, starting with the security of the underlying MPC protocol, then proving unforgeability in the random oracle model. We end with a brief analysis of two optimizations and discuss security in the quantum random oracle model.

6.1 Proof of Security of the Underlying MPC Protocol

The protocol is described in Section 2.9.1, here we prove it is secure against an all-but-one corruption in the semi-honest model.

Lemma 6.1. *Suppose there exists a (t, ϵ_{PRG})-PRG. Then there exists a simulator for the MPC protocol of §2.9.1 such that no distinguisher running in time t can distinguish between the real-world execution and ideal-world execution defined by this simulator with better than ϵ_{PRG} probability.*

Proof. We first describe a simulator $\mathsf{Sim}_P(1^\kappa, y, C)$ that outputs the view of all parties except for P. Denote the input and output sizes of C by m and l respectively. We use $x \leftarrow X$ to denote choosing a value from X at random and assigning it to x. The simulator works as follows:

1. If $P = n$, set $\mathsf{state}_i \leftarrow \{0,1\}^k$ for all $i \neq P$. Otherwise, set $\mathsf{state}_i \leftarrow \{0,1\}^k$, for $i \notin \{n, P\}$ and set $\mathsf{state}_n \leftarrow \{0,1\}^{k+|C|}$.
2. Pick $\hat{z} \leftarrow \{0,1\}^m$, $\mathsf{msgs}_P \leftarrow \{0,1\}^{|C|}$.
3. Use $\{\mathsf{state}_i\}_{i \neq P}$, $\hat{z}$ and msgs_P to simulate the online phase of the MPC protocol until the output reconstruction step, such that the simulator obtains the shares of outputs $[y]$ for $i \neq P$, denoted as $[y]_i$. Compute $[y]_P := \bigoplus_{i \neq P} [y]_i \oplus y$. Append $[y]_P$ to msgs_P.

Hybrid$_\mathbf{1}$. Same as the real-world protocol, except use true randomness, instead of seed-derived, for party P. String aux is computed as described in the protocol, based on the true randomness.

It is easy to see that the probability of distinguishing **Hybrid$_\mathbf{1}$** and the real-world protocol in running time t is no more than ϵ_{PRG}.

Hybrid$_\mathbf{2}$. Replace aux in **Hybrid$_\mathbf{1}$** by a uniformly random string of the same length.

If $P = n$, then aux is not part of the view of the adversary; if $P \neq n$, then bits of aux are computed by XORing one bit of randomness from each seed from party $i \neq P$, then XORing one bit of randomness from party P (which is uniformly random in **Hybrid$_\mathbf{1}$**). Therefore aux is uniformly random in **Hybrid$_\mathbf{1}$**.

Therefore, **Hybrid$_\mathbf{1}$** and **Hybrid$_\mathbf{2}$** are identical.

Hybrid$_\mathbf{3}$. Same as **Hybrid$_\mathbf{2}$**, except that $\hat{z}$ is changed to uniformly random string; The last message from party P is replaced by a message computed from the output as defined in the simulator. In more detail, use $\{\mathsf{state}_i\}_{i \neq P}$, $\hat{z}$ and msgs_P to simulate the online phase of the MPC protocol locally, such that in the end, the simulator obtains share of outputs $[y]_i$ for $i \neq P$. Compute $[y]_P := \bigoplus_{i \neq P} [y]_i \oplus y$. Replace the last message from party P for reconstructing the output to $[y]_P$.

It is easy to see that $\hat{z}$ is uniformly random in both hybrids since the share of the mask held by party P is uniformly random. $[y]_P$ is identically distributed in the two hybrids given the perfect correctness of the protocol: in both worlds, $[y]_P$ is a deterministic function of the output y and the messages send by parties other than P.

Therefore, $\mathbf{Hybrid_3}$ and $\mathbf{Hybrid_2}$ are identical.

□

6.2 Security Proof of the Signature Scheme

In this section, we give a dedicated proof of security for the signature scheme constructed by making the KKW proof protocol non-interactive with the Fiat-Shamir transform. In doing so, our goals are both to give a complete proof (taking into account certain optimizations mentioned in the text), as well as to highlight the concrete-security bound we obtain. The theorem below proves EUF-CMA security, we believe it can be generalized to strong unforgeability.

In this section κ is a security parameter, and G is a hash function modeled as a random oracle (different from the permutation G used in Picnic-UR). We abstract our scheme by assuming that the key-generation algorithm Gen outputs a pair (C, w) with $C(w) = 1$, where we view C as the public key and w as the private key. We assume $|C| \geq \kappa$ and $w \in \{0,1\}^\kappa$. Our hardness assumption is that, given C as output by Gen, it is hard to find w' for which $C(w') = 1$. More formally, we say that Gen is (t, ϵ)-*one way* if for all adversaries $\mathcal{A}$ running in time at most t we have

$$\Pr[(C, w) \leftarrow \mathsf{Gen}; w' \leftarrow \mathcal{A}(C) : C(w') = 1] \leq \epsilon.$$

Theorem 6.2. *Suppose the PRG used is (t, ϵ_{PRG})-secure,* Gen *is (t, ϵ_{OW})-one-way, and Π is the MPC protocol described in Section 2.9.1. Model H_0, H_1, H_2, and G as random oracles where H_0, H_1, H_2 have 2κ-bit output length. Then any attacker carrying out an adaptive chosen-message attack on* Picnic2 *(Figure 4), running in time t, making q_s signing queries, and making q_0, q_1, q_2, q_G queries, respectively, to the random oracles, succeeds in outputting a valid forgery with probability at most*

$$\Pr[\mathsf{Forge}] \leq O(q_s \cdot \tau \cdot \epsilon_{PRG}) + O\left(\frac{(q_0 + q_1 + q_2 + Mnq_s)^2}{2^{2\kappa}}\right) + \epsilon_{OW} + q_G \cdot \epsilon(M, n, \tau),$$

where

$$\epsilon(M, n, \tau) = \max_{M-\tau \leq k \leq M} \left\{ \frac{\binom{k}{M-\tau}}{\binom{M}{M-\tau} \cdot n^{k-M+\tau}} \right\}.$$

Picnic2 Signing

Keys: The public key is a circuit C; the private key is a value w for which $C(w) = 1$. Values M, n, τ are parameters of the protocol.

To sign message m, the signer does the following.

Step 1 For each $j \in [M]$:

1. Choose uniform $\mathsf{seed}_j^* \in \{0,1\}^\kappa$ and use it to generate values $\mathsf{seed}_{j,1}, \ldots, \mathsf{seed}_{j,n}$ with a PRG. Also compute $\mathsf{aux}_j \in \{0,1\}^{|C|}$ as described in the text. For $i = 1, \ldots, n-1$, let $\mathsf{state}_{j,i} := \mathsf{seed}_{j,i}$; let $\mathsf{state}_{j,n} := \mathsf{seed}_{j,n} \| \mathsf{aux}_j$.
2. For $i \in [n]$, compute $\mathsf{com}_{j,i} := H_0(\mathsf{state}_{j,i})$.
3. The signer runs the online phase of the n-party protocol Π (as described in the text) using $\{\mathsf{state}_{j,i}\}_i$, beginning by computing the masked inputs $\{\hat{z}_{j,\alpha}\}$ (based on w and the $\{\lambda_{j,\alpha}\}$ defined by the preprocessing). Let $\mathsf{msgs}_{j,i}$ denote the messages broadcast by S_i in this protocol execution.
4. Let $h_j := H_1(\mathsf{com}_{j,1}, \ldots, \mathsf{com}_{j,n})$ and let $h'_j := H_2(\{\hat{z}_{j,\alpha}\}, \mathsf{msgs}_{j,1}, \ldots, \mathsf{msgs}_{j,n})$.

Step 2 Compute $(\mathcal{C}, \mathcal{P}) := G(m, h_1, h'_1, \ldots, h_M, h'_M)$, where $\mathcal{C} \subset [M]$ is a set of size τ, and $\mathcal{P}$ is a list $\{p_j\}_{j \in \mathcal{C}}$ with $p_j \in [n]$. The signature includes $(\mathcal{C}, \mathcal{P})$.

Step 3 For each $j \in [M] \setminus \mathcal{C}$, the signer includes seed_j^*, h'_j in the signature. Also, for each $j \in \mathcal{C}$, the signer includes $\{\mathsf{state}_{j,i}\}_{i \neq p_j}$, com_{j,p_j}, $\{\hat{z}_{j,\alpha}\}$, and msgs_{j,p_j} in the signature.

Figure 4: The signing algorithm in the Picnic2 signature scheme.

Picnic2 Verification

A signature $(\mathcal{C}, \mathcal{P}, \{\mathsf{seed}^*_j, h'_j\}_{j \notin \mathcal{C}}, \{\{\mathsf{state}_{j,i}\}_{i \neq p_j}, \mathsf{com}_{j,p_j}, \{\hat{z}_{j,\alpha}\}, \mathsf{msgs}_{j,p_j}\}_{j \in \mathcal{C}})$ on a message m is verified as follows:

1. For every $j \in \mathcal{C}$ and $i \neq p_j$, set $\mathsf{com}_{j,i} := H_0(\mathsf{state}_{j,i})$; then compute the value $h_j := H_1(\mathsf{com}_{j,1}, \ldots, \mathsf{com}_{j,n})$.
2. For $j \notin \mathcal{C}$, use seed^*_j to compute h_j as the signer would.
3. For each $j \in \mathcal{C}$, run an execution of Π among the parties $\{S_i\}_{i \neq p_j}$ using $\{\mathsf{state}_{j,i}\}_{i \neq p_j}$, $\{\hat{z}_\alpha\}$, and msgs_{j,p_j}; this yields $\{\mathsf{msgs}_i\}_{i \neq p_j}$ and an output bit b. Check that $b \stackrel{?}{=} 1$. Then compute $h'_j := H_2(\{\hat{z}_{j,\alpha}\}$ $\mathsf{msgs}_{j,1}, \ldots, \mathsf{msgs}_{j,n})$.
4. Check that $(\mathcal{C}, \mathcal{P}) \stackrel{?}{=} G(m, h_1, h'_1, \ldots, h_M, h'_M)$.

Figure 5: The verification algorithm in the Picnic2 signature scheme.

In the Picnic2 specification, the parameters (M, n, τ) are chosen such that $\epsilon(M, n, \tau) \leq 2^\kappa$.

Proof. Fix some attacker $\mathcal{A}$. Let q_s denote the number of signing queries made by $\mathcal{A}$; let q_0, q_1, q_2, respectively, denote the number of queries to H_0, H_1, H_2 made by $\mathcal{A}$, and let q_G denote the number of queries to G made by $\mathcal{A}$. To prove security we define a sequence of experiments involving $\mathcal{A}$, where the first corresponds to the experiment in which $\mathcal{A}$ interacts with the real signature scheme. We let $\Pr_i[\cdot]$ refer to the probability of an event in experiment i. We let t denote the running time of the entire experiment, i.e., including both $\mathcal{A}$'s running time and the time required to answer signing queries and to verify $\mathcal{A}$'s output.

Experiment 1. This corresponds to the interaction of $\mathcal{A}$ with the real signature scheme. In more detail: first Gen is run to obtain (C, w), and $\mathcal{A}$ is given the public key C. In addition, we assume the random oracles H_0, H_1, H_2, and G are chosen uniformly from the appropriate spaces. $\mathcal{A}$ may make signing queries, which will be answered as in Figure 4; $\mathcal{A}$ may also query any of the random oracles. Finally, $\mathcal{A}$ outputs a message/signature pair; we let Forge denote the event that the message was not previously queried by $\mathcal{A}$ to its signing oracle, and the signature is valid. We are interested in upper-bounding $\Pr_1[\mathsf{Forge}]$.

Experiment 2. We abort the experiment if, during the course of the experiment, a collision in H_0, H_1, or H_2 is found. Suppose $q = \max\{q_0, q_1, q_2\}$, then the number of queries to any oracle throughout the experiment (by either the adversary or the signing algorithm) is at most $(q + Mnq_s)$. Thus,

$$|\Pr_1[\mathsf{Forge}] - \Pr_2[\mathsf{Forge}]| \leq \frac{3(q + Mnq_s)^2}{2^{2\kappa}}.$$

Experiment 3. Here we modify the way signing is done. Specifically, when signing a message m we begin by choosing $(\mathcal{C}, \mathcal{P})$ uniformly. Steps 1 and 3 of the signing algorithm are computed as before, but in step 2 we simply set the output of G equal to $(\mathcal{C}, \mathcal{P})$. Formally, a signature on a message m is now computed as follows:

Step 0 Choose uniform $(\mathcal{C}, \mathcal{P})$, where $\mathcal{C} \subset [M]$ is a set of size τ, and $\mathcal{P} = \{p_j\}_{j \in \mathcal{C}}$ with $p_j \in [n]$.

Step 1 For each $j \in [M]$:

1. Choose uniform $\mathsf{seed}_j^* \in \{0,1\}^\kappa$ and use it to generate values $\mathsf{seed}_{j,1}$, …, $\mathsf{seed}_{j,n}$ and $\mathsf{aux}_j \in \{0,1\}^{|C|}$. For $i = 1, \ldots, n-1$, let $\mathsf{state}_{j,i} :=$ $\mathsf{seed}_{j,i}$; let $\mathsf{state}_{j,n} := \mathsf{seed}_{j,n} \| \mathsf{aux}_j$.
2. For $i \in [n]$, compute $\mathsf{com}_{j,i} := H_0(\mathsf{state}_{j,i})$.
3. Run the online phase of the n-party protocol Π using $\{\mathsf{state}_{j,i}\}_i$, beginning by computing the masked inputs $\{\hat{z}_{j,\alpha}\}$ (based on w and the $\{\lambda_{j,\alpha}\}$ defined by the preprocessing). Let $\mathsf{msgs}_{j,i}$ denote the messages broadcast by S_i in this protocol execution.
4. Let $h_j := H_1(\mathsf{com}_{j,1}, \ldots, \mathsf{com}_{j,n})$ and $h'_j := H_2(\{\hat{z}_{j,\alpha}\}, \mathsf{msgs}_{j,1}, \ldots, \mathsf{msgs}_{j,n})$.

Step 2 Set $G(m, h_1, h'_1, \ldots, h_M, h'_M)$ equal to $(\mathcal{C}, \mathcal{P})$. (I.e., if $\mathcal{A}$ subsequently makes the query $G(m, h_1, h'_1, \ldots, h_M, h'_M)$, return $(\mathcal{C}, \mathcal{P})$ as the output.) Include $(\mathcal{C}, \mathcal{P})$ in the signature.

Step 3 For each $j \in [M] \setminus \mathcal{C}$, the signer includes seed_j^*, h'_j in the signature. Also, for each $j \in \mathcal{C}$, the signer includes $\{\mathsf{state}_{j,i}\}_{i \neq p_j}$, com_{j,p_j}, $\{\hat{z}_{j,\alpha}\}$, and msgs_{j,p_j} in the signature.

The only difference between this experiment and the previous one occurs if, in the course of answering a signing query, the query to G in step 2 was ever made before (by either the adversary or as part of answering some other signing query). Letting $\mathsf{InputColl}_G$ denote this event, we have

$$|\Pr_3[\mathsf{Forge}] - \Pr_2[\mathsf{Forge}]| \leq \Pr_3[\mathsf{InputColl}_G].$$

Experiment 4. Here we again modify the way signing is done. Now, the signer chooses uniform $\{\mathsf{seed}_{j,i}\}_{i=1}^{n}$ for all $j \in \mathcal{C}$. That is, signatures are now computed as follows:

Step 0 Choose uniform $(\mathcal{C}, \mathcal{P})$, where $\mathcal{C} \subset [M]$ is a set of size τ, and $\mathcal{P} = \{p_j\}_{j \in \mathcal{C}}$ with $p_j \in [n]$.

Step 1 For each $j \in [M]$:

1. If $j \notin \mathcal{C}$, choose uniform $\mathsf{seed}_j^* \in \{0,1\}^{\kappa}$ and use it to generate values $\mathsf{seed}_{j,1}$, …, $\mathsf{seed}_{j,n}$. If $j \in \mathcal{C}$, choose uniform $\mathsf{seed}_{j,1}$, …, $\mathsf{seed}_{j,n} \in \{0,1\}^{\kappa}$.
2. Compute $\mathsf{aux}_j \in \{0,1\}^{|C|}$ based on $\{\mathsf{seed}_{j,i}\}_i$. For $i = 1, \ldots, n-1$, let $\mathsf{state}_{j,i} := \mathsf{seed}_{j,i}$; let $\mathsf{state}_{j,n} := \mathsf{seed}_{j,n} \| \mathsf{aux}_j$.
3. For $i \in [n]$, compute $\mathsf{com}_{j,i} := H_0(\mathsf{state}_{j,i})$.
4. Run the online phase of the n-party protocol Π using $\{\mathsf{state}_{j,i}\}_i$, beginning by computing the masked inputs $\{\hat{z}_{j,\alpha}\}$ (based on w and the $\{\lambda_{j,\alpha}\}$ defined by the preprocessing). Let $\mathsf{msgs}_{j,i}$ denote the messages broadcast by S_i in this protocol execution.
5. Let $h_j := H_1(\mathsf{com}_{j,1}, \ldots, \mathsf{com}_{j,n})$ and $h'_j := H_2(\{\hat{z}_{j,\alpha}\}, \mathsf{msgs}_{j,1}, \ldots, \mathsf{msgs}_{j,n})$.

Step 2 Set $G(m, h_1, h'_1, \ldots, h_M, h'_M)$ equal to $(\mathcal{C}, \mathcal{P})$. (I.e., if $\mathcal{A}$ subsequently makes the query $G(m, h_1, h'_1, \ldots, h_M, h'_M)$, return $(\mathcal{C}, \mathcal{P})$ as the output.) Include $(\mathcal{C}, \mathcal{P})$ in the signature.

Step 3 For each $j \notin \mathcal{C}$, include seed_j^*, h'_j in the signature. For each $j \in \mathcal{C}$, include $\{\mathsf{state}_{j,i}\}_{i \neq p_j}$, com_{j,p_j}, $\{\hat{z}_{j,\alpha}\}$, and msgs_{j,p_j} in the signature.

It is easy to see that if the pseudorandom generator is (t, ϵ_{PRG})-secure, then

$$|\Pr_4[\mathsf{Forge}] - \Pr_3[\mathsf{Forge}]| \leq q_s \cdot \tau \cdot \epsilon_{PRG}$$

and

$$|\Pr_4[\mathsf{InputColl}_G] - \Pr_3[\mathsf{InputColl}_G]| \le q_s \cdot \tau \cdot \epsilon_{PRG}.$$

We now bound $\Pr_4[\mathsf{InputColl}_G]$. Fix some previous query $(m, h_1, h'_1, \ldots, h_M, h'_M)$ to G, and look at a query $G(\hat{m}, \hat{h}_1, \hat{h}'_1, \ldots, \hat{h}_M, \hat{h}'_M)$ made while responding to some signing query. (In the rest of this discussion, we will use $\hat{\cdot}$ to represent values computed as part of answering that signing query.) For some fixed $j \in \hat{\mathcal{C}}$, it is not hard to see that the probability of the event $\hat{h}_j = h_j$ is maximized if h_j was output by a previous query $H_1(\mathsf{com}_1, \ldots, \mathsf{com}_n)$, and each com_i was output by a previous query $H_0(\mathsf{state}_i)$. (In all cases, the relevant prior query must be unique since the experiment is aborted if there is a collision in H_0 or H_1.) In that case, the probability that $\hat{h}_j = h_j$ is at most

$$(2^{-\kappa} + 2^{-2\kappa})^n + 2^{-2\kappa} \le 2 \cdot 2^{-2\kappa}$$

(assuming $n \ge 3$), and thus the probability that $\hat{h}_j = h_j$ for all $j \in \hat{\mathcal{C}}$ is at most $2^{-\tau \cdot (2\kappa - 1)}$. Taking a union bound over all signing queries and all queries made to G (including those made during the course of answering signing queries), we conclude that

$$\Pr_4[\mathsf{InputColl}_G] \le q_s \cdot (q_s + q_G) \cdot 2^{-\tau \cdot (2\kappa - 1)}.$$

Experiment 5. Here we again modify the way signing is done. Now:

- For each $j \in \mathcal{C}$, choose uniform com_{j,p_j} (i.e., without making the corresponding query to H_0).
- For each $j \notin \mathcal{C}$, choose uniform h'_j (i.e., without making the corresponding query to H_2).

So, signatures are now computed as follows:

Step 0 Choose uniform $(\mathcal{C}, \mathcal{P})$, where $\mathcal{C} \subset [M]$ is a set of size τ, and $\mathcal{P} = \{p_j\}_{j \in \mathcal{C}}$ with $p_j \in [n]$.

Step 1 For each $j \in [M]$:

1. If $j \notin \mathcal{C}$, choose uniform $\mathsf{seed}^*_j \in \{0,1\}^\kappa$ and use it to generate values $\mathsf{seed}_{j,1}, \ldots, \mathsf{seed}_{j,n}$. If $j \in \mathcal{C}$, choose uniform $\mathsf{seed}_{j,1}, \ldots, \mathsf{seed}_{j,n} \in \{0,1\}^\kappa$.

2. Compute $\mathsf{aux}_j \in \{0,1\}^{|C|}$ based on $\{\mathsf{seed}_{j,i}\}_i$. For $i = 1, \ldots, n-1$, let $\mathsf{state}_{j,i} := \mathsf{seed}_{j,i}$; let $\mathsf{state}_{j,n} := \mathsf{seed}_{j,n} \| \mathsf{aux}_j$.
3. For $j \in \mathcal{C}$, choose uniform $\mathsf{com}_{j,p_j} \in \{0,1\}^{2\kappa}$. For all other j, i, set $\mathsf{com}_{j,i} := H_0(\mathsf{state}_{j,i})$.
4. Run the online phase of the n-party protocol Π using $\{\mathsf{state}_{j,i}\}_i$, beginning by computing the masked inputs $\{\hat{z}_{j,\alpha}\}$ (based on w and the $\{\lambda_{j,\alpha}\}$ defined by the preprocessing). Let $\mathsf{msgs}_{j,i}$ denote the messages broadcast by S_i in this protocol execution.
5. Let $h_j := H_1(\mathsf{com}_{j,1}, \ldots, \mathsf{com}_{j,n})$. If $j \in \mathcal{C}$, set $h'_j := H_2(\{\hat{z}_{j,\alpha}\}, \mathsf{msgs}_{j,1}, \ldots, \mathsf{msgs}_{j,n})$; otherwise, choose uniform $h'_j \in \{0,1\}^{2\kappa}$.

Step 2 Set $G(m, h_1, h'_1, \ldots, h_M, h'_M)$ equal to $(\mathcal{C}, \mathcal{P})$. (I.e., if $\mathcal{A}$ subsequently makes the query $G(m, h_1, h'_1, \ldots, h_M, h'_M)$, return $(\mathcal{C}, \mathcal{P})$ as the output.) Include $(\mathcal{C}, \mathcal{P})$ in the signature.

Step 3 For each $j \notin \mathcal{C}$, include seed^*_j, h'_j in the signature. For each $j \in \mathcal{C}$, include $\{\mathsf{state}_{j,i}\}_{i \neq p_j}$, com_{j,p_j}, $\{\hat{z}_{j,\alpha}\}$, and msgs_{j,p_j} in the signature.

The only difference between this experiment and the previous one occurs if, during the course of answering a signing query, state_{j,p_j} (for some $j \in \mathcal{C}$) is queried to H_0 at some other point in the experiment, or $(\{\hat{z}_{j,\alpha}\}, \mathsf{msgs}_{j,1}, \ldots, \mathsf{msgs}_{j,n})$ (for some $j \notin \mathcal{C}$) is ever queried to H_2 at some other point in the experiment. Denoting this event by $\mathsf{InputColl}_H$, we thus have

$$|\Pr_5[\mathsf{Forge}] - \Pr_4[\mathsf{Forge}]| \leq \Pr_5[\mathsf{InputColl}_H].$$

Experiment 6. We again modify the signing algorithm. Now, for $j \in \mathcal{C}$ the signer uses the simulator for Π (namely, Sim_Π) to generate the views of the parties $\{S_i\}_{i \neq p_j}$ in an execution of Π when evaluating C with output 1. This results in values $\{\mathsf{state}_{j,i}\}_{i \neq p_j}$, masked input-wire values $\{\hat{z}_{j,\alpha}\}$, and msgs_{j,p_j}. From the respective views, $\{\mathsf{msgs}_{j,i}\}_{i \neq p_j}$ can be computed, and h_j, h'_j can be computed as well. Thus, signatures are now computed as follows:

Step 0 Choose uniform $(\mathcal{C}, \mathcal{P})$, where $\mathcal{C} \subset [M]$ is a set of size τ, and $\mathcal{P} = \{p_j\}_{j \in \mathcal{C}}$ with $p_j \in [n]$.

Step 1 For $j \notin \mathcal{C}$:

1. Choose uniform $\mathsf{seed}^*_j \in \{0,1\}^\kappa$ and use it to generate values $\mathsf{seed}_{j,1}$, …, $\mathsf{seed}_{j,n}$. Compute $\mathsf{aux}_j \in \{0,1\}^{|C|}$ based on $\{\mathsf{seed}_{j,i}\}_i$. For $i = 1, \ldots, n-1$, let $\mathsf{state}_{j,i} := \mathsf{seed}_{j,i}$; let $\mathsf{state}_{j,n} := \mathsf{seed}_{j,n} \| \mathsf{aux}_j$.
2. For all i, set $\mathsf{com}_{j,i} := H_0(\mathsf{state}_{j,i})$.
3. Let $h_j := H_1(\mathsf{com}_{j,1}, \ldots, \mathsf{com}_{j,n})$. Choose uniform $h'_j \in \{0,1\}^{2\kappa}$.

For each $j \in \mathcal{C}$:

1. Compute $(\{\mathsf{state}_{j,i}\}_{i \neq p_j}, \{\hat{z}_{j,\alpha}\}, \mathsf{msgs}_{j,p_j}) \leftarrow \mathsf{Sim}_\Pi(p_j)$. Compute $\{\mathsf{msgs}_{j,i}\}_{i \neq p_j}$ based on this information.
2. Choose uniform $\mathsf{com}_{j,p_j} \in \{0,1\}^{2\kappa}$. For all other i, set $\mathsf{com}_{j,i} := H_0(\mathsf{state}_{j,i})$.
3. Let $h_j := H_1(\mathsf{com}_{j,1}, \ldots, \mathsf{com}_{j,n})$ and $h'_j := H_2(\{\hat{z}_{j,\alpha}\}, \mathsf{msgs}_{j,1}, \ldots, \mathsf{msgs}_{j,n})$.

Step 2 Set $G(m, h_1, h'_1, \ldots, h_M, h'_M)$ equal to $(\mathcal{C}, \mathcal{P})$. (I.e., if $\mathcal{A}$ subsequently makes the query $G(m, h_1, h'_1, \ldots, h_M, h'_M)$, return $(\mathcal{C}, \mathcal{P})$ as the output.) Include $(\mathcal{C}, \mathcal{P})$ in the signature.

Step 3 For each $j \notin \mathcal{C}$, the signer includes seed^*_j, h'_j in the signature. Also, for each $j \in \mathcal{C}$, the signer includes $\{\mathsf{state}_{j,i}\}_{i \neq p_j}$, com_{j,p_j}, $\{\hat{z}_{j,\alpha}\}$, and msgs_{j,p_j} in the signature.

Observe that w is no longer used for generating signatures. Recall, the adversary in the underlying MPC protocol Π has distinguishing advantage ϵ_{PRG} (see Lemma 6.1). It is immediate that

$$|\Pr_6[\mathsf{Forge}] - \Pr_5[\mathsf{Forge}]| \leq \tau \cdot q_s \cdot \epsilon_{PRG}$$

and

$$|\Pr_6[\mathsf{InputColl}_H] - \Pr_5[\mathsf{InputColl}_H]| \leq \tau \cdot q_s \cdot \epsilon_{PRG}.$$

We now bound $\Pr_6[\mathsf{InputColl}_H]$. For any particular signing query and any $j \in \mathcal{C}$, the value state_{j,p_j} has min-entropy at least κ and is not used anywhere else in the experiment. Similarly, for any $j \notin \mathcal{C}$, the value $\{\hat{z}_{j,\alpha}\}$ has min-entropy at least κ, since the input is κ-bit and they are all uniform according to the simulator defined in the next section. and is not used anywhere else in the experiment. Thus,

$$\Pr_6[\mathsf{InputColl}_H] \leq M \cdot q_s \cdot (Mq_s + q_0 + q_2) \cdot 2^{-\kappa}.$$

Experiment 7. We first define some notation. At any point during the experiment, we classify a pair (h, h') in one of the following ways:

1. If h was output by a previous query $H_1(\mathsf{com}_1, \ldots, \mathsf{com}_n)$, and each com_i was output by a previous query $H_0(\mathsf{state}_i)$ where the $\{\mathsf{state}_i\}$ form a valid preprocessing, then say (h, h') *defines correct preprocessing.*

2. If h was output by a previous query $H_1(\mathsf{com}_1, \ldots, \mathsf{com}_n)$, and each com_i was output by a previous query $H_0(\mathsf{state}_i)$, and h' was output by a previous query $H_2(\{\hat{z}_\alpha\}, \mathsf{msgs}_1, \ldots, \mathsf{msgs}_n)$ where $\{\mathsf{state}_i\}, \{\hat{z}_\alpha\}, \{\mathsf{msgs}_i\}$ are consistent with an online execution of Π among all parties with output 1 (but the $\{\mathsf{state}_i\}$ may not form a valid preprocessing), then say (h, h') *defines correct execution.*

3. In any other case, say (h, h') is *bad.*

(Note that in all cases the relevant prior query, if it exists, must be unique since the experiment is aborted if there is ever a collision in H_0, H_1, or H_2.)

In Experiment 7, for each query $G(m, h_1, h'_1, \ldots, h_M, h'_M)$ made by the adversary (where m was not previously queried to the signing oracle), check if there exists an index j for which (h_j, h'_j) defines correct preprocessing and correct execution. We let Invert be the event that this occurs for some query to G. Note that if that event occurs, the $\{\mathsf{state}_i\}, \{\hat{z}_\alpha\}$ (which can be determined from the oracle queries of the adversary) allow computation of w' for which $C(w') = 1$. Thus, $\Pr_7[\mathsf{Invert}] \le \epsilon_{OW}$.

We claim that

$$\Pr_7[\mathsf{Forge} \wedge \overline{\mathsf{Invert}}] \le q_G \cdot \epsilon(M, n, \tau).$$

To see this, assume Invert does not occur. For any query $G(m, h_1, h'_1, \ldots, h_M, h'_M)$ made during the experiment (where m was not previously queried to the signing oracle), let Pre denote the set of indices for which (h_j, h'_j) defines correct preprocessing (but not correct execution), and let $k = |\mathsf{Pre}|$. Let $(\mathcal{C}, \mathcal{P})$ be the (random) answer from this query to G. The attacker can only possibly generate a forgery (using this G-query) if (1) $[M] \setminus \mathcal{C} \subseteq \mathsf{Pre}$, and (2) for all $j \in \mathsf{Pre} \cap \mathcal{C}$, the value p_j is chosen to be the unique party such that the views of the remaining parties $\{S_i\}_{i \neq p_j}$ are consistent . Since $|M \setminus \mathcal{C}| = M - \tau$, the number of ways the first event can occur is $\binom{k}{M-\tau}$; given this, there are $k - (M - \tau)$ elements remaining $\mathsf{Pre} \cap \mathcal{C}$. Thus, the overall probability with which the attacker can generate a forgery using this G-query is

$$\begin{aligned}\epsilon(M,n,\tau,k) &= \frac{\binom{k}{M-\tau}\cdot n^{M-k}}{\binom{M}{M-\tau}\cdot n^{\tau}} \\ &= \frac{\binom{k}{M-\tau}}{\binom{M}{M-\tau}\cdot n^{k-M+\tau}} \\ &\leq \epsilon(M,n,\tau) = \max_k \{\epsilon(M,n,\tau,k)\}\,.\end{aligned}$$

The final bound is obtained by taking a union bound over all queries to G.

□

6.3 Tree-Based Optimizations

In this section we discuss standard constructions of seed tree and Merkle tree and their use in Picnic2. These are well-known cryptographic objects, which are used in a standard way. Nevertheless, in this section we briefly sketch security arguments corresponding to their use. We first consider some properties of the seed tree construction, then discuss the use of seed and Merkle trees in Picnic2.

6.3.1 Seed Tree

The beginning of this section is taken (with minor modifications) from the Picnic specification [Tea19a], Section 7.3.1, that describes how the tree is constructed, and how seeds are efficiently revealed.

When signing, the signer must generate a set of seeds, then reveal a subset of these based on the challenge. The seeds are then used by the verifier to check that the MPC protocol was setup or simulated correctly. By deriving seeds deterministically in a binary tree, then using the leaf seeds in the protocol, the signer can reveal large subsets of the seeds efficiently by revealing intermediate nodes in the tree. In Picnic2, the signer must reveal $M-\tau$ of the M initial seeds and one of the n seeds in each of the τ MPC instances that are checked by the verifier.

The tree is initialized with random data at the root node. For each non-leaf node in the tree, having seed `parent_seed`, compute the 2κ-bit digest

$$d := H(\texttt{parent\_seed}\|salt\|j\|i)$$

where j is the MPC instance number, and i is the index of the parent node. The function H is a hash function with 2κ-bit outputs. Then set the left child of the node to the leftmost κ bits of d, and the right child to the rightmost κ bits of d. The *salt* is a random, per-signature value that is included to prevent multi-target attacks (along with the counter). The values *salt*, j and i are always public, but only some of the `parent_seed` values are revealed.

Collision-resistance Non-malleability of Picnic2 signatures requires that the seed tree be collision-resistant. This means it must be hard to find distinct seeds that expand to the same set of leaf seeds. In practice 2nd preimage resistance should be sufficient, since an attacker must find a second seed that expand to the same set of seeds appearing in a valid signature. However, we use collision resistance because it allows us to prove that the proof protocol has computationally unique responses (also called quasi-unique responses).

Theorem 6.3. *The seed tree construction is collision-resistant if the hash function H is collision-resistant.*

Proof. The algorithm to reveal seeds ensures that any node without a sibling is revealed directly (as opposed to being re-derived from another seed). Therefore, both halves of the output bits of H are used, so if distinct seeds s and s' derive the same set of seeds, we have a collision, $H(s) = H(s')$ where s is a parent of two leaf seeds. □

Hiding and Pseudorandomness The derived seeds must be pseudorandom for security of the protocol. Further, the unrevealed seeds must remain hidden, or else the signature will leak information about the private key. Therefore, given the revealed seeds, it must be difficult to distinguish the unrevealed seeds from random values. For this we will rely on the function H being a PRF keyed by the parent seed, so that given half the output bits of $H(\texttt{parent\_seed}\|salt\|j\|i)$ the sibling value the other half of the output bits are indistinguishable from a random value.

Theorem 6.4. *Given the opening information for a set of revealed seeds, the unrevealed seeds remain indistinguishable from random values, assuming H is a secure PRF when keyed with an unrevealed seed.*

Proof. Consider a subtree with root p, having left child $r = \mathrm{leftHalf}(H(p\|\ldots))$, and right child $s = \mathrm{rightHalf}(H(p\|\ldots))$. We must show that if r is revealed, s remains indistinguishable from random (the same argument holds when r

and s are reversed). Since H is assumed to be a PRF with key p, and p is unrevealed, the output $H(p\| \dots)$ is indistinguishable from a random value. In a random value the two halves of the output are independent, so having one half gives no information about other, so the hidden half remains indistinguishable from random.

The value p is unrevealed, if it has a sibling that is revealed, the same argument is applied recursively up the tree (with height bounded by $O(\log(M))$, polynomial in κ). The root is only revealed when there are no unrevealed seeds. □

In practice H is SHA-3, and inputs always have a fixed length.

6.3.2 Use in Picnic2

Here we review where the the seed tree and Merkle tree optimizations are used in Picnic2 and argue that the security properties above are sufficient to maintain security.

The seed tree construction is used in two places. First, in Step 1.1, instead of choosing $\{\mathsf{seed}^*_j\}_{j=1}^{M}$ at random, the signer derives them using a seed tree. Then in Step 3, the signer outputs seeds from the tree that allows the verifier to recompute the $M - \tau$ revealed seeds.

- The privacy property of the seed tree construction (Theorem 6.4), ensures that the unrevealed seeds are indistinguishable from random.
- For malleability, if the seeds output by the signer are modified, collision resistance ensures that at least one of the $M - \tau$ reconstructed seeds is different. Then verification will fail, since seed^*_j is used as the root of a seed tree, which is collision-resistant, meaning the values $\mathsf{seed}_{j,i}$ will be different.

The second use is in Step 1.1, in each instance j, the n seeds $\{\mathsf{seed}_{j,i}\}$ are derived with the seed tree construction with root seed^*_j. Then in Step 3, instead of revealing $\{\mathsf{seed}_{j,i}\}_{i \neq p_j}$, the signer reveals seeds from the tree that allow the verifier to recompute this set.

- The privacy property of the seed tree construction ensures that seed_{j,p_j} is indistinguishable from random.

- For malleability, if the revealed seeds are modified such that the recomputed seeds $\{\mathsf{seed}_{j,i}\}_{i \neq p_j}$ are different, then verification will fail when $\{\mathsf{com}_{j,i}\}_{i \neq p_j}$ are recomputed (or there is a collision in H_0, H_1 or G). Collision resistance of the seed tree construction ensures that changing the seeds output by the signer will cause the seeds recomputed by the verifier to be different.

The Merkle tree is used as a commitment to a set of commitments, the values $\{h'_j\}_{j=1}^{M}$. Using a Merkle tree does not affect the hiding property of the commitments h'_j (which are public when a Merkle tree is not used), so , we only need to consider whether the Merkle tree is a binding commitment. It can be shown that providing two openings for a left with respect to the same root gives a collision in H (the function used to form the Merkle tree).

6.4 QROM Security

Security of Picnic2 is formally proven in the random oracle model (ROM), similar to the Picnic-FS parameters. However, we chose not to specify parameter sets using the KKW proof protocol and the Unurh transform. First, because the Unruh transform relies on special soundness, it does not immediately apply to KKW. Second, we believe that Picnic2 (the Fiat-Shamir transform of KKW) is a secure signature scheme in the quantum random oracle model (QROM).

In the past few years, the research community has made good progress towards understanding the security of FS signature schemes in the QROM. Recent preprints by Don et al. [DFMS19] and Liu and Zhandry [LZ19] have shown that a larger class of FS signature schemes are secure in the QROM. If a Σ protocol has a property called *quantum computationally unique responses* (also called *collapsing*), then the signature scheme obtained by applying the Fiat-Shamir transform is secure in the QROM. Additionally, Σ must also have 2-special soundness. This last requirement makes the results inapplicable to Picnic, since ZKB++ has 3-special soundness, and Picnic2 is not known to be special sound. It appears – though we have not formally shown – that the results of Don et al. [DFMS19] can be generalized to 3-special sound protocols (with a further loss in tightness, as in the ROM) and thus be applicable to the Picnic-FS parameters. However, we do not yet know whether they can be applied to Picnic2. [3]

[3] We're grateful for discussions with Jelle Don, Serge Fehr, Christian Majenz, Christian

We are working on a QROM analysis of the Picnic2 parameter sets as specified. The proof uses ideas from the literature and from our ROM proof, but is specific for Picnic2. The main difference that Picnic2 has when compared to general sigma protocols is that it is possible to extract a witness from part of the hash query history of a successful forger without rewinding (as shown in our ROM proof).

7 Analysis with Respect to Known Attacks

In this section we analyze the Picnic signature scheme with respect to known attacks. First, we observe that in case we deal with ideal primitives, Corollary 5.4 already gives us a provable bound for EUF-CMA security. Since those primitives are, however, instantiated with concrete building blocks, we consider concrete attacks on those building blocks. In our scheme, we use the classical approach to turn Σ-protocols into signature schemes in the random oracle model. Based on the fact that, since the introduction of the random oracle model [BR93], no attack which arises from the assumption that a hash function behaves as a random oracle (except for some artificial counterexamples such as [CGH98]) was found [KM15], we claim that the best attacks against our scheme are attacks which also invalidate the claims made for the underlying symmetric primitives.

All cryptographic primitives, except the one-way function LowMC, rely on the SHA3 function SHAKE [NIS15], a well established and standardized primitive, and we use it in a standard way. For those primitives, we have already gained substantial confidence regarding security due to extensive cryptanalysis efforts within the community. We therefore do not see these building blocks as a central attack surface and assume that the bounds given in [NIS15, Table 4] hold. We note that improvements in attacks against those primitives also lead to improvements in the attacks against our signature scheme.

7.1 Usage and Security Margin of LowMC

Consequently, we henceforth put focus on attacks on the one-way function f. Essentially, the function f could be any one-way function, but since we found block ciphers—and, in particular the LowMC family of block

Schaffner (the authors of [DFMS19]) to help us understand their work.

ciphers [ARS+15, ARS+16]—gave the most efficient signatures, we decided to use them in our signature scheme. In particular, we assume that using LowMC as described below yields a suitable family of one-way functions $\{f_u\}_{u \in \mathsf{K}_\kappa}$. We use this function to establish a suitable relation between secret and public keys. In particular, let

$$f_u(x) := E(x, u),$$

and let E denote LowMC encryption with respect to a single block u under key x. The keys in our signature scheme are generated as follows. First, one chooses a LowMC encryption key x, as well as a single block u uniformly at random. Then, the the public verification key pk, as well as the secret signing key sk are defined as follows

$$\mathsf{pk} := (y, u) = (f_u(x), u), \ \mathsf{sk} := (\mathsf{pk}, x).$$

The choice of the number of rounds within LowMC comes with a significant security margin. For security level L1 with the specified 20 rounds, the best attack known is on 12 rounds. For security level L3 with the specified 30 rounds, the best attack known is on 19 rounds. For security level L5 with the specified 38 rounds, the best attack known is on 26 rounds. And even those attacks require an attacker to see two plaintext-ciphertext pairs for the same key, whereas within our signature scheme an attacker only ever sees a single input-output pair for every key.

7.2 Attacks in the Single-User Setting

In the single-user setting, the attacker only ever sees a single key pair for the Picnic signature scheme, i.e., a single plaintext-ciphertext pair $(f_u(x), u)$ of LowMC with respect to a uniformly random key x and a uniformly random block u. Consequently, in this setting cryptanalytic results for LowMC also directly apply to our scheme, and also the claimed bounds carry over to the Picnic signature scheme. Note that, one could even globally fix x to further shrink the size of the public verification key pk. However, we chose not to do so, as we also want to consider attacks in the multi-user setting (as discussed below).

7.3 Attacks in the Multi-User Setting

The multi-user setting more accurately models reality, in that there are multiple users, each with a public key, and the adversary is considered successful if he can attack any one of the users.

Multi-User $\mathsf{EUF\text{-}CMA}$. Even if single-user $\mathsf{EUF\text{-}CMA}$ security generically implies multi-user $\mathsf{EUF\text{-}CMA}$ ($\mathsf{MU\text{-}EUF\text{-}CMA}$) security under a polynomial loss [GMS02], we put concrete focus on attack scenarios which become applicable by moving to the multi-user setting. Here, the adversary may see many signing key pairs and we need to be cautious with respect to more sophisticated attacks that might apply.

In particular—in contrast to the single-user setting—our decision to choose an independent and uniformly random block u, being the encryption function $E(\cdot, u)$ of LowMC, per signing key pair turns out to be important. This is because using the same, fixed block u with independent keys $x_1, \ldots, x_n$ for each of the n users would allow an adversary to apply multi-user key recovery attacks [Bih02] and generic time-memory trade-off attacks like [Hel80] and in particular time/memory/key trade-off attacks [BMS05]. In these attacks one of n block cipher keys may be recovered in less time than the cost of recovering a single key, and the attacks become more efficient for large n. Intuitively, the random block chooses a unique function per user, and work done to attack one user (function) can not be used to simultaneously attack another user (function). In addition, Banegas and Bernstein [BB17] have recently shown that parallel collision search attacks [vOW94] can also be applied in the quantum setting which also supports making a random choice of u per user. Finally, we note that one could choose a smaller value that is unique per user (with a potential decrease in security) to reduce the size of the public key. However, since public keys in our schemes are already small (at most 64 bytes), our design uses a full random block to be as conservative as possible.

Key-Substitution Attacks. These are attacks where an adversary who is given a signature σ_A on message M under A's public key pk_A manages to come up with a public key pk_E (different from pk_A) such that σ_A verifies under pk_E and message M. Menezes and Smart in [MS04] provide a formal model to cover such attacks, which are not covered by $\mathsf{EUF\text{-}CMA}$ security. We explicitly consider such attacks. Security against these types of attacks can be generically achieved. This has been shown in [MS04], and we recall their theorem below.

Theorem 7.1 ([MS04], Theorem 6)**.** *Let* $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ *be an* EUF-CMA *secure signature scheme. Then,* $(\mathsf{Gen}, \mathsf{Sign}', \mathsf{Verify})$ *with* $\mathsf{Sign}' := \mathsf{Sign}(\mathsf{sk}, \mathsf{pk} \| m)$ *and* pk *being an unambiguous encoding of the public key is a secure signature scheme in the multi-user setting.*

We stress that the above result in particular holds for sEUF-CMA secure signature schemes.

As discussed in Section 3 (see also Section 4.3 of the specification), the public key is prepended to the message on signing, and the specification provides an unambiguous encoding (since the public key is a pair of bitstrings, the encoding is trivial). Consequently, we have the following corollary.

Corollary 7.2. *Picnic-FS and Picnic-UR provide security in the multi-user setting in the sense of [MS04, Definition 6].*

7.4 Multi-Target Attacks

In [DKP+19], Dinur and Nadler describe attacks against the version 1.0 specification of Picnic (the `Picnic-FS`, and `Picnic-UR` parameter sets). At the time, the Picnic2 parameter sets were not specified, but the attack applies equally to a direct instantiation of the KKW protocol. Their attacks are multi-target attacks, where an attacker has a list of values of the form $y_1 := H(x_1), \ldots, y_S := H(x_S)$, and recovering any of the x_i leads to a successful attack. Specifically, the x_i values are k bits long, and the attacker can recover the k-bit singing key. The y_i values can come from (i) a single signature (there are about 2^7 in a signature), (ii) from many signatures created by the same signer, or (iii) from signatures from multiple signers.

In Dinur and Nadler's attack, the x_i values are the seeds used for each party, in each MPC instance. The function H expands x_i to a random tape, used during the MPC protocol simulation. In Picnic the seeds of two of three parties are revealed to the verifier, and in Picnic2 $n - 1$ of n are revealed. If an attacker learns the missing seed, they can recover the secret shared input, the signing key. Specifically H is the SHAKE XOF, and the output length depends on the parameter set, but is always above 600 bits.

What makes the attack non-obvious, is that the output of H (the random tapes) are not revealed directly. Dinur and Nadler show that given the states of the opened parties, it's possible to solve for some of the unopened party's random tape. They show this is a property of MPC protocols, that doesn't affect the usual MPC security notion (Briefly, the MPC protocol must guarantee privacy of inputs, not the randomness, and leaking some randomness is allowable provided it does not affect privacy). They also quantify the number of random bits that can be recovered from an MPC instance, and the cost. In all cases, it is possible to efficiently recover more than k bits, so that testing a candidate x_i value can be done. However, the bits from each target have different positions in the random tape, complicating an efficient implementation of the attack. In a typical multi-target attack, the attacker iterates over candidate values x', computes $y' = H(x')$, then compares y' to the y_i efficiently using a data structure (e.g., a hash table or search tree). But here comparing the candidate tape y' (with all bits known) to the target tapes y_i (where a different subset of bits are known for each), is not obviously efficient. Dinur and Nadler show that it can be made efficient, and precisely quantify costs under various settings. In the best case, when all signatures are created by a single signer, their attacks cost $2^{k-7}/S$ (information theoretically optimal). In other cases, the attack is cost is higher, but still below the expected security level.

Mitigation The 2.0 version of the specification makes a change to mitigate these attacks (in all parameter sets). The change adds additional information to the input of H, called a *salt*, so that a candidate seed x', cannot be tested by comparing y' to all y_i, since each y_i is computed using a different salt. The salt ensures that y' would need to be recomputed with the correct salt before each comparison. To address all three variants of the attack, the salt must be unique per signature, per signer and per invocation of H. The first change is to choose a random per signature salt, 256 bits long. This ensures that (with high probability), the salt is unique across all signatures recorded by an adversary.

Then, to ensure that salts are unique within a signature, we also include a pair of counters, the first value corresponds to the MPC instance number, and the second corresponds to the invocation number of H. The specification already uses a domain separation technique, where different hash functions are created for different purposes, as follows, $H_i(x) = H(i\|x)$. This mechanism also helps ensure salts are unique, e.g., when computing the seed tree we can use H_i and when computing the Merkle tree use H_j, and not worry about (salt, counters) pairs being repeated in both trees.

The main cost of the mitigation is a increase in signature size, of 256 bits, which is small relative to the overall size of the signature. Some additional data must also be hashed, but CPU costs in our benchmarks did not increase appreciably. This is likely due to the fact that hash inputs were short to begin with, and remain short (smaller than the hash function block size), even with the salt.

8 Expected Security Strength

Since a Picnic keypair is a block cipher key and plaintext/ciphertext pair, we chose to define parameters at the L1/AES-128, L3/AES-192 and L5/AES-256 security levels. By the CFP, this means that L2 is implicitly defined by L3 and L4 is defined by L5.

We expect each parameter set to provide security equivalent to AES. For example, at L1, we expect 128-bits of security against classical attacks, and at least 64-bits against quantum attacks. Like AES, key recovery attacks using Grover's algorithm against LowMC are the best known quantum attacks on Picnic. The estimate of 64-bits comes from an idealized version of Grover's algorithm capable of running for a long time. Like AES this may be conservative, even more so in the case of LowMC, since the circuit is much larger than AES, due to the large amount of constant data required to implement it.[4] For example, at level L1, the Picnic LowMC instance requires 34KBytes of constant data. These constants must either be encoded into the circuit, or the circuit must be expanded to recompute them on the fly. Thus the LowMC circuit is orders of magnitude larger than AES, making attacking LowMC with Grover's algorithm at least as difficult as attacking AES.

We similarly set the number of parallel iterations to provide 128-bit security against classical attacks, and 64-bits against quantum attacks using an idealized version of Grover's algorithm. Like with LowMC, the circuit required to break ZKB++ soundness with Grover's algorithm is orders of magnitude larger than the AES circuit.

In more detail, suppose an attacker is trying to forge a proof as a generic search problem. In particular, if an attacker can find a permutation of a set of transcripts that hash to a challenge chosen in advance, he can forge a proof. Consider a T round protocol (Picnic specifies 219, 329 and 438 rounds

[4] Previously, when we compared the LowMC circuit size to AES, we were looking only at AND gates, but here we're considering all gates.

at levels L1, L3 and L5, resp.). Then there are 3^T possible challenges that can come from hashing those $3T$ transcripts (since there are 3 challenges).

Now consider an attacker who constructs invalid ZKB++ "proofs" such that for each ZKB++ iteration, he can give a valid response to two of the challenges but not the third. If we model the hash function as a random oracle, the probability of getting a challenge for which he can respond is $(\frac{2}{3})^T$, and thus we expect that if the attacker searches a space of $(\frac{3}{2})^T$ candidates (i.e., permutations of transcripts that are constructed in this manner) he can find one. Grover's algorithm allows the attacker to search the space in time $(\frac{3}{2})^{T/2}$.

However, the items in the space are larger, and changes in one value (e.g., a seed value) requires re-computing many others (the random tape, the MPC transcript, and the commitments). Clearly this is far more expensive than a single AES evaluation, and so we assume that Grover's algorithm applied to breaking ZKB++ soundness is at least as costly as AES key recovery.

Using Larger LowMC Keys We could reduce the feasibility of Grover key recovery attacks against LowMC by increasing the keysize and keeping the block length fixed. There would still be a chance to use Grover's algorithm to break the soundness of ZKB++ (unless the number of parallel iterations was increased). However, a quick inspection reveals that the computation of attacking soundness is computationally much more complex than attacking LowMC. For example, checking whether a candidate secret key corresponds to a given public key requires one LowMC evaluation, while checking whether a set of cheating commitments leads to a challenge that does not catch the cheating requires hundreds of SHA3 computations.

8.1 LowMC Parameter Selection

The choice of LowMC parameters may seem aggressive in the context of a general-purpose block cipher. The LowMC spec recommends an additional 1.3 times the number of rounds, as a security margin against unknown attacks. Picnic does not use these additional rounds.

The general block cipher security definition gives attackers as much power as possible, to model the worst case scenario. Consider the CPA security game, where attackers may choose plaintexts, query the encryption oracle many times, and must only distinguish encryptions of chosen plaintexts in order to break the cipher (as opposed to recovering plaintext or private keys).

This strong security definition is sensible when the primitive will be used in a variety of (potentially unforeseen) applications.

By contrast, for the security of Picnic signatures, attackers are severely restricted. They are given a single plaintext/ciphertext pair, for a randomly chosen block and key, and succeed if they can recover the key. Signatures are zero-knowledge proofs, so by definition provide no additional useful information. Thus the success criteria for the LowMC attacker in our context is key recovery, which is more difficult than indistinguishability, while the capabilities are more limited. In this context, the parameters we have chosen for LowMC are arguably very conservative. We believe that reducing the number of rounds further would maintain security, but chose to use the full number of rounds as a security margin, given LowMC is a relatively new design.

Further, it is difficult to quantitatively support parameter selection in our restricted attack model, since most research focuses on the standard security definition. Our claim is that the complexity of a key recovery attack against LowMC, given only a Picnic public key, is at least as difficult as attacking the CPA security of AES (for equivalent key size, and with Picnic the block size always matches the key size).

As we improve our understanding of LowMC security in this restricted attack model, we expect to be able to justify reducing the number of rounds further. The motivation is the direct impact this has on the size of Picnic signatures. For example, at L1 with 128-bit blocksize and keysize, with 10 s-boxes, the recommended number of rounds is 20 and Picnic-FS signatures are about 33KB. Reducing the number of rounds to 10 would make signatures 24KB.

8.2 Hash Function Security

Picnic depends on secure hash functions when computing signatures, for commitments and the challenge. In our security analysis we have modeled these as random oracles. While choosing parameters we also took into account some some more specific security properties (all implied by a random oracle).

All hash functions are implemented with SHAKE128 with 256-bit digests at security level L1, and SHAKE256 at levels L3 and L5, with 384 and

512-bit digests, respectively.[5] We expect the concrete security provided by SHAKE for collision and preimage resistance claimed in [NIS15], extended to quantum attacks.

For preimage resistance, in the classical case it is common to assume $O(2^n)$ operations for standard hash functions. When considering quantum algorithms, Grover's algorithm can find preimages with $O(2^{n/2})$ operations. Therefore, we assume that our uses of SHAKE128 and SHAKE256 provide this level of preimage resistance.

When considering quantum algorithms, in theory it may be possible to find collisions using a generic algorithm of Brassard et al. [BHT98] with cost $O(2^{n/3})$ (for n-bit digests). A detailed analysis of the costs of the algorithm in [BHT98] by Bernstein [Ber09] found that in practice the quantum algorithm is unlikely to outperform the $O(2^{n/2})$ classical algorithm. Multiple cryptosystems have since made the assumption that standard hash functions with n-bit digests provide $n/2$ bits of collision resistance against quantum attacks (for examples, see papers citing [Ber09]). We make this assumption as well, and in particular, that SHAKE128 with 256-bit digests provides 128 bits of PQ security, SHAKE256 with 384-bits provides 192-bits and SHAKE256 with 512-bit digests provides 256-bits.

9 Advantages and Limitations

9.1 Compatibility with Existing Protocols

Here we describe some work we did to demonstrate compatibility of Picnic signatures existing protocols protocols, TLS, and X.509. We also prototyped protecting Picnic private key operations on a commercial hardware security module.

9.2 TLS and X.509 Compatibility

The optimized implementation of Picnic has been integrated with the Open Quantum Safe (OQS) project.[6] Then, using a modified version of OpenSSL[7]

[5] We considered using SHAKE256 at all three levels for simplicity, but L1 signing and verify times increased by roughly 10%.

[6] https://github.com/open-quantum-safe/liboqs

[7] https://github.com/christianpaquin/openssl

modified to use OQS, we were able to create X.509 certificates signed with Picnic and certificates certifying Picnic public keys. These keys and certificates were then used to establish TLS 1.2 connections, where the key exchange algorithm was one of the the LWE-Frodo or SIDH algorithms from OQS. To our knowledge, these were the world's first TLS connections to use both post-quantum secure key exchange and authentication algorithms.

OpenSSL had to be patched in one place to handle larger signature sizes, since the TLS 1.2 standard has a limit of $2^{16} - 1$ bytes. Signatures for the L1 parameter sets are under this limit, but for L3 and L5 we would likely need an extension to support larger signatures. Ideally a future version of TLS would allow larger signatures, but this is not pressing as we expect the L1 parameter set to provide sufficient security in the short and medium term. Otherwise the integration was smooth, and performance seemed acceptable in our limited experiments. In particular the certificate stack (X.509/ASN.1) worked unmodified with signatures this large, something we did not expect.

We then compiled the Apache web server against our version of OpenSSL that uses OQS. No source code modifications to Apache were necessary, we simply had to configure the build to use the OQS version of OpenSSL. We configured Apache to host static HTML files, that we fetched over HTTPS with various ciphersuites and measured the latency observed by the client. The results are given in Section 11.7.

9.3 Hardware Security Module Compatibility

Cryptographic keys are often protected by specialized hardware known as *hardware security modules* (HSMs). An HSM is a tamper-resistant device that stores keys and performs operations in response to calls to a limited API. The primary security goal is that private keys never leave the device (with exceptions for backup and export to other similar devices). Secondary goals are tamper proof logging and isolation of cryptographic operations from the rest of the system, forming a strong security boundary. If an attacker compromises the system, they can use the key, but cannot export it for use on another system, and ideally, cannot use it without leaving logs.

For signature keys, example operations are generating keys and signing (digests). Upon key generation, a key identifier is output, which can be used in sign calls to refer to the private key (that may be marked non-exportable). An example API is PKCS #11, standardized so that applications can be agnostic of the underlying hardware.

Many such devices are available on the market, with a range of features, performance and security hardening. They may be small peripheral devices similar to a smartcard, or standalone, network connected servers. Often the firmware on these devices is fixed by the manufacturer, and prototyping new algorithms is not possible. One device that does allow the owner to provide some custom firmware implementing new cryptographic algorithms is the Utimaco SecurityServer Se50 LAN V4. We added support for Picnic on this device and describe our experience here.

On the Utimaco HSM, the owner may (i) use the provided cryptographic modules (e.g., RSA, ECDSA, SHA-256, and RNG) in the firmware, (ii) write their own cryptographic module that doesn't depend on any of the provided modules, or (iii) write a module implementing a new primitive that uses some of the provided modules. To implement Picnic on the HSM, we experimented with option (iii). In particular we leveraged the RNG, ASN.1 and SHA3 modules from Utimaco, and implemented the remainder of our spec in a custom module (named PICNIC). The PICNIC module was a port of our reference implementation that replaced the RNG and SHA3 with calls to the Utimaco modules. Additionally we added a module named CERT, that uses the PICNIC and ASN1 modules to create X.509 certificates signed with Picnic.

Whether it is desirable to create certificates on the HSM, vs. creating them on the host application and using the HSM only to sign them is debatable. On one hand having more functionality in the HSM may allow CA policy to be better enforced in the event of a compromise of the host application. On the other hand, having more functionality in the HSM increases its attack surface.

Since having the HSM be a limited singing oracle is a special case of having it create certificates, in our demo we created certificates on the HSM. If this more complicated design can be demonstrated to work, then the simpler case should also work.

The certificates are standard X.509 v3 certificates, but with custom object identifiers (OIDs) for Picnic keys and signatures. We confirmed that the resulting certificates parsed correctly in existing viewers (e.g., http://www.lapo.it/asn1js/).

Typically the sign API of an HSM accepts a hash of the data to be signed. In contrast, the sign API of the Picnic spec accepts the (unhashed) data to be signed, and hashes it internally. The Picnic spec allows digests to be signed, but this is intended only for very large messages, as it requires stronger security properties from the hash function (as with other Schnorr-like signatures, with EdDSA being another example, see RFC 8032). In our demo, we did not have issues sending larger amounts of data to the HSM (we tested up to roughly 10k bytes), so we expect the sign API without pre-hashing the message to work for most PKI scenarios.

Scenario: Post-Quantum Public-Key Infrastructure. We implemented a small demo to test Picnic in a public-key infrastructure (PKI) scenario, from the perspective of a certificate authority (CA).

The demo architecture has two main software components.

1. A host application, running on a Windows PC.
2. A pair of firmware modules, running on the HSM, housed in a machine room in a building across the street.

The application is connected to the HSM with a 100MBit connection, The latency of a no-op call between host and HSM was about 24 ms. We also tried running the host application from a laptop at home where latency was about 200ms, and saw larger variance in the roundtrip times for operations but similar mean times.

We implemented the following CA operations.

1. The HSM generates and stores new Picnic key pair, and creates a self-signed certificate. This is the root certificate of the PKI.
2. The host application generates and stores a new Picnic end-entity (EE) key, and then the CA issues a certificate for the EE key using the signing key from the previous step. This is not a typical CA operation, but was a useful stepping stone during development.
3. The host application sends a certificate signing request (CSR) to the HSM. The CSR is created with OpenSSL, and the subject public key is an RSA key pair. The HSM issues a cert for the RSA public key, signed with Picnic.

For item 3, it would have been more realistic to use Picnic as the subject public key and sign the CSR with Picnic as well, but our version of OpenSSL (the OQS OpenSSL fork described above), did not support creating CSRs with post-quantum algorithms yet.

Our new modules were tested in the HSM simulator first, then cross-compiled for the HSM itself and uploaded.

Software. The software we used for this experiment is available at `https://microsoft.github.io/Picnic/` under the MIT license. Note that although our code is MIT licensed, building it and running it (even in the simulator), may require a license for software and tools from Utimaco.

The main effort involved was porting the reference implementation to build with the HSM's tools. It took between two and three person-weeks, and this included time to get familiar with the HSM tools and development process. Porting the Picnic library to the HSM required the following steps:

- Create an empty HSM module from the HSM's SDK.
- Add the Picnic source and header files to the module code and update the module's makefile for the c6000 cross-compiler.
- Replace the standard C libraries with the HSM provided libraries and update calls where the names differ. Most of these changes deal with memory management and string handling in error handling code.
- Update the RNG and SHA-3 calls in Picnic to use the HSM modules for these functions.
- The c6000 compiler is C89 compliant and lacks features of more modern C standards. Several small changes were required dealing mainly with variable declarations.
- Create a new public interface for the Picnic module that more closely resembles the other HSM's crypto modules for consistency.
- When the code is building and functioning properly in the HSM simulator, cross-compiled the code for the HSM; sign it; load it into the HSM and verify it is working correctly.

Performance and Discussion. The goal of this prototype was to demonstrate that using post-quantum signatures in a PKI scenario is practical, and that there are no major impediments to deployment even with existing commercially available HSM hardware. In particular, using new types of keys, and creating signatures with a new algorithm, having larger signatures than traditional algorithms, and hashing the message on the HSM was possible, and did not pose significant engineering challenges.

Some benchmarks are given in Table 5. We signed messages of three sizes, 100B, 1KB and 10KB, covering the range of message sizes we would expect in our PKI scenario. We measured the round trip time of a call to the HSM from the host application. There was no significant difference for the different message sizes. For the L1-FS parameter set, the round trip time is about half a second, and goes up to about four seconds for the L5-FS parameter set.

For many PKI applications the number of certificates created is relatively small, and this performance would be considered adequate. We stress that this is an unoptimized implementation, and we don't have a breakdown of where the time was spent, i.e., network time vs. computation time on the HSM. For improving computation time, using our optimized implementation would be a first step.

	Sign 100B	Sign 1KB	Sign 10KB	Keygen
L1-FS	0.4474	0.4477	0.4504	0.0050
L3-FS	1.6478	1.6474	1.6509	0.0069
L5-FS	4.0854	4.0841	4.0860	0.0096

Table 5: Mean round trip times in seconds for calls to an HSM creating Picnic signatures (average of 10 calls).

10 Additional Security Properties

10.1 Side-Channel Attacks

Key Generation. Key generation requires generating a random LowMC key and plaintext, and computing the LowMC block cipher. A fast implementation of the LowMC block cipher may use precomputed data, and have cache-timing side channels, because the access pattern depends on the secret key. However, there are a couple mitigations:

1. Since key generation happens infrequently, a slower LowMC implementation with a constant access pattern can be be used.

2. Even if a side-channel is present in key generation, since only one encryption with a given secret key is ever computed, and known attacks require observing multiple runs, the feasibility of a successful attack is unlikely.

Signing. Generally speaking, since the three party protocol simulated during signing is circuit-based, the same operations are performed, regardless of the values on the input wires of the circuit.

Signing is not constant time in the absolute sense, but is constant time with respect to the operations that depend on the ephemeral random values (that in turn depend on the signing key). The timing (and signature size) variation is due to the different operations performed depending on the (public) bits of the challenge. The reference and optimized implementations are constant time relative to the secret key.

The Picnic design has what seems like a natural mitigation to some side-channel attacks, since the LowMC secret keys used by each of the players is a randomized secret sharing of the actual key. The shares are only ever used once, and the randomization means that access pattern information learned by an attacker in one parallel iteration of ZKB++ or KKW can not be combined with information from other iterations. Since many side-channel attacks require multiple observations (called traces), we expect this mitigation to be effective.

Note that the derandomized variant of the sign algorithm (i.e., where the per-signature randomness is derived from the message to be signed and the secret key), may in fact use the same shares multiple times, when signing the same message. Therefore if a side-channel attacker can repeatedly cause a signature to be computed on the same message, while observing the signing device, they may be able to collect multiple traces. The Picnic spec allows randomized signatures, and recommends that when deriving the ephemeral value additional entropy by included. Our implementations do not do this at the moment, to allow for easier testing. Randomization may also help mitigate fault attacks and differential attacks against Picnic, though we have not yet investigated this topic (see [PSS+17, ABF+17]).

The circuit decomposition technique is similar to the side channel countermeasure called masking, commonly used to protect block cipher implementations from side channel attacks. An early, well cited paper on the topic is Goubin and Patarin [GP99]. With further study, we may find that the ZKB++ circuit decomposition provides other types of side channel resistance, for example, resistance to differential power analysis.

10.2 Security Impact of Using Weak Ephemeral Values

The specification recommends that the per-signature random values used when computing a signature be derived from the signing key and the message, to simplify testing and to mitigate the security impact of a defective random number generator during signing. The goal is that signatures are secure, provided the random number generator was secure during key generation.

Like (EC)DSA, given the random values used when computing a signature, it is possible to recover the signer's secret key. Recall that in each parallel iteration of ZKB++, three seed values are generated, one for each party in the MPC protocol. In normal operation, two of these seed values are revealed, and one is kept secret. The MPC protocol remains secure when two of the three parties are corrupted (i.e., have their seed exposed, which exposes their state, input and output shares). Given the entire third seed, it is possible to recover the input share of the third player, and recover the secret key.

Unlike (EC)DSA, slight biases in the random number generator do not allow the secret to be recovered from multiple signatures. This is because the seed values are never used directly; they are always expanded with an extendable output functions (XOF) into a random tape, and the random tape values are used.

Regardless of how the seeds are generated, a bias in the XOF output may lead to an attack. For example, if the XOF/PRF used to derandomize (EC)DSA, EdDSA, and other ElGamal-like signatures was biased, the ephemeral value is biased, and the lattice attacks studied in the context of RNG biases apply [Ble00, HS01]. For Picnic, it's not clear if this could be exploited.

10.3 Parameter Integrity

With some cryptographic primitives if the system parameters are changed, security is lost. For example, if an elliptic curve secret key is used on a weak curve, the primitive may still work, but leak the secret key.

In Picnic, the parameters are small integer values like the parallel repetition count that tend to be hard-coded in software, and the LowMC matrices and constants. Using weak parameters for LowMC could weaken the cipher to the point where key recovery attacks are feasible. If weak parameters are used for key generation it is possible to generate a weak keypair, however, it will not produce signatures that verify with respect to the correct parameters. So the common PKI practice of signing a certificate request with the subject key will catch keys generated with invalid parameters.

Creating signatures with invalid parameters can also be a security risk. Suppose a set of weak LowMC parameters were used, so that the signing algorithm proves knowledge of the signing key k, for an new circuit E', i.e., $E'_k(x) = \mathsf{pk}$, where E' is LowMC with invalid parameters. The signature would be invalid in most cases, unless key generation and verification also used E'. However, the invalid signature contains enough information to recover $E'_k(x)$, from which it may be possible to recover k.

11 Efficiency and Memory Usage

This section gives performance benchmarks of the Picnic signature scheme. A single core/thread was used for all benchmarks. All times are in milliseconds. We benchmark three implementations:

Reference. An expository C implementation. Makes no performance optimizations.

Optimized-C. A somewhat optimized implementation using C only.

Optimized. An optimized implementation that uses processor-specific compiler intrinsics for vector instructions, e.g. SSE2 and AVX2 on Intel x86-64 and NEON on ARM v8.

The **optimized-C** and **optimized** implementations are constant-time.

11.1 Description of the Benchmark Platforms

11.1.1 Platform A

The primary benchmarking platform, **Platform A**, has the following specifications:

CPU. Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz

Memory. 16 GB

OS. Ubuntu 18.04.1

Compiler. GCC 7.3.0

Intel Turbo Boost (dynamic frequency scaling) was disabled. For comparison, OpenSSL version 1.0.2g reports 0.03 ms for ECDSA signing and 0.08 ms for verification on Platform A[8].

11.1.2 Platform B

The secondary benchmarking platform, **Platform B** (Raspberry Pi 3 Model B), has the following specifications:

CPU. Quad Core 1.2GHz Broadcom BCM2837 64-bit CPU, ARM Cortex A53 (ARMv8)

Memory. 1 GB RAM

OS. openSUSE Tumbleweed 20180905

Compiler. GCC 8.2.1

For comparison, OpenSSL version 1.0.2j reports 11 ms for ECDSA signing and 40 ms for verification on Platform B.

11.1.3 Platform C

A third platform, **Platform C**, is an older x64-based system, has the following specifications:

CPU. Intel(R) Xeon(R) CPU E31230 @ 3.20GHz

Memory. 8 GB

OS. Ubuntu 18.04.2 LTS

Compiler. GCC 7.3.0

For comparison, OpenSSL 1.0.2g reports 0.05 ms for ECDSA signing and 0.12 ms for ECDSA verification on Platform C.

[8] As reported by the command `openssl speed ecdsap256`

11.2 Description of the Benchmarking Methodology

Timings results for key generation, signing and signature verification were averaged over 1000 runs on x64 and 100 runs on ARM.

On Platforms A and C we measured CPU cycles using the `perf_event` performance monitoring subsystem of the Linux kernel. On Platform B CPU cycles were measured using the hardware performance counter available via the `MRS` instruction. The optimized implementations will use the GCC link time optimization (`-flto`) feature by default if it is available (it was available on all three of our benchmarking platforms).

11.3 Benchmark Results: Sizes

In Table 6 we give the size of Picnic keys, and signatures. Note that these are the same for all implementations.

Parameter Set	Pub. key	Priv. key	Sig (max)	Sig (avg., std. dev.)
`Picnic-L1-FS`	32	16	34032	32838, 107
`Picnic-L1-UR`	32	16	53961	
`Picnic2-L1-FS`	32	16	13802	12359, 213
`Picnic-L3-FS`	48	24	76772	74134, 198
`Picnic-L3-UR`	48	24	121845	
`Picnic2-L3-FS`	48	24	29750	27173, 443
`Picnic-L5-FS`	64	32	132856	128176, 315
`Picnic-L5-UR`	64	32	209506	
`Picnic2-L5-FS`	64	32	54732	46282, 613

Table 6: Key and signature sizes (in bytes) by security level. For the FS variants, the signature length varies based on the challenge, therefore we give the maximum possible size, along with the average size and standard deviation computed over 100 signatures.

11.4 Benchmark Results: Timings

In this section we describe the time required for various operations, on the three benchmark platforms. In all tables presented in this section we give the timing information as milliseconds and CPU cycles.

Parameters	Keygen	Sign	Verify
Picnic-L1-FS	0.05	38.31	25.18
(cycles)	182135	137915579	90638943
Picnic-L1-UR	0.04	47.93	32.36
(cycles)	160250	172560379	116494076
Picnic2-L1-FS	0.04	851.85	515.93
(cycles)	149749	3066663719	1857340295
Picnic-L3-FS	0.12	128.30	84.70
(cycles)	427625	461886045	304933466
Picnic-L3-UR	0.11	152.51	102.36
(cycles)	392130	549035979	368491954
Picnic2-L3-FS	0.10	2830.60	1538.25
(cycles)	362481	10190171124	5537696230
Picnic-L5-FS	0.21	308.96	204.53
(cycles)	746566	1112238170	736317245
Picnic-L5-UR	0.21	342.98	230.12
(cycles)	752867	1234712908	828445835
Picnic2-L5-FS	0.19	7080.01	3595.40
(cycles)	691790	25488037138	12943455830

Table 7: Benchmarks for the **reference** implementation, on benchmark Platform A.

In Tables 7, 8, and 9 we present the benchmark results of all implementations on Platform A. On this platform we observe speed improvements of the **optimized** implementation over the **optimized-C** implementation by a factor of about 2 for Picnic and 1.5 for Picnic2. This is largely explained by support for AVX2 vector instructions on Platform A. Currently, the Picnic2 implementation makes less use of vector instructions than in Picnic parameters.

Next we give the results of the evaluation of our implementations on Platform B in Tables 10, 11, and 12. Again we observe improved performance figures for the **optimized** implementation, but they are not as significant as on Platform A. GCC's optimizer vectorizes the code on Platform B more aggressively for **optimized-C** implementation and thus the performance gains of the **optimized** implementation are smaller.

Finally, Tables 13, 14 and 15 show benchmarks for Platform C. Here, the **optimized-C** and **optimized** implementations are very close, since this

Parameters	Keygen	Sign	Verify
Picnic-L1-FS	0.00	2.82	2.34
(cycles)	13102	10135801	8421534
Picnic-L1-UR	0.00	3.49	2.87
(cycles)	14007	12552077	10344492
Picnic2-L1-FS	0.01	106.91	42.64
(cycles)	22607	384870013	153501082
Picnic-L3-FS	0.01	6.74	5.66
(cycles)	25710	24256134	20363282
Picnic-L3-UR	0.01	8.64	7.12
(cycles)	27271	31109463	25615076
Picnic2-L3-FS	0.01	328.68	99.27
(cycles)	30972	1183244054	357386886
Picnic-L5-FS	0.01	12.37	10.59
(cycles)	38418	44526676	38119483
Picnic-L5-UR	0.01	15.02	12.64
(cycles)	39309	54073507	45504820
Picnic2-L5-FS	0.01	708.82	178.63
(cycles)	50681	2551768397	643079749

Table 8: Benchmarks for the **optimized-C** implementation, on benchmark Platform A.

Parameters	Keygen	Sign	Verify
`Picnic-L1-FS`	0.00	1.44	1.15
(cycles)	12391	5191118	4151833
`Picnic-L1-UR`	0.00	1.80	1.45
(cycles)	12613	6471898	5236920
`Picnic2-L1-FS`	0.01	63.87	27.93
(cycles)	21026	229947918	100546772
`Picnic-L3-FS`	0.00	3.37	2.79
(cycles)	14441	12145246	10055243
`Picnic-L3-UR`	0.00	4.28	3.55
(cycles)	15181	15410759	12764570
`Picnic2-L3-FS`	0.01	182.76	62.16
(cycles)	20160	657944759	223785326
`Picnic-L5-FS`	0.01	5.87	4.92
(cycles)	20443	21135646	17708704
`Picnic-L5-UR`	0.01	7.34	6.18
(cycles)	20803	26428140	22247160
`Picnic2-L5-FS`	0.01	374.09	107.68
(cycles)	35716	1346724260	387637876

Table 9: Benchmarks for the **optimized** implementation, on benchmark Platform A.

Parameters	Keygen	Sign	Verify
`Picnic-L1-FS`	0.41	300.48	196.38
(cycles)	491452	360581357	235657525
`Picnic-L1-UR`	0.38	356.09	238.06
(cycles)	457703	427311119	285668078
`Picnic2-L1-FS`	0.32	6143.37	3668.65
(cycles)	379151	7372048160	4402376487
`Picnic-L3-FS`	0.93	1030.39	681.57
(cycles)	1117910	1236464123	817889494
`Picnic-L3-UR`	0.94	1194.93	802.89
(cycles)	1128756	1433913576	963465350
`Picnic2-L3-FS`	0.84	20253.54	10594.90
(cycles)	1007441	24304251305	12713878163
`Picnic-L5-FS`	1.82	2600.40	1726.59
(cycles)	2180092	3120475633	2071908013
`Picnic-L5-UR`	1.85	2832.86	1903.62
(cycles)	2223283	3399435377	2284344835
`Picnic2-L5-FS`	1.72	49321.25	23866.37
(cycles)	2065770	59185502655	28639647451

Table 10: Benchmarks for the **reference** implementation, on benchmark Platform B.

Parameters	Keygen	Sign	Verify
`Picnic-L1-FS`	0.04	16.50	14.18
(cycles)	517030	19797395	17018574
`Picnic-L1-UR`	0.04	19.85	16.94
(cycles)	44960	23817176	20322239
`Picnic2-L1-FS`	0.04	691.72	238.06
(cycles)	44855	830065732	285669555
`Picnic-L3-FS`	0.08	48.52	43.11
(cycles)	90591	58219483	51728618
`Picnic-L3-UR`	0.08	58.01	50.51
(cycles)	90630	69607278	60609890
`Picnic2-L3-FS`	0.07	2150.92	562.78
(cycles)	84626	2581108219	675330804
`Picnic-L5-FS`	0.10	91.25	83.12
(cycles)	119775	109497015	99741418
`Picnic-L5-UR`	0.11	104.50	94.05
(cycles)	135453	125396028	112861188
`Picnic2-L5-FS`	0.12	4714.90	1015.69
(cycles)	139243	5657876778	1218823201

Table 11: Benchmarks for the **optimized-C** implementation, on benchmark Platform B.

Parameters	Keygen	Sign	Verify
Picnic-L1-FS	0.04	15.73	13.32
(cycles)	47742	18870632	15981838
Picnic-L1-UR	0.04	18.11	15.24
(cycles)	42767	21730459	18284627
Picnic2-L1-FS	0.03	660.14	236.05
(cycles)	40827	792170348	283255251
Picnic-L3-FS	0.07	45.24	40.22
(cycles)	86515	54290201	48260659
Picnic-L3-UR	0.07	54.61	47.66
(cycles)	87774	65536464	57190412
Picnic2-L3-FS	0.07	2041.81	560.53
(cycles)	87578	2450174097	672630661
Picnic-L5-FS	0.09	79.44	71.37
(cycles)	110962	95331677	85641892.
Picnic-L5-UR	0.12	98.39	87.60
(cycles)	142196	118071213	105117874
Picnic2-L5-FS	0.12	4446.97	1001.88
(cycles)	141311	5336358266	1202255981

Table 12: Benchmarks for the **optimized** implementation, on benchmark Platform B.

Parameters	Keygen	Sign	Verify
`Picnic-L1-FS`	0.07	54.18	35.57
(cycles)	2267156	1733910732	113827369
`Picnic-L1-UR`	0.06	65.35	43.85
(cycles)	2011758	209108687	140323212
`Picnic2-L1-FS`	0.06	1380.24	906.94
(cycles)	1900840	4416763062	2902196149
`Picnic-L3-FS`	0.17	186.24	122.99
(cycles)	544439	595962163	393554832
`Picnic-L3-UR`	0.16	219.66	148.08
(cycles)	506233	702919158	473848875
`Picnic2-L3-FS`	0.15	4376.62	2545.02
(cycles)	488005	14005170619	8144070561
`Picnic-L5-FS`	0.31	467.45	309.81
(cycles)	1004010	1495844637	991399622
`Picnic-L5-UR`	0.32	514.08	344.88
(cycles)	1012476	1645054261	1103615594
`Picnic2-L5-FS`	0.31	10509.07	5713.30
(cycles)	982634	33629020802	18282558758

Table 13: Benchmarks for the **reference** implementation, on benchmark Platform C.

platform does have AVX2 support (and our implementation does not have an optimized implementation using only AVX and SSE).

11.5 Memory Requirements

In this section we give the memory requirements for our implementations. The memory requirements of an implementation are assumed to be the same for all platforms.

We note that at this time, our implementations of the Picnic2 parameter sets have not been optimized to reduce memory consumption.

Parameters	Keygen	Sign	Verify
`Picnic-L1-FS`	0.01	4.41	3.56
(cycles)	17373	14118186	11400107
`Picnic-L1-UR`	0.01	5.66	4.53
(cycles)	17793	18102403	14487939
`Picnic2-L1-FS`	0.01	180.00	88.63
(cycles)	25635	575998506	283601036
`Picnic-L3-FS`	0.01	10.88	9.29
(cycles)	33241	34829503	29730907
`Picnic-L3-UR`	0.01	14.42	12.01
(cycles)	33144	46138285	38442986
`Picnic2-L3-FS`	0.01	521.31	196.81
(cycles)	37724	1668205946	629804850
`Picnic-L5-FS`	0.01	19.52	16.81
(cycles)	45311	62457936	53796359
`Picnic-L5-UR`	0.01	24.48	21.07
(cycles)	45498	78350160	67436627
`Picnic2-L5-FS`	0.02	1095.25	344.55
(cycles)	58268	3504807600	1102558535

Table 14: Benchmarks for the **optimized-C** implementation, on benchmark Platform C.

Parameters	Keygen	Sign	Verify
`Picnic-L1-FS`	0.01	4.20	3.40
(cycles)	17676	13435455	10886965
`Picnic-L1-UR`	0.01	5.48	4.37
(cycles)	17781	17539617	13976186
`Picnic2-L1-FS`	0.01	153.99	86.40
(cycles)	24729	492772673	276474753
`Picnic-L3-FS`	0.01	10.17	8.36
(cycles)	30355	32532479	26762749
`Picnic-L3-UR`	0.01	13.70	11.10
(cycles)	30785	43827423	35505712
`Picnic2-L3-FS`	0.01	420.59	189.56
(cycles)	36181	1345886906	606593762
`Picnic-L5-FS`	0.01	17.67	14.67
(cycles)	39579	56541841	46933599
`Picnic-L5-UR`	0.01	22.64	18.97
(cycles)	39663	72449921	60707627
`Picnic2-L5-FS`	0.02	847.51	327.60
(cycles)	53246	2712044663	1048318112

Table 15: Benchmarks for the **optimized** implementation, on benchmark Platform C.

Parameter set	Sign	Verify
`Picnic-L1-FS`	190,148	132,247
`Picnic-L1-UR`	268,080	239,367
`Picnic2-L1-FS`	7,916,440	6,649,248
`Picnic-L3-FS`	380,380	290,507
`Picnic-L3-UR`	552,752	492,779
`Picnic2-L3-UR`	19,060,800	18,610,464
`Picnic-L5-FS`	632,544	468,162
`Picnic-L5-UR`	984,696	733,426
`Picnic2-L5-UR`	33,937,992	32,194,200

Table 16: Peak memory usage (stack and heap combined) of reference implementation, in bytes. This excludes memory used for the LowMC constants, which was stored as static data in program binary (see §11.6).

11.5.1 Reference Implementation Detailed Memory Usage

Memory usage was benchmarked using the Valgrind[9] tool Massif. Massif was run on an example program, that generates a key pair, creates a signature, then verifies it, using the API in picnic.h. Then the tool `massifcherrypick`[10] was used to determine the peak memory usage of specific functions.

Massif was invoked with the command:

```
valgrind --tool=massif --stacks=yes ./example
```

When creating a signature, peak memory usage ranged from about 190K to 33M bytes, as shown in Table 16, while verification ranged from about 132K to 32M bytes.

Massif measures memory usage by sampling so there is some variability in these measurements. The variance was low so these are a reasonable estimate, since we're only interested in peak usage.

11.5.2 Optimized Implementation Detailed Memory Usage

With the same methodology as used for the reference implementation, peak memory usage data was generated with `valgrind`. The data is presented in Table 17.

[9] http://valgrind.org/docs/manual/ms-manual.html

[10] https://github.com/lnishan/massif-cherrypick

Parameter set	Sign	Verify
Picnic-L1-FS	136,912	84,352
Picnic-L1-UR	200,203	131,218
Picnic2-L1-FS	8,253,952	7,576,712
Picnic-L3-FS	284,632	163,560
Picnic-L3-UR	427,747	269,498
Picnic2-L3-FS	19,840,560	10,170,216
Picnic-L5-FS	463,016	259,784
Picnic-L5-UR	706,982	441,116
Picnic2-L5-FS	34,522,576	34,058,656

Table 17: Peak memory usage (stack and heap combined) of the optimized implementation, in bytes. This excludes memory used for the LowMC constants, which was stored as static data in program binary (see §11.6).

11.6 Size of Precomputed Constants and Data

Size of LowMC Constants. The LowMC block cipher uses a large amount of constant data when compared to traditional block ciphers. This data may be computed on-the-fly as needed, or precomputed and stored in advance. All our implementations compile this data into the binary.

We did not investigate the cost of re-computing the LowMC constants at runtime. The output of the Grain LSFR is used as a self-shrinking generator to create the constants.

The **optimized-C** and **optimized** implementations use an alternative but equivalent representation of LowMC. They implement an optimized linear layer [DKP+19], which allows to greatly reduced the size of the LowMC constants and also gives a significant performance boost. The sizes of those matrices are are given in Tables 19 and 19 in the Key Matrices and Linear Matrices columns. The total reduction in size of precomputed constants ranges from 2.4 to 4.9 times.

11.7 TLS Performance

As described in Section 9.1 we used OQS, OpenSSL and Apache to benchmark Picnic and other post-quantum cryptography in the context of HTTPS.

Params	Linear Matrices	Round Constants	Key Matrices	Total
L1	40,960	320	43,008	84,288
L3	138,240	720	142,848	281,808
L5	311,296	1,216	319,488	632,000

Table 18: Size of constant data (in bytes) required by LowMC as used by the reference implementation, with the parameters used in Picnic at security levels L1, L3 and L5.

Params	Linear Matrices	Round Constants	Key Matrices	Total
L1	20,288	128	14,336	34,752
L3	61,824	160	30,720	92,704
L5	79,232	192	49,152	128,576

Table 19: Size of constant data (in bytes) required by LowMC as used by the optimized implementations, with the parameters used in Picnic at security levels L1, L3 and L5.

We set up a web server in Microsoft Azure (a standard D2s v3 instance[11]), and hosted HTML files of size 45B, 1KB, 10KB, 100KB and 1MB. The ciphersuites we benchmarked were ECDHE_RSA (as a baseline), LWEFRODO_RSA, LWEFRODO_PICNIC, SIDH_RSA and SIDH_PICNIC. The key exchange algorithms LWEFRODO and SIDH are post-quantum candidates: security of LWEFRODO is based on the lattice problem *learning with errors* and security of SIDH is based on the *supersingular isogeny Diffie-Hellman* problem. Details of these schemes are available on the OQS project website, and the corresponding submissions to the NIST PQ Project. Note that we used the versions of SIDH and Frodo that were in OQS at the time, and these were probably behind the versions submitted to NIST. This should not affect our conclusions regarding the performance of Picnic signatures in TLS.

All ciphersuites used AES256-GCM-SHA384 for the symmetric-key primitives (e.g., ECDHE-RSA-AES256-GCM-SHA384, LWEFRODO-PICNIC--AES256-GCM-SHA384, etc.). All instances of Picnic used the L1-FS parameter set. The server certificate was signed with Picnic, and had a Picnic subject public key. This doubles the bandwidth increase due to Picnic, from about 32KB to 64KB. However, in actual deployment we expect the CA sig-

[11] The D2s v3 instance has 2 vcpus, 8 GB memory. The operating system we used was 16.04.3 LTS (GNU/Linux 4.11.0-1015-azure x86_64).

nature to be created with a stateful hash-based signature scheme (like LMS or XMSS), and to be 1-5KB in size. Therefore, the performance given here is arguably pessimistic. However, support for stateful hash-based signatures was not present in OQS, and adding support for them with the time and resources available to us was not possible.

Our experiment used two client machines. The first was a Lenovo Thinkpad x270 connected over WiFi in a home with cable internet service. This is labeled "slow network" in Table 20. The second was a Dell Poweredge R710 server on the Microsoft campus, labeled "fast network" in Table 20. The fetch times of the faster machine is about 2-3 times as fast as the slower one. Both were running Ubuntu 17.10.

We then ran the benchmarking program http_load[12] (also compiled with OQS-OpenSSL), with the parameters `-parallel 1 -seconds 60`. This uses a single thread to fetch a URL repeatedly for approximately 60 seconds. We then computed the mean fetch time as the total time taken divided by the number of fetches completed.

The results of Table 20 show that for large pages, all ciphersuites have similar performance. This can be explained by (i) the cost of network communication dominating the CPU time and (ii) the bandwidth overhead of the ciphersuites becomes negligible for large pages. For smaller pages, the additional bandwidth cost of Picnic signatures is apparent. For example, for the 45B page, the LWEFRODO-PICNIC suite is 1.7X slower than LWEFRODO-RSA on the slower network, while on the faster network it is 1.4x slower. For every network speed there will be a page size where the cost of Picnic vs. RSA does not affect performance. For our client on the slower network this is somewhere between 100K and 1M, and for our client on the faster network it is somewhere between 10K and 100K.

Limitations. This provides only a limited understanding of how changing from RSA signatures to Picnic signatures would impact web and TLS performance from the client perspective. The benchmarks here could be improved by fetching real web sites, which contain a mix of small and large objects, hosted on a set of servers, and the TLS context is re-used for multiple requests. We also only considered the case of an idle server, able to dedicate all of its resources to serving a client request. As costs may increase on the

[12] ACME Labs, http://www.acme.com/software/http_load/, we used version `09Mar2016`, and modified it to use a newer version of OpenSSL, we had to change the way OpenSSL was initialized.

Ciphersuite	Page Size	Mean fetch time Slow network	Mean fetch time Fast network
ECDHE-RSA	45B	0.470	0.299
	1K	0.526	0.299
	10K	0.527	0.300
	100K	1.226	0.452
	1M	3.001	0.750
LWEFRODO-RSA	45B	0.578	0.366
	1K	0.660	0.365
	10K	0.645	0.369
	100K	1.335	0.518
	1M	2.874	0.741
LWEFRODO-PICNIC	45B	0.984	0.513
	1K	1.118	0.513
	10K	1.158	0.519
	100K	1.733	0.594
	1M	3.337	0.764
SIDH-RSA	45B	0.655	0.385
	1K	0.698	0.385
	10K	0.729	0.387
	100K	1.370	0.541
	1M	3.758	0.836
SIDH-PICNIC	45B	1.084	0.523
	1K	1.106	0.524
	10K	1.093	0.528
	100K	1.738	0.600
	1M	3.158	0.802

Table 20: Time in seconds to fetch pages of varying size over HTTPS when different ciphersuites are used for TLS. The Picnic L1-FS parameter set is used. The symmetric algorithms of each ciphersuite were the same, AES256-GCM-SHA384.

server (bandwidth, memory and CPU), it would also be instructive to measure these, for example, by concurrently making many requests on a server and measuring requests per second for the different ciphersuites.

Picnic implementation. These experiments were done with the optimized Picnic implementation in the fall of 2017, and the experiment was not repeated with the latest implementation for version 2.0 of this document, or with the Picnic2 parameter sets.

References

[ABF+17] Christopher Ambrose, Joppe W. Bos, Bjorn Fay, Marc Joye, Manfred Lochter, and Bruce Murray. Differential attacks on deterministic signatures. Cryptology ePrint Archive, Report 2017/975, 2017.

[AGR+16] Martin R. Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. MiMC: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. In *ASIACRYPT*, pages 191–219, 2016.

[ARS+15] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. In *EUROCRYPT*, 2015.

[ARS+16] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. *IACR Cryptology ePrint Archive*, 2016:687, 2016.

[BB17] Gustavo Banegas and Daniel J. Bernstein. Low-communication parallel quantum multi-target preimage search. Cryptology ePrint Archive, Report 2017/789, 2017. http://eprint.iacr.org/2017/789.

[BCG+12] Julia Borghoff, Anne Canteaut, Tim Güneysu, Elif Bilge Kavun, Miroslav Knezevic, Lars R. Knudsen, Gregor Leander, Ventzislav Nikov, Christof Paar, Christian Rechberger, Peter Rombouts, Søren S. Thomsen, and Tolga Yalçin. PRINCE - a low-latency block cipher for pervasive computing applications - extended abstract. In *ASIACRYPT*, 2012.

[BDP+18] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, Ronny Van Keer, and Benoît Viguier. KangarooTwelve: Fast hashing based on keccak-p. In Bart Preneel and Frederik Vercauteren, editors, *ACNS 18*, volume 10892 of *LNCS*, pages 400–418. Springer, Heidelberg, July 2018.

[Ber09] Daniel J. Bernstein. Cost analysis of hash collisions: Will quantum computers make SHARCS obsolete? 2009. http://cr.yp.to/hash/collisioncost-20090823.pdf.

[BHT98] Gilles Brassard, Peter Høyer, and Alain Tapp. Quantum cryptanalysis of hash and claw-free functions. In Claudio L. Lucchesi and Arnaldo V. Moura, editors, *LATIN 1998*, volume 1380 of *LNCS*, pages 163–169. Springer, Heidelberg, April 1998.

[Bih02] Eli Biham. How to decrypt or even substitute des-encrypted messages in 2^{28} steps. *Inf. Process. Lett.*, 84(3):117–124, 2002.

[Ble00] Daniel Bleichenbacher. On the generation of one-time keys in dl signature schemes. Presentation at IEEE P1363 Working Group meeting, November 2000. Unpublished., 2000.

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

[BMS05] Alex Biryukov, Sourav Mukhopadhyay, and Palash Sarkar. Improved time-memory trade-offs with multiple data. In *Selected Areas in Cryptography, 12th International Workshop, SAC 2005, Kingston, ON, Canada, August 11-12, 2005, Revised Selected Papers*, pages 110–127, 2005.

[BPW12] David Bernhard, Olivier Pereira, and Bogdan Warinschi. How not to prove yourself: Pitfalls of the Fiat-Shamir heuristic and applications to Helios. In Xiaoyun Wang and Kazue Sako, editors, *ASIACRYPT 2012*, volume 7658 of *LNCS*, pages 626–643. Springer, Heidelberg, December 2012.

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM CCS*, 1993.

[CCF$^+$16] Anne Canteaut, Sergiu Carpov, Caroline Fontaine, Tancrède Lepoint, María Naya-Plasencia, Pascal Paillier, and Renaud Sirdey. Stream ciphers: A practical solution for efficient homomorphic-ciphertext compression. In *FSE*, 2016.

[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 Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 17*, pages 1825–1842. ACM Press, October / November 2017.

[CDS94] Ronald Cramer, Ivan Damgård, and Berry Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. In *CRYPTO*, 1994.

[CGH98] Ran Canetti, Oded Goldreich, and Shai Halevi. The random oracle methodology, revisited (preliminary version). In *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing, Dallas, Texas, USA, May 23-26, 1998*, pages 209–218, 1998.

[CGP$^+$12] Claude Carlet, Louis Goubin, Emmanuel Prouff, Michaël Quisquater, and Matthieu Rivain. Higher-order masking schemes for s-boxes. In *FSE*, 2012.

[Dam10] Ivan Damgård. On Σ-protocols. 2010. http://www.cs.au.dk/~ivan/Sigma.pdf.

[DFMS19] Jelle Don, Serge Fehr, Christian Majenz, and Christian Schaffner. Security of the fiat-shamir transformation in the quantum random-oracle model. Cryptology ePrint Archive, Report 2019/190, 2019.

[DKP$^+$19] Itai Dinur, Daniel Kales, Angela Promitzer, Sebastian Ramacher, and Christian Rechberger. Linear Equivalence of Block Ciphers with Partial Non-Linear Layers: Application to LowMC. In *EUROCRYPT*, 2019. To appear.

[DP08] Christophe De Cannière and Bart Preneel. Trivium. In *New Stream Cipher Designs - The eSTREAM Finalists*. 2008.

[DPVAR00] Joan Daemen, Michaël Peeters, Gilles Van Assche, and Vincent Rijmen. Nessie proposal: Noekeon. In *First Open NESSIE Workshop*, 2000.

[FKMV12] Sebastian Faust, Markulf Kohlweiss, Giorgia Azzurra Marson, and Daniele Venturi. On the non-malleability of the fiat-shamir transform. In *INDOCRYPT*, 2012.

[FS86] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO*, pages 186–194, 1986.

[GLSV14] Vincent Grosso, Gaëtan Leurent, François-Xavier Standaert, and Kerem Varici. Ls-designs: Bitslice encryption for efficient masked software implementations. In *FSE*, 2014.

[GMO16a] Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. ZKBoo: Faster zero-knowledge for boolean circuits. Cryptology ePrint Archive, Report 2016/163, 2016. http://eprint.iacr.org/2016/163.

[GMO16b] Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. ZKBoo: Faster zero-knowledge for boolean circuits. In *USENIX Security*, 2016.

[GMS02] Steven D. Galbraith, John Malone-Lee, and Nigel P. Smart. Public key signatures in the multi-user setting. *Inf. Process. Lett.*, 83(5):263–266, 2002.

[GP99] Louis Goubin and Jacques Patarin. DES and differential power analysis (the "duplication" method). In Çetin Kaya Koç and Christof Paar, editors, *CHES'99*, volume 1717 of *LNCS*, pages 158–172. Springer, Heidelberg, August 1999.

[Gro96] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *STOC*, 1996.

[Hel80] Martin Hellman. A cryptanalytic time-memory trade-off. *IEEE transactions on Information Theory*, 26(4):401–406, 1980.

[HS01] Nick Howgrave-Graham and Nigel P. Smart. Lattice attacks on digital signature schemes. *Des. Codes Cryptography*, 23(3):283–290, 2001.

[IKOS09] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge proofs from secure multiparty computation. *SIAM Journal on Computing*, 39(3):1121–1152, 2009.

[Kat10] Jonathan Katz. *Digital Signatures*. Springer, 2010.

[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 18*, pages 525–537. ACM Press, October 2018.

[KM15] Neal Koblitz and Alfred J. Menezes. The random oracle model: a twenty-year retrospective. *Des. Codes Cryptography*, 77(2-3):587–610, 2015.

[LZ19] Qipeng Liu and Mark Zhandry. Revisiting post-quantum fiat-shamir. Cryptology ePrint Archive, Report 2019/262, 2019.

[MJSC16] Pierrick Méaux, Anthony Journault, François-Xavier Standaert, and Claude Carlet. Towards stream ciphers for efficient FHE with low-noise ciphertexts. In *EUROCRYPT*, 2016.

[MS04] Alfred Menezes and Nigel P. Smart. Security of signature schemes in a multi-user setting. *Des. Codes Cryptography*, 33(3):261–274, 2004.

[NIS15] NIST. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. National Institute of Standards and Technology (NIST), FIPS PUB 202, U.S. Department of Commerce, 2015.

[NNL01] Dalit Naor, Moni Naor, and Jeffery Lotspiech. Revocation and tracing schemes for stateless receivers. In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 41–62. Springer, Heidelberg, August 2001.

[PSS$^+$17] Damian Poddebniak, Juraj Somorovsky, Sebastian Schinzel, Manfred Lochter, and Paul Rosler. Attacking deterministic signature schemes using fault attacks. Cryptology ePrint Archive, Report 2017/1014, 2017.

[Tea19a] The Picnic Design Team. The Picnic signature algorithm specification, March 2019. Version 2.1, Available at https://microsoft.github.io/Picnic/.

[Tea19b] The Picnic Design Team. The Picnic website, March 2019. https://microsoft.github.io/Picnic/.

[Unr12] Dominique Unruh. Quantum proofs of knowledge. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 135–152. Springer, Heidelberg, April 2012.

[Unr15] Dominique Unruh. Non-interactive zero-knowledge proofs in the quantum random oracle model. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 755–784. Springer, Heidelberg, April 2015.

[Unr16] Dominique Unruh. Computationally binding quantum commitments. In *EUROCRYPT*, 2016.

[vOW94] Paul C. van Oorschot and Michael J. Wiener. Parallel collision search with application to hash functions and discrete logarithms. In *CCS '94, Proceedings of the 2nd ACM Conference on Computer and Communications Security, Fairfax, Virginia, USA, November 2-4, 1994.*, pages 210–218, 1994.

A Change History

Version 1.0 November 2017. Initial version, part of the first round submission to the NIST Post-Quantum Cryptography Standardization Process.

Version 2.0 March 2019. Update for the second round submission to the NIST Post-Quantum Cryptography Standardization Process. The main change is to add the background material and security analysis of the Picnic2 parameter sets.