The Picnic Signature Scheme

Design Document

Melissa Chase, David Derler, Steven Goldfeder,
Claudio Orlandi, Sebastian Ramacher, Christian Rechberger,
Daniel Slamanig, Greg Zaverucha

November 29, 2017
Version 1.0

Contents

1 Introduction

Picnic is a signature scheme which is designed to provide security against attacks by quantum computers, in addition to attacks by classical computers. The building blocks are a zero-knowledge proof system (with post-quantum security), and symmetric key primitives like hash functions and block ciphers, with well-understood post-quantum security. In particular, Picnic does not require number-theoretic, or structured hardness assumptions.

In this document we first present the building blocks of the Picnic signature scheme in Section 2. Second, in Section 3 we present two variants of the Picnic signature scheme and various optimizations to the building blocks that we can employ in our schemes. Third we specify the parameters for some building blocks in Section 4 (and leave complete details of the parameters to the specification document). In the subsequent Section 5 of this document we include the formal security proofs for the proposed instantiations of Picnic. Section 6 presents an analysis of the algorithm with respect to known attacks and Section 7 provides a thorough description of the expected security strength. Finally, Section 8 discuss advantages and limitations, Section 9 discusses additional security properties and Section 10 presents results on efficiency and memory usage of the Picnic scheme.

1.1 The Picnic Design Team

Picnic was designed collaboratively by the following group of people.

Melissa Chase, Microsoft
David Derler, Graz University of Technology
Steven Goldfeder, Princeton
Claudio Orlandi, Aarhus University
Sebastian Ramacher, Graz University of Technology
Christian Rechberger, Graz University of Technology & DTU
Daniel Slamanig, AIT Austrian Institute of Technology
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), and Alexander Grass and Angela Promitzer (contributions to the optimized implementation).

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 three algorithms $\mathsf{KG}, \mathsf{Com}, \mathsf{Open}$ with the following properties:

$\mathsf{KG}(1^\kappa)$: The key generation algorithm, on input the security parameter κ, outputs a public key pk (we henceforth assume pk to be an implicit input to the subsequent algorithms).

$\mathsf{Com}(M)$: On input of 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' for a randomly generated $\mathsf{pk} \leftarrow \mathsf{KG}(1^\kappa)$ the probability ensembles $\{C, \mathsf{pk} : (C, D) \leftarrow \mathsf{Com}(M)\}_{\kappa \in \mathbb{N}}$ and $\{C, \mathsf{pk} : (C, D) \leftarrow \mathsf{Com}(M')\}_{\kappa \in \mathbb{N}}$ 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 on input a random public key generated by $\mathsf{KG}(1^\kappa)$ 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 $R \xleftarrow{R} \{0,1\}^\kappa$ 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 language L 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

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

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 to $(s-1)/t$ of the possible challenges, where $t = |\mathsf{E}|$ is the size of the challenge space. In case this value is too large, it is possible to reduce the soundness error using well known properties of Σ-protocols which we restate here for completeness (see [Dam10, CDS94] for details).

Lemma 2.3. *The properties of Σ-protocols are invariant under parallel repetition. In particular, the ℓ-fold parallel repetition of a Σ-protocol for relation R with challenge length t yields a new Σ-protocol with challenge length ℓt.*

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

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

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 $\mathsf{S}_{\mathsf{R}}, \mathsf{Sim}$ such that for all (quantum) PPT adversaries $\mathcal{A}$

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

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 $\mathsf{Sim}, \mathsf{S}_{\mathsf{R}}$ 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 $\mathsf{S}_{\mathsf{R}}, \mathsf{Sim}$ satisfying the zero-knowledge definition and a PPT (quantum) extractor such that for all (quantum) PPT adversaries $\mathcal{A}$

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

is negligible in λ, where $\mathsf{S}_{\mathsf{R}}, \mathsf{Sim}$, 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 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.

2.5 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 $(\mathtt{Prove}_H, \mathtt{Verify}_H)$ defined as follows:

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

```
1. For $i \in [t]$:
    (a) Start $\mathsf{P}$ on $(1^\kappa, x, w)$ and obtain first message $\mathsf{a}_i$.
    (b) For $j \in [M]$, set $\mathsf{e}_{i,j} \overset{R}{\leftarrow} \mathsf{E} \backslash \{\mathsf{e}_{i,1}, \ldots, \mathsf{e}_{i,j-1}\}$ 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 $(J_1, \ldots, J_t) \leftarrow H(m, (\mathsf{a}_i)_{i \in [t]}, (\mathsf{e}_{i,j})_{(i,j) \in [t] \times [M]}, (g_{i,j})_{(i,j) \in [t] \times [M]})$
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 $(J_1, \ldots, J_t) \leftarrow H(m, (\mathsf{a}_i)_{i \in [t]}, (\mathsf{e}_{i,j})_{(i,j) \in [t] \times [M]}, (g_{i,j})_{(i,j) \in [t] \times [M]})$
2. For $i \in [t]$ check that all $\mathsf{e}_{i,1}, \ldots, \mathsf{e}_{i,M}$ are pairwise distinct.
3. For $i \in [t]$ check whether $V$ accepts the proof with respect to $x$,
   first message $\mathsf{a}_i$, challenge $\mathsf{e}_{i,J_i}$ and response $\mathsf{z}_i$.
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.

2.6 (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 2.6 ((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,

$\mathcal{D}$:

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

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 $\texttt{Output}_i$ 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

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

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

Experiment 1: Decomposition Experiment

a $(2,3)$-*decomposition* of ϕ if the following two properties hold when running $\mathsf{EXP}_{\mathsf{decomp}}^{(\phi,\mathsf{x})}$:

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

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

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

- $(x_1, x_2, x_3) \leftarrow \texttt{Share}(x, k_1, k_2, k_3)$ samples random $x_1, x_2, x_3 \in R^m$ such that $x_1 + x_2 + x_3 = x$.
- $y_i \leftarrow \texttt{Output}_i(\mathsf{view}_i^{(n)})$ selects the ℓ output wires of the circuit as stored in the view $\mathsf{view}_i^{(n)}$.
- $y \leftarrow \texttt{Reconstruct}(y_1, y_2, y_3) = y_1 + y_2 + y_3$
- $\mathsf{view}_i^{(j+1)} \leftarrow \texttt{Update}_i^{(j)}(\mathsf{view}_i^{(j)}, \mathsf{view}_{i+1}^{(j)}, k_i, k_{i+1})$ 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 ($w_c = w_a \cdot w_b$):

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

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 2.6. 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}.

2.7 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.

$\mathsf{P.Commit}$:

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 $(\mathsf{a}^{(i)})_{i\in[t]}$.

$\mathsf{P.Prove}$: 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) $\mathsf{View}^{(i)}_e \leftarrow \mathtt{Update}(\ldots\mathtt{Update}(x^{(i)}_{\mathsf{e}^{(i)}}, x^{(i)}_{e+1}, k^{(i)}_e, k^{(i)}_{e+1})\ldots)$

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

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

   (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

```
$\mathsf{Prove}_H(1^\kappa, y, x)$ :
    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 $\pi$ 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 $\mathsf{Open}(C_j^{(i)}, D_j^{(i)}) = \bot$, output $\mathsf{Reject}$. 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 $\mathsf{Reject}$. 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 $\mathsf{Reject}$. 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 $\mathsf{Reject}$.
    5. Output $\mathsf{Accept}$.
```

Scheme 3: The ZKBoo non-interactive proof system.

pseudorandomly as:

$$(x_1, x_2, x_3) \leftarrow \mathtt{Share}(x, k_1, k_2, k_3) := \\ x_1 = R_1(0),\ x_2 = R_2(0),\ x_3 = x - x_1 - x_2.$$

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 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.

2.8 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.

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 4.1.

2.9 Signature Schemes

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

Definition 2.7 (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 2.8 ($\mathsf{EUF\text{-}CMA}$)**.** 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 2.9 ($\mathsf{sEUF\text{-}CMA}$)**.** 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 $\mathcal{Q}^{\mathsf{Sign}}$.

3 The picnic-FS and picnic-UR Signature Schemes

Let us recall that the Picnic signature scheme essentially uses a Fiat-Shamir (FS) transformed, or Unruh (UR) transformed, respectively, version of the ZKB++ protocol to prove knowledge of a witness with respect to a language L_R with associated witness relation R of pre-images of a one-way function $f_u : \mathsf{D}_\kappa \to \mathsf{R}_\kappa$ sampled uniformly at random from a family of one-way functions $\{f_u\}_{u \in \mathsf{K}_\kappa}$:

$$((y,u),x) \in R \iff y = f_u(x).$$

In the so-obtained signature scheme the public verification key pk contains the image y and a description of f, and the secret signing key sk is a random key x from D_κ.

In Scheme 5 we provide a general description of our two signature schemes. In both schemes `Prove` is implemented with ZKB++, in picnic-FS it is made non-interactive with the FS transform, while in picnic-UR, Unruh's transform is used. For the security analysis of the schemes we refer the reader to Sections 5.2.1 and 5.2.2.

In both schemes we instantiate the one way function family $\{f_u\}$ with LowMC. (See Section 2.8 for an overview and Section 4.1 for a discussion of our parameter choices.) To sample from the family, sample a random message u from the LowMC message space; to compute $f_u(x)$, output the LowMC encryption of u under key x. We will assume that this is a secure one-way function family; Section 6 discusses this assumption in terms of known attacks, and [CCF+16] proves that when the key space and message space are the same size as is the case in our scheme it follows from the assumption that LowMC is a secure block cipher (i.e. a pseudo-random permutation).

$\underline{\mathsf{Gen}(1^\kappa):}$ 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 $p = (r, s) \leftarrow \texttt{Prove}_H((y, u), x)$ 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:

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

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

Scheme 5: Generic description the picnic-FS and picnic-UR signature schemes.

3.1 Instantiation and Optimizations of Unruh's Transform

We can apply Unruh's transform to ZKB++ in a relatively straightforward manner by modifying our protocol. Although ZKB++ has 3-special soundness, whereas Unruh's transform is only proven secure for Σ-protocols with 2-special soundness, the proof is easily modified to 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:

```
$\texttt{Prove}_H(1^\kappa, (x, m), w)$ :
  1. For $i \in [t]$:
     (a) Start $\mathsf{P}$ on $(1^\kappa, x, w)$ and obtain first message $\mathsf{a}_i$.
     (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 $i \in \{1, \dots, t\}$, 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 the public key as input and requesting the required number of output bytes from the XOF. Complete details are given in the specification document.

3.3 Random Tapes

We generate random tapes using SHAKE as a KDF by first hashing the seed and then requesting the required number of output bytes from the XOF. 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 $H : \{0,1\}^* \to \{0,1,2\}^t$ (implemented using SHAKE) and rejection sampling: we split the output bits in pairs of two bits and reject all pairs with both bits set.

3.5 Function G

As explained in Section 3.1, G 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.

4.0.1 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] needs similarly to Trivium 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.

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.

4.1 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].

[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.

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 Number of Parallel Repetitions

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 must set our repetition count to $t := 438$. 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.

5 Formal Security Analysis

In this section we first formally confirm that the modifications/optimizations induced by moving from ZKBoo [GMO16b] to ZKB++ [CDG+17] as presented in Section 2.7 do not impact the security of the Σ-protocol. Then we move on to formally analyze the security of the Picnic signature scheme. In particular, we separately analyze the security of the two variants picnic-FS and picnic-UR schemes, which differ in the way the underlying Σ-protocol based on ZKB++ is made non-interactive.

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 2.6 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 2.7. 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. $\square$

5.2 Security Analysis of Picnic

In the following sections we show that making the ZKB++ protocol non-interactive via the respective transformation yields a simulation-extractable non-interactive zero-knowledge proof system. As we will show in Section 5.3, this property directly yields strongly unforgeable signatures, i.e., the schemes satisfy strong existential unforgeability under chosen message attacks (sEUF-CMA) security.

5.2.1 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 provides simulation-extractability.*

5.2.2 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 6. 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 if we choose a random function H and a random output B, then run A_0^H followed by A_C to generate x, and then run $A_1^H(x, B)$, that for a random j, the j-th query A_1^H 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 $a = (y_1, y_2, y_3, C_1, C_2, C_3)$, $C_j = H_{com}(o_1)$, $C_{j+1} = H_{com}(o_2)$ 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 $\textsf{hash-input}(\pi^*) = \textsf{hash-input}(\pi')$ 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 $H(\textsf{hash-input}(\pi'))$ 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 $\textsf{hash-input}$ in a later proof - and $\textsf{hash-input}$ 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 $\textsf{hash-input}$ 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 $\textsf{hash-input}(\pi^*) = \textsf{hash-input}(\pi')$, 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 $\textsf{hash-input}(\pi^*) = \textsf{hash-input}(\pi')$, then $g^{(i)} = g'^{(i)}$ and $b^{(i)} = b'^{(i)}$ for all i, by definition of $\textsf{hash-input}$. 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 $\mathsf{MallSim}$ occurs with probability at most

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

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 $\pi = e, \{b^{(i)}, g^{(i)}, z^{(i)}\}_{i=1\dots t}$ such that for $\{a^{(i)}, (g_1^{(i)}, g_2^{(i)}, g_3^{(i)})\} =$ $\textsf{hash-input}(\pi)$ and $J_1 || \dots || J_t := H_0(\textsf{hash-input}(\pi))$*

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

(ii) $(a^{(i)}, J_i, (\textsf{open}_0(z^{(i)}), \textsf{open}_1(z^{(i)})))$ *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 $n_{\textsf{twovalid}}(c)$ be the number of i's for which there are 2 set-valid transcripts. Let $\textsf{E}_{\textsf{valid}}(c)$ be the set of challenge tuples which correspond to only set-valid conversations. (Note that $|\textsf{E}_{\textsf{valid}}(c)| = 2^{n_{\textsf{twovalid}}(c)}$.) 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 $\textsf{E}_{\textsf{valid}}(c)$. 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 $(\frac{2}{3})^t * 2^{n_{\textsf{twovalid}}(c)-t}$.

Let $\textsf{Cand}$ be the set of all candidates. Let $F : \textsf{Cand} \to \{0,1\}$ 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, $C_j = H_{com}(\nu_{1,j}) = H_{com}(\nu_{2,j-1})$ 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.

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

Here, the second equality follows from the definition of SigExtFail and Ev_{iii}, and the description of the extractor. The third equality follows from the fact that ¬MallSim means that the hash function on hash-input(π) has not been reprogrammed, and the fact that 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.

5.3 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 2.9). 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 pre-image 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 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.

6.1 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 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.

6.2 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).

6.3 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 EUF-CMA. Even if single-user EUF-CMA security generically implies multi-user EUF-CMA (MU-EUF-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 6.1 ([MS04], Theorem 6)**.** *Let* $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ *be an* $\mathsf{EUF\text{-}CMA}$ *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 $\mathsf{sEUF\text{-}CMA}$ 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 6.2. *picnic-FS and picnic-UR provide security in the multi-user setting in the sense of [MS04, Definition 6].*

7 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.[3] For example, at level L1, the Picnic LowMC instance requires 84KBytes 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're 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 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.

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

7.1 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 signatures are about 33KB. Reducing the number of rounds to 10 would make signatures 24KB.

7.2 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.[4] 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.

8 Advantages and Limitations

8.1 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

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

protecting Picnic private key operations on a commercial hardware security module.

8.2 TLS and X.509 Compatibility

The optimized implementation of Picnic has been integrated with the Open Quantum Safe (OQS) project.[5] Then, using a modified version of OpenSSL[6] 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 10.7.

8.3 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

[5]`https://github.com/open-quantum-safe/liboqs`
[6]`https://github.com/christianpaquin/openssl`

(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 3. 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 3: Mean round trip times in seconds for calls to an HSM creating Picnic signatures (average of 10 calls).

9 Additional Security Properties

9.1 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 implementation is constant time relative to the secret key.

Our fast implementation of the binary matrix multiplication step for LowMC uses precomputed data (the so-called "method of four Russians"). Look-ups to the table of precomputed data are done based on the LowMC state and round key. Therefore, the access pattern to memory is dependent on secret data (the access pattern of one of the three players is sensitive and should remain secret). This is mitigated 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++ can not be combined with information from other iterations. Since cache-timing side-channel attacks overwhelmingly 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 reference implementation 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.

9.2 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 hashed, then 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.

Alternative Parameter Selections. 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.

9.3 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.

10 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.

Optimzied-CT. An optimized implementation that uses processor-specific compiler instrinsics for vector instructions, e.g. SSE2 and AVX2 on Intel x86-64 and NEON on ARM v8, and uses a less optimized, but constant-time matrix-vector multiplication algorithm.

10.1 Description of the Benchmark Platforms

10.1.1 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 17.04

Compiler. GCC 6.3

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[7].

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

10.1.2 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 Leap 42.2

Compiler. GCC 6.2

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

10.1.3 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 16.04.3 LTS

Compiler. GCC 5.4

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

10.2 Description of the Benchmarking Methodology

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

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.

10.3 Benchmark Results: Sizes

In Table 4 we give the size of Picnic keys, and signatures. Note that these are the same for all implementations.

Parameter Set	Public key	Private key	Signature
`picnic-L1-FS`	32	16	34000
`picnic-L1-UR`	32	16	53929
`picnic-L3-FS`	48	24	76740
`picnic-L3-UR`	48	24	121813
`picnic-L5-FS`	64	32	132824
`picnic-L5-UR`	64	32	209474

Table 4: Key and signature sizes (in bytes) by security level. For the FS variants, the signature length varies based on the challenge, here we state the largest possible signature. On average the signature will be slightly less than this.

10.4 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.

In Tables 5, 6, 7 and 8 we present the benchmark results of all implementations on Platform A. On this platform we observe speed improvements of the **optimized** implementation compared to the **optimized-C** implementation of 20% up to 50% percent. **optimized-CT** is slower than the other optimized implementations, but has comparable performance to **optimized-C**'s L5 numbers. Additionally, Tables 9, 10 and 11 also give benchmark results with the GCC link time optimization (LTO) feature enabled. The **optimized-C** implementation shows improved performance figures with LTO enabled. The same holds for the **optimized** implementation except for `Picnic-L5-FS`, which is interestingly now closer to the performance figures of `Picnic-L5-UR`.

Next we give the results of the evaluation of our implementations on Platform B in Tables 12, 13, 15 and 14. 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

Parameters	Keygen	Sign	Verify
Picnic-L1-FS	0.05	36.50	23.91
(cycles)	163850	131390415	86062091
Picnic-L1-UR	0.04	44.12	29.48
(cycles)	146193	158826399	106128443
Picnic-L3-FS	0.10	122.85	81.03
(cycles)	364079	442257404	291723398
Picnic-L3-UR	0.10	144.96	97.40
(cycles)	369536	521842013	350635309
Picnic-L5-FS	0.20	298.11	196.85
(cycles)	722494	1073183185	7086526474
Picnic-L5-UR	0.21	329.86	221.46
(cycles)	740633	1187481996	797249015

Table 5: Benchmarks for the **reference** implementation, on benchmark Platform A.

Parameters	Keygen	Sign	Verify
Picnic-L1-FS	0.02	3.35	2.29
(cycles)	86938	12055646	8245436
Picnic-L1-UR	0.02	4.33	3.06
(cycles)	86631	15590725	11014466
Picnic-L3-FS	0.06	11.86	8.07
(cycles)	208123	42683333	29039354
Picnic-L3-UR	0.06	13.81	9.55
(cycles)	214387	49699928	34373293
Picnic-L5-FS	0.11	38.04	25.81
(cycles)	378326	136942395	92906288
Picnic-L5-UR	0.10	39.70	27.35
(cycles)	375633	142906149	98449354

Table 6: Benchmarks for the **optimized-C** implementation, on benchmark Platform A.

Parameters	Keygen	Sign	Verify
`Picnic-L1-FS`	0.01	2.10	1.45
(cycles)	41635	7544639	5228793
`Picnic-L1-UR`	0.01	2.78	2.01
(cycles)	43189	10024950	7223730
`Picnic-L3-FS`	0.06	7.41	5.23
(cycles)	199127	26669631	18813324
`Picnic-L3-UR`	0.06	9.47	6.76
(cycles)	205247	34092091	24331615
`Picnic-L5-FS`	0.06	14.88	10.49
(cycles)	204178	53572733	37749441
`Picnic-L5-UR`	0.06	19.75	14.33
(cycles)	209861	71097205	51603839

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

Parameters	Keygen	Sign	Verify
`Picnic-L1-FS`	0.01	5.55	3.73
(cycles)	28674	19978765	13411910
`Picnic-L1-UR`	0.01	6.29	4.29
(cycles)	28961	22628402	15429533
`Picnic-L3-FS`	0.03	17.39	11.77
(cycles)	105491	62601962	42387971
`Picnic-L3-UR`	0.03	19.33	13.26
(cycles)	104988	69605218	47723434
`Picnic-L5-FS`	0.03	36.95	25.01
(cycles)	95693	133023933	90049150
`Picnic-L5-UR`	0.03	39.50	27.08
(cycles)	96362	142205369	97473554

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

Parameters	Keygen	Sign	Verify
`Picnic-L1-FS`	0.00	2.66	1.82
(cycles)	13884	9571297	6536919
`Picnic-L1-UR`	0.00	3.38	2.37
(cycles)	13581	12156864	8537437
`Picnic-L3-FS`	0.01	9.69	6.56
(cycles)	38675	34886597	23612030
`Picnic-L3-UR`	0.01	11.66	8.00
(cycles)	40011	41966109	28804518
`Picnic-L5-FS`	0.03	32.51	22.09
(cycles)	97557	117026050	79538476
`Picnic-L5-UR`	0.03	34.07	23.44
(cycles)	99274	122644901	84389751

Table 9: Benchmarks for the **optimized-C** implementation with LTO enabled, on benchmark Platform A.

Parameters	Keygen	Sign	Verify
`Picnic-L1-FS`	0.00	1.95	1.36
(cycles)	9321	7011708	4910677
`Picnic-L1-UR`	0.00	2.64	1.91
(cycles)	9183	9488095	6888169
`Picnic-L3-FS`	0.01	6.61	4.63
(cycles)	39587	23806517	16677732
`Picnic-L3-UR`	0.01	8.84	6.29
(cycles)	43995	31834377	22661290
`Picnic-L5-FS`	0.02	14.71	10.64
(cycles)	59769	52965895	38296034
`Picnic-L5-UR`	0.02	18.67	13.60
(cycles)	64496	67220513	48966031

Table 10: Benchmarks for the **optimized** implementation with LTO enabled, on benchmark Platform A.

Parameters	Keygen	Sign	Verify
`Picnic-L1-FS`	0.01	5.41	3.70
(cycles)	29032	19464760	13311915
`Picnic-L1-UR`	0.01	6.12	4.24
(cycles)	29321	22037120	15280480
`Picnic-L3-FS`	0.03	17.07	11.61
(cycles)	104756	61455243	41785122
`Picnic-L3-UR`	0.03	19.01	13.08
(cycles)	104206	68430920	47075719
`Picnic-L5-FS`	0.03	36.47	24.70
(cycles)	94907	131297602	88911452
`Picnic-L5-UR`	0.03	39.21	26.90
(cycles)	96887	141142254	96853787

Table 11: Benchmarks for the **optimized-CT** implementation with LTO enabled, on benchmark Platform A.

of the **optimized** implementation are smaller. Furthermore, on Platform B the **optimized-CT** implementation performs significantly better than the other two optimized implementations. Taking the results of Platform A into account, these results suggest that **optimized-C** and **optimized** profits from larger caches and faster memory access.

10.5 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.

10.5.1 Reference Implementation Detailed Memory Usage

Memory usage was benchmarked using the Valgrind[8] 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`[9] was used to determine the peak memory usage of specific functions.

[8]http://valgrind.org/docs/manual/ms-manual.html
[9]`https://github.com/lnishan/massif-cherrypick`

Parameters	Keygen	Sign	Verify
`Picnic-L1-FS`	2.10	276.70	182.70
(cycles)	2518975	332042859	219239633
`Picnic-L1-UR`	2.09	338.02	228.30
(cycles)	2508982	405624756	273961444
`Picnic-L3-FS`	2.58	954.93	633.51
(cycles)	3095449	1145919907	760208950
`Picnic-L3-UR`	2.60	1135.01	766.11
(cycles)	3117483	1362017384	919332592
`Picnic-L5-FS`	3.43	2392.14	1596.34
(cycles)	4121029	2870568905	1915611672
`Picnic-L5-UR`	3.48	2648.95	1789.57
(cycles)	4175474	3178740376	2147486636

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

Parameters	Keygen	Sign	Verify
`Picnic-L1-FS`	2.04	48.59	32.84
(cycles)	2445968	58311725	39405284
`Picnic-L1-UR`	2.04	52.62	36.13
(cycles)	2450562	63140439	43350515
`Picnic-L3-FS`	2.13	184.77	124.70
(cycles)	2551716	221728344	149644259
`Picnic-L3-UR`	2.12	192.18	130.62
(cycles)	2546301	230616187	156739611
`Picnic-L5-FS`	2.24	407.46	273.45
(cycles)	2692059	488955553	328145361
`Picnic-L5-UR`	2.25	420.28	284.97
(cycles)	2705811	504336418	341969284

Table 13: Benchmarks for the **optimized-C** implementation, on benchmark Platform B.

Parameters	Keygen	Sign	Verify
`Picnic-L1-FS`	2.03	42.47	29.23
(cycles)	2436758	50965521	35075005
`Picnic-L1-UR`	2.02	46.48	32.49
(cycles)	2423651	55771020	38987895
`Picnic-L3-FS`	2.13	180.50	122.02
(cycles)	2552768	216601623	146422995
`Picnic-L3-UR`	2.13	190.11	128.84
(cycles)	2553342	228132612	154607251
`Picnic-L5-FS`	2.23	389.52	260.22
(cycles)	2679835	467418735	312262480
`Picnic-L5-UR`	2.25	403.05	271.93
(cycles)	2699871	483664428	326321733

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

Parameters	Keygen	Sign	Verify
`Picnic-L1-FS`	1.88	33.37	23.73
(cycles)	2258863	40039857	28479368
`Picnic-L1-UR`	1.95	36.57	26.46
(cycles)	2334104	43889479	31749179
`Picnic-L3-FS`	2.10	138.53	97.05
(cycles)	2525485	166237027	116465158
`Picnic-L3-UR`	2.13	148.37	104.17
(cycles)	2553532	178049673	125000932
`Picnic-L5-FS`	2.22	303.43	209.78
(cycles)	2664574	364116590	251733508
`Picnic-L5-UR`	2.28	318.10	632.07
(cycles)	2736375	381725885	758481668

Table 15: Benchmarks for the **optimized-CT** implementation, on benchmark Platform B.

Parameters	Keygen	Sign	Verify
`Picnic-L1-FS`	0.07	51.92	33.92
(cycles)	212881	166159669	108529436
`Picnic-L1-UR`	0.06	62.51	41.71
(cycles)	193569	200039945	133478227
`Picnic-L3-FS`	0.14	172.82	113.70
(cycles)	462757	553026245	363831127
`Picnic-L3-UR`	0.15	203.82	136.54
(cycles)	471108	652218966	436915884
`Picnic-L5-FS`	0.28	415.36	275.93
(cycles)	901355	1329138266	882984854
`Picnic-L5-UR`	0.29	459.44	308.87
(cycles)	922864	1470196303	988380511

Table 16: Benchmarks for the **reference** implementation, on benchmark Platform C.

Parameters	Keygen	Sign	Verify
`Picnic-L1-FS`	0.03	4.50	3.12
(cycles)	100469	14403397	9998428
`Picnic-L1-UR`	0.03	5.84	4.19
(cycles)	100825	18678042	13413739
`Picnic-L3-FS`	0.07	14.68	10.14
(cycles)	239214	46980834	32456544
`Picnic-L3-UR`	0.07	19.87	14.05
(cycles)	234956	63576699	44966117
`Picnic-L5-FS`	0.14	46.41	31.76
(cycles)	435994	148516664	101624906
`Picnic-L5-UR`	0.14	51.44	36.37
(cycles)	440258	164615202	116384582

Table 17: Benchmarks for the **optimized-C** implementation, on benchmark Platform C.

Parameters	Keygen	Sign	Verify
`Picnic-L1-FS`	0.02	3.50	2.42
(cycles)	48587	11193948	7745406
`Picnic-L1-UR`	0.02	4.87	3.48
(cycles)	48760	15594829	11132090
`Picnic-L3-FS`	0.07	13.46	9.39
(cycles)	226283	43076300	30061948
`Picnic-L3-UR`	0.07	17.36	12.42
(cycles)	230344	55549079	39737866
`Picnic-L5-FS`	0.10	32.21	22.22
(cycles)	311304	103056039	71098134
`Picnic-L5-UR`	0.10	37.77	26.82
(cycles)	314544	120862142	85830001

Table 18: Benchmarks for the **optimized** implementation, on benchmark Platform C.

Massif was invoked with the command:

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

When creating a signature, peak memory usage ranged from about 227K to 1091K bytes, as shown in Table 19, while verification ranged from about 183K to 959K 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,

Parameter set	Sign	Verify
`Picnic-L1-FS`	227,103	183,154
`Picnic-L1-UR`	315,579	254,772
`Picnic-L3-FS`	480,835	379,860
`Picnic-L3-UR`	645,335	565,034
`Picnic-L5-FS`	802,197	652,539
`Picnic-L5-UR`	1,091,277	959,553

Table 19: 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.

Parameter set	Precomputed Data	Sign	Verify
`Picnic-L1-FS`	1,835,744	133,598	79,724
`Picnic-L1-UR`	1,835,744	196,889	126,590
`Picnic-L3-FS`	7,078,944	230,300	108,570
`Picnic-L3-UR`	7,078,944	373,415	214,508
`Picnic-L5-FS`	11,797,792	398,580	189,216
`Picnic-L5-UR`	11,797,792	642,546	370,548

Table 20: 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.

since we're only interested in peak usage.

10.5.2 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 20. The row containing precomputed data only applies to the **optimized-C** and **optimized** implementations. A more detail discussion of the precomputed constants and data follows below.

10.6 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**, **optimized**, and **optimized-CT** implementations use an alternative but equivalent representation of LowMC. They implement a reduced linear layer [PPRR17], 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 Table 22 in the Reduced Key Matrices column.

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 21: 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	41,600	960	14,400	56,960
L3	185,280	1,920	30,784	217,984
L5	315,512	2,432	49,216	364,160

Table 22: 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.

Size of Precomputed Data. In order to reduce the time required to implement the binary matrix-vector multiplication step in the linear layer of LowMC, the **optimized-CT** and **optimized** implementations can optionally pre-compute some data based on the constants. The precomputation and multiplication are implemented following the "method of four Russians" [ADKF70, Bar06]. The size of the data is given in Table 23. Once the implementation has this data, it no longer needs the constant data from Table 22 (except the round constants).

Params	Linear Matrices	Key Matrices
L1	1,311,360	458,816
L3	5,899,200	983,104
L5	9,962,688	1,572,928

Table 23: Size of precomputed data (in bytes) required by LowMC, with the parameters used in Picnic at security levels L1, L3 and L5.

10.7 TLS Performance

As described in Section 8.1 we used OQS, OpenSSL and Apache to benchmark Picnic and other post-quantum cryptography in the context of HTTPS. We set up a web server in Microsoft Azure (a standard D2s v3 instance[10]), 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 signature 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 24. The second was a Dell Poweredge R710 server on the Microsoft campus, labeled "fast network" in Table 24. 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[11] (also compiled with

[10]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).

[11]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

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 24 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 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.

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 24: 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.

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.

[ADKF70] V. Arlazarov, E. Dinic, M. Kronrod, and I. Faradzev. On economical construction of the transitive closure of a directed graph. *Soviet Math Dokl.*, 1970.

[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.

[Bar06] Gregory V. Bard. Accelerating cryptanalysis with the method of four russians. *IACR Cryptology ePrint Archive*, 2006:251, 2006.

[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.

[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, Claudion Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. Post-quantum zero-knowledge and signatures from symmetric-key primitives. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, Dallas, USA, October 30 - November 3, 2017*, 2017. to appear.

[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.

[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.

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

[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.

[PPRR17] Léo Perrin, Angela Promitzer, Sebastian Ramacher, and Christian Rechberger. Improvements to the linear layer of lowmc: A faster picnic. Cryptology ePrint Archive, Report 2017/1148, 2017. http://eprint.iacr.org/2017/1148.

[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.

[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.

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

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 $s$
```

$p \leftarrow \texttt{Sim}(x)$: 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 $H(a^{(1)}, \dots, a^{(t)}) := e$.
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 $p \leftarrow [e, (b^{(1)}, z^{(1)}), (b^{(2)}, z^{(2)}), \cdots, (b^{(t)}, z^{(t)})]$.

Scheme 6: The zero knowledge simulator