Efficient Post-Quantum Zero-Knowledge and Signatures (Draft)

Steven Goldfeder
Princeton
`stevenag@cs.princeton.edu`

Melissa Chase
Microsoft Research
`melissac@microsoft.com`

Greg Zaverucha
Microsoft Research
`gregz@microsoft.com`

November 22, 2017

Abstract

In this paper, we present a new post-quantum digital signature algorithm that derives its security entirely from assumptions about symmetric-key primitives, which are very well studied and believed to be quantum-secure (with increased parameter sizes). We present our new scheme with a complete post-quantum security analysis, and benchmark results from a prototype implementation.

Our construction is an efficient instantiation of the following design: the public key is $y = f(x)$ for preimage-resistant function f, and x is the private key. A signature is a non-interactive zero-knowledge proof of x, that incorporates a message to be signed. Our security analysis uses recent results of Unruh (EUROCRYPT'12,'15,'16) that show how to securely convert an interactive sigma protocol to a non-interactive one in the quantum random oracle model (QROM). The Unruh construction is generic, and does not immediately yield compact proofs. However, when we specialize the construction to our application, we can reduce the size overhead of the QROM-secure construction to 1.6x, when compared to the Fiat-Shamir transform, which does not have a rigorous post-quantum security analysis. Our implementation results compare both instantiations, with multiple choices of f for comparison. Our signature scheme proposal uses the block cipher LowMC for f, as it gives the shortest signatures.

In addition to reducing the size of signatures with Unruh's construction, we also improve the size of proofs in the underlying sigma protocol (of Giacomelli et al., USENIX'16) by a factor of two. This is of independent interest as it yields more compact proofs for arbitrary choices of f in the classical case as well. Further, this reduction in size comes at no additional computational cost.

1 Introduction

At a high level, our signature scheme uses a non-interactive zero-knowledge (NIZK) proof-of-knowledge as a signature scheme. The zero-knowledge proof is made non-interactive with a generic transform similar to the well-known Fiat-Shamir (FS) transform [9]. As the basis of our scheme, we use the recently proposed ZKB proof system [10], which allows for proving statements of the form, "I know x such that $f(x) = y$", where f can be an arbitrary function such as SHA-256. Our key generation algorithm will make use of a function f that is assumed to be preimage resistant against a quantum adversary. That is, for a uniformly distributed x, given $y = f(x)$, it is difficult to find an x' such that $f(x') = y$. The signature will then be a non-interactive a proof-of-knowledge of x, that incorporates the message to be signed (as in the FS transform). This way of designing a signature scheme is similar to one described by Bellare and Goldwasser [2], but differs in the way the message is bound to the proof. In our scheme the message may be hashed when computing the challenge in the proof.

Using a NIZK proof of a preimage for an arbitrary function f is appealing, since public and private keys are short, key generation is simple, and the security of f is well understood (or commonly assumed). The two main challenges to make this practical are (1) efficiently realizing the NIZK proof and (2) post-quantum security analysis NIZK.

ZKB builds on the MPC-in-the-head paradigm of Ishai *et al.* [12], that we describe informally here. The multiparty computation protocol (MPC) will implement f, and the input is the witness x. For example, the MPC could compute $y = \text{SHA-256}(x)$ where parties each have a share of x and y is public. The idea is to have the prover simulate a multiparty computation protocol "in his 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 computation was done correctly from the perspective of the corrupted players, and if so, he has some assurance that the output is correct. Iterating this for many rounds then gives the verifier high assurance.

In terms of practicality, each round is very computationally efficient, as it requires no number theoretic computations. The results of [10] have shown that computational costs are practical, even with an unoptimized implementation. The main challenge is signature size. Each round of the ZKB protocol outputs a transcript of the MPC protocol, and a large number of rounds are required for soundness. For example, for 80-bits of security against a classical attacker, ZKB proof of a SHA-256 preimage requires 137 rounds and is 835KB.

We made many improvements to ZKB, that taken together reduces the proof size by a factor of two when compared to [10]. We reduce the amount of randomness needed for each round of the protocol. In the original protocol, there were three independent sources of randomness: (1) the randomness to compute a secret sharing of the witness for the relation being proved, (2) a random seed for a PRG, and (3) randomness for the commitments. we show how it is sufficient (without making any additional hardness assumptions) to use a single source of randomness; the seed for the PRG.

Then we modify the verification algorithm by having the verifier re-compute parts of the MPC transcript, instead of including it in the proof. Since the verifier must recompute these values to check that they are correct anyway, this does not increase computational cost. The comparison happens indirectly, when the hash of the recomputed transcripts are compared to the hash provided in the proof.

Moving to quantum safe parameters requires more that 137 rounds however, 438 by our analysis, so it is clear that proofs of SHA-256 preimages are too large. There is much flexibility in the choice of f, and we chose a block cipher optimized for MPC called LowMC [1]. LowMC is designed to minimize the number of non-linear operations (multiplication gates), to reduce the amount of communication required by MPC protocols. Choosing f as an instance of LowMC reduces signature sizes dramatically. Our most efficient quantum safe parameter set at the 128-bit security level has 150KB signatures.

The second main challenge is the security analysis of the NIZK. It is not difficult to show that the *interactive* ZKB protocol is secure against a quantum adversary (with suitable parameters to account for generic advantage based on Grover's algorithm). The difficulty is when transforming ZKB to a non-interactive proof. The Fiat-Shamir transform is simple and efficient, however, it lacks a proof in the quantum random oracle model (QROM). There are no known quantum attacks on the transform either, and many papers assume post-quantum security of FS signatures, if the primitives of the sigma protocol are quantum safe. See [4, 7] for more details.

In a series of papers [14, 15, 16], Unruh studies this problem, and presents a generic transform for turning a sigma protocol into a signature scheme with a security proof in the QROM. However, the ZKB protocol doesn't completely match the properties that Unruh's analysis requires. We show how to modify Unruh's security proof to apply to ZKB.

We also give concrete parameters and an implementation for both the Unruh and FS transforms and ZKB, and measure the overhead of Unruh's transform over regular Fiat-Shamir. The Unruh construction is generic, and does not immediately yield compact proofs. However, when we specialize the construction to our application, we find the overhead was surprisingly low, as a generic application of Unruh's transform incurs a 4x increase in cost when compared to FS. For our LowMC parameter set targeting 128-bits of post-quantum security, the signature size overhead is about 1.6x (150KB for Unruh vs. 91KB for FS). In our current implementation the CPU overhead is roughly a factor of 1.2x. This shows that choosing a transform with a strong post-quantum security analysis can be practical.

2 Preliminaries and Definitions

Our definition of sigma protocols follows [6]. For a more formal definition, see [7].

Sigma protocol A *sigma protocol* (equivalently denoted Σ-protocol) is a three flow protocol between a prover P and verifier V where transcripts have the form (r, c, s) where r and s are computed by P and c is a challenge chosen by V. Let f be a relation such that $f(x) = y$, where y is common input and x is a witness known only to P. V accepts if $\phi(y, r, c, s) = 1$ for an efficiently computable predicate ϕ. Given two accepting transcripts

(r, c, s) and (r, c', s') where $c \neq c'$, there is an efficient algorithm to extract a witness x' such that $f(x') = y$. There also exists an efficient simulator, given y and a randomly chosen c, outputs a transcript (r, c, s) for y that is indistinguishable from a real run of the protocol for x, y.

n-special soundness The sigma protocol described in the previous paragraph is 2-special sound, since two transcripts with different challenges are sufficient to extract a witness. A sigma protocol has *n-special soundness* if n transcripts $(r, c_1, s_1), \ldots (r, c_n, s_n)$ with distinct c_i guarantee that a witness may be efficiently extracted.

The following definition of a pseudorandom function (PRF) is specialized to this work; it uses a common parameter for the key, input and output length. The notation D^{F_n} means that D is given oracle access to F_n.

Definition 1 (Pseudorandom Function) *Let $F : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$ be an efficiently computable, length-preserving keyed function. We say that F is a* **pseudorandom function** *(PRF), if for all probabilistic polynomial time distinguishers D,*

$$|\Pr[D^{F_k}(1^n) = 1] - \Pr[D^{f_n}(1^n) = 1]|$$

is negligible where $k \leftarrow \{0,1\}^n$ is chosen uniformly at random and f_n is chosen uniformly at random from the set of functions mapping n-bit strings to n-bit strings.

We now define a weaker notion of a pseudorandom function in which we put an upper bound on the number of queries that the distinguisher can make to its oracle.

Definition 2 (q-Pseudorandom Function) *Let F_k and f_n be as defined in Definition 1, and let q be a positive integer constant. We say that F is a* **q-pseudorandom function** *(q-PRF) if for all probabilistic polynomial time distinguishers D that make at most q queries to their oracle,*

$$|\Pr[D^{F_k}(1^n) = 1] - \Pr[D^{f_n}(1^n) = 1]|$$

is negligible.

Clearly, a pseudorandom function is also a q-pseudorandom function for any constant q. When considering concrete security of PRFs against quantum attacks, we assume that an n-bit function provides $n/2$ bits of security. In particular we make this assumption about the LowMC block cipher with the parameters chose in Section 5.

This paper requires a simple type of PRG a stateless function that expands a seed to a random string.

Definition 3 (Pseudorandom Generator) *An (n, ℓ)* pseudorandom generator *(PRG) is a function $P : \{0,1\}^n \to \{0,1\}^\ell$ that expands an n-bit seed to an ℓ-bit random string. Informally, the PRG is said to be* secure *if for randomly chosen seeds, the output is indistinguishable from the uniform distribution on $\{0,1\}^\ell$.*

Concretely, we assume that AES-256 in counter mode provides 128 bits of PRG security, when used to expand 256-bit seeds to outputs less than 1KB in length.

Definition 4 (Preimage Resistance) *Let $H : \{0,1\}^* \to \{0,1\}^n$ be a cryptographic hash function. H is said to be* preimage-resistant *if, given $y \in \{0,1\}^n$, finding an $x \in \{0,1\}^*$ such that $H(x) = y$ is is computationally intractable, i.e., costs at least $O(2^{n/2})$ operations.*

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.

Definition 5 (Collision Resistance) *Let $H : \{0,1\}^* \to \{0,1\}^n$ be a cryptographic hash function. H is said to be* collision-resistant *if finding a pair (x, x') such that $x \neq x'$ is computationally intractable, i.e., costs at least $2^{n/2}$ operations.*

When considering quantum algorithms, in theory it may be possible to find collisions using a generic algorithm of Brassard et al. [5] with cost $O(2^{n/3})$. A detailed analysis of the costs of the algorithm in [5] by Bernstein [3] 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 [3]). We make this assumption as well, and in particular, that SHA-256 provides 128 bits of PQ security.

A computationally secure commitment scheme allows a sender to commit to a message such that given the transcript of the commitment phase the receiver, a computationally bound receiver cannot guess the committed message with probability non-negligibly better than at random. The sender is computationally bound to the committed message, i.e. cannot open to another message with greater than negligible probability.

Definition 6 (Commitment Scheme) *Formally a (non-interactive) commitment scheme consists of three algorithms* KG, Com, Ver *with the following properties:*

- KG *is the key generation algorithm, on input the security parameter it outputs a public key* pk.
- Com *is the commitment algorithm. On input of a message M it outputs $[C(M), D(M)] = \mathsf{Com}(\mathsf{pk}, M, R)$ where R are the coin tosses. $C(M)$ is the commitment string, while $D(M)$ is the decommitment string which is kept secret until opening time.*
- Ver *is the verification algorithm. On input C, D, it either outputs a message M or $\perp$.*

We note that if the sender refuses to open a commitment we can set $D = \perp$ and $\mathsf{Ver}(\mathsf{pk}, C, \perp) = \perp$. Computationally secure commitments must satisfy the following properties

Correctness *If* $[C(M), D(M)] = \mathsf{Com}(M, R)$ *then* $\mathsf{Ver}(\mathsf{pk}, C(M), D(M)) = M$.

Secure hiding *For every message pair* M, M' *the probability ensembles* $\{C(M)\}_{n\in\mathbb{N}}$ *and* $\{C(M')\}_{n\in\mathbb{N}}$ *are computationally indistinguishable for security parameter* n.

Secure Binding *We say that an adversary* $\mathcal{A}$ *wins if it outputs* C, D, D' *such that* $\mathsf{Ver}(C, D) = M$, $\mathsf{Ver}(C, D') = M'$ *and* $M \neq M'$. *We require that for all efficient algorithms* $\mathcal{A}$, *the probability that* $\mathcal{A}$ *wins is negligible in the security parameter.*

To simplify our notation, we will often not explicitly write the public key pk when we make use of commitments. Our implementation uses hash-based commitments, which requires modeling the hash function as a random oracle in our security analysis. Note also that randomizing the Com function may not be necessary if M has high entropy.

3 ZKB++

ZKB is a new proof system for zero-knowledge proofs on arbitrary circuits described in [10]. We describe the protocol here, but also present ZKB++, a much improved version of ZKB which enables proofs that are less than half the size of ZKB[1].

3.1 ZKB

We now present the details of of the ZKB protocol. While ZKB is presented with various possible parameter options, we present only the final version with the best parameters. Moreover, while ZKB presents both interactive and non-interactive protocol versions, we present only the non-interactive version since our main goal is building a signature scheme for which we need the non-interactive version. In ZKB, the non-interactive version is not fully specified, so we refer to the accompanying implementation to fill in details where necessary.

Overview

ZKB builds on the MPC-in-the-head paradigm of Ishai *et al.* [12], 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 his 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 computation was done correctly from the perspective of the corrupted players, and if so, he has some assurance that the output is correct. Iterating this for many rounds then gives the verifier high assurance.

[1]In [10] the protocol is named "ZKBoo", which abbreviate as ZKB here.

ZKB implements the idea of [12] as follows. In order to prove knowledge of a witness for a relation $r := \{(x, y), \phi(x) = y\}$, we begin with a circuit that computes ϕ, and then the circuit is first decomposed into three parts. Intuitively, the decomposition is a three player MPC – it contains a `share` function that splits the input into three shares, three functions $\texttt{output}_{i\in\{1,2,3\}}$ that take as input all of the input shares and some randomness and produce an output share for each of the parties, and a function `reconstruct` that takes as input the three output shares and reconstructs the circuit's final output. The decomposition is *correct*, and has 2-*privacy* which intuitively means that revealing the views of any two players does not leak information about the witness x.

With this decomposition, ZKB then uses the MPC-in-the-head technique to prove knowledge of x. The prover simulates n independent runs of the three player MPC protocol and commits to the views – 3 views per run. The 3-party MPC protocol has a *2-privacy* property, which informally means that if for each run, two players are corrupted and the adversary learns their view, x still remains private. Then, using the Fiat-Shamir heuristic, the prover sends the commitments and output shares from each view to the random oracle computes a challenge – the challenge tells the prover which two of the three views to open for each of the n runs. Because of the two-privacy property, opening two views for each run does not leak information about the witness. The number of runs, n, is chosen to achieve negligible soundness error – i.e. intuitively it would be infeasible for the prover cheat without getting caught in at least one of the runs. The verifier checks that (1) the output of the each of the three views reconstructs to y, (2) each of the two open views were computed correctly, and (3) the challenge was computed correctly.

We now give a detailed description of the non-interactive ZKB protocol. Throughout this paper, when we do arithmetic on the indices of the players, there's an implicit mod3 that omit to simplify the notation.

Definition 7 ((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 invertible 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` *computed the MPC 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, y_i.

We define the following experiment $\mathsf{EXP}^{(\phi,\mathsf{x})}_{\mathsf{decomp}}$ *which runs the decomposition over a circuit* ϕ *on input* x*:*

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

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: $y_i \leftarrow \texttt{output}(\mathsf{View}_i)$

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

(Correctness) *For all circuits* ϕ*, for all inputs* x *and for the* y_i*'s produced by , for all circuits* ϕ*, for all inputs* x*,*

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

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

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

is negligible.

3.1.1 The linear decomposition of a circuit

ZKB uses an explicit (2,3)-decomposition, which we recall here. 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, binary addition and binary 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 $x_1 + x_2 + x_3 = x$.

- $\mathsf{view}_i^{(j+1)} \leftarrow \mathtt{update}_i^{(j)}(\mathsf{view}_i^{(j)}, \mathsf{view}_{i+1}^{(j)}, k_i, k_{i+1})$ computes P_i's view of the output wire of gate g_j and appends it to the view. Notice that it takes as input the views and random tapes of both party P_i as well as party P_{i+1}. We use w_k to refer to the k-th wire, and we use $w_k^{(i)}$ to refer to the value of w_k in party P_i's view. The update operation depends on the type of gate g_j, and are defined as follows:

  – case 1: ***addition by constant** ($w_b = w_a + k$)**.*

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

  – case 2: ***multiplication by constant** ($w_b = w_a \times k$)**.*

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

  – case 3: ***binary addition** ($w_c = w_a + w_b$)**.*

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

  – case 4: ***binary multiplication** ($w_c = w_a \times w_b$)**.*

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

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

  Note that with the exception of case 1, 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}.

- $y_i \leftarrow \mathtt{output}_i(\mathsf{view}_i^{(n)})$ selects the ℓ output wires of the circuit as stored in the view $\mathsf{view}_i^{(n)}$.

- $y \leftarrow \mathtt{reconstruct}(y_1, y_2, y_3) = y_1 + y_2 + y_3$

While we don't give the details here, [10] shows that this decomposition meets the correctness and 2-privacy requirements of Definition 7. We will see this in more detail in Section 4.1 when we give our post-quantum security analysis of ZKB++.

3.1.2 Choice of R

The functions that we work with in this paper are mostly specified on the bit-level, and therefore a natural choice for the ring is $\mathbb{Z}_2$. For a boolean circuit then, bitwise AND operations (i.e. binary multiplication) require communication among parties, but XOR and INV gates can be computed locally.

3.1.3 ZKB Complete Protocol

Given a $(2,3)$-decomposition $\mathcal{D}$ for a function ϕ, the ZKB protocol is a zero knowledge proof system for relations of the form $R := \{(y,x) : y = \phi(x)\}$. We recall the details of ZKB in Figure 3.1.3.

The ZKB non-interactive proof system

For public ϕ and $y \in L_\phi$, the prover has $x : y = \phi(x)$. $\texttt{Com}(\cdot)$ is a secure commitment scheme. The prover and verifier have access to a hash function $H(\cdot)$. The integer t is the number of parallel iterations.

$p \leftarrow \texttt{Prove}(x)$:

1. For each iteration $r_i, i \in [1,t]$:
   - (a) Sample random tapes $k_1^{(i)}, k_2^{(i)}, k_3^{(i)}$
   - (b) Simulate the MPC protocol "in the head" to get an output view $\mathsf{View}_j^{(i)}$ and output share $y_j^{(i)}$ for each player . In particular, for each player P_j:
     - i. $(x_1^{(i)}, x_2^{(i)}, x_3^{(i)}) = \texttt{Share}(x, k_1^{(i)}, k_2^{(i)}, k_3^{(i)})$
     - ii. $\mathsf{View}_j^{(i)} = \texttt{Update}(\texttt{Update}(\cdots \texttt{Update}(x_j^{(i)}, x_{j+1}^{(i)}, k_j^{(i)}, k_{j+1}^{(i)}) \ldots) \ldots) \ldots)$
     - iii. $y_j^{(i)} = \texttt{Output}(\mathsf{View}_j^{(i)})$
     - iv. Commit: $[C_j^{(i)}, D_j^{(i)}] = \texttt{Com}(k_j^{(i)}, \mathsf{View}_j^{(i)})$
   - (c) Denote $a^{(i)} = (y_1^{(i)}, y_2^{(i)}, y_3^{(i)}, C_1^{(i)}, C_2^{(i)}, C_3^{(i)})$
2. Compute the challenge: $e = H(a^{(1)}, a^{(2)}, \cdots, a^{(t)})$. Interpret the challenge such that for $i \in [1,t], e^{(i)} \in \{1,2,3\}$
3. For each iteration $r_i, i \in [1,t]$, Denote $z^{(i)} = (D_e^{(i)}, D_{e+1}^{(i)})$
4. Output $p = [(a^{(1)}, z^{(1)}), (a^{(2)}, z^{(2)}), \cdots, (a^{(t)}, z^{(t)})]$

$b \leftarrow \texttt{Verify}(y, p)$:

1. Compute the challenge: $e' = H(a_1, a_2, \cdots, a_t)$. Interpret the challenge such that for $i \in [1, t], e'^{(i)} \in \{1, 2, 3\}$

2. For each iteration $r_i, i \in [1, t]$:

   (a) If $\exists j \in \{e'^{(i)}, e'^{(i)} + 1\}$ s.t. $\mathtt{Ver}(C_j^{(i)}, D_j^{(i)}) = \perp$, output Reject

   (b) Else, $\forall j \in \{e'^{(i)}, e'^{(i)} + 1\}$,

   $$\{k_j^{(i)}, \mathsf{View}_j^{(i)}\} = \mathtt{Ver}(C_j^{(i)}, D_j^{(i)})$$

   (c) If $\mathtt{Reconstruct}(y_1^{(i)}, y_2^{(i)}, y_3^{(i)}) \neq y$, output Reject

   (d) If $\exists j \in \{e'^{(i)}, e'^{(i)} + 1\}$ s.t. $y_j^{(i)} \neq \mathtt{Output}(\mathsf{View}_j^{(i)})$, output Reject

   (e) For each wire value $w_j^{(e)} \in \mathsf{View}_e$, if

   $$w_j^{(e)} \neq \mathtt{update}(\mathsf{view}_e^{(j-1)}, \mathsf{view}_{e+1}^{(j-1)}, k_e, k_{e+1})$$

   output Reject

3. output Accept

3.1.4 Serializing the views: ZKB approach

In the decomposition as described in Definition 7, the view is updated with the output wire value for each gate. While conceptually a player's view includes the values that they computed locally, when the view is serialized, it is sufficient to include only the wire values of the gates that require non-local computations (i.e., the binary multiplication gates). The verifier can recompute the parts of the view due to local computations, and they don't need to be serialized. Giving the verifier locally computed values does not even save any computation as the verifier will still need to recompute the values in order to check them.

In the ZKB paper, the serialized view included the following:

- input share
- output wire values for binary multiplication gates
- output share

The size of a view depends on the circuit as well as the ring that it is computed over. Let $\phi : (\mathbb{Z}_{2^\ell})^m \to (\mathbb{Z}_{2^\ell})^n$ be the circuit being computed over $\mathbb{Z}_{2^\ell}$ such that there are m input wires, n output wires, and each wire can be expressed with ℓ bits. Moreover, assume that the circuit has b binary-multiplication gates. The size of a view in bits is thus given by:

$$|\mathsf{View}_i| = \ell(m+n+b)$$

3.1.5 ZKB proof size

Continuing with the above notation, we can now calculate the size of the ZKB proofs. Indeed the sizes that we give correspond to the sizes that were output by the ZKB reference code and reported in their paper. Recall from Figure 3.1.3 that for each iteration, a proof includes

1. $a = (y_1, y_2, y_3, C_1, C_2, C_3)$, where y_i is the output share of party P_i and c_i is the commitment to View_i
2. $z = (D_e, D_{e+1})$, where $e \in \{1,2,3\}$ is the challenge.

Assume that the random tapes are of size κ, and the commitments are of size c bits. In the hash based commitment scheme used by ZKB, the openings D of the commitments contain the value being committed to as well as the randomness used for the commitments. Let s denote the size of the randomness in bits used for each commitment. The size of the output share y_i is the same as the output size of the circuit, $(\ell \times n)$. Assume that there are r iterations.The total proof size is thus given by

$$\begin{aligned}
|p| &= r \times [|a| + |z|] \\
&= r \times [3 \times (|y_i| + |c_i|) + 2 \times (|\mathsf{View}_i| + |k_i| + s)] \\
&= r \times [3 \times (\ell n + c) + 2 \times (\ell \times (m+n+b+s) + \kappa)] \\
&= r \times [3c + 2\kappa + 2s + \ell \times (5n + 2m + 2b)]
\end{aligned}$$

3.1.6 Commitment scheme used by ZKB

3.2 ZKB++

We now present ZKB++, a modification of ZKB. While the protocol follows along the same lines, we have several observations that allow us to reduce the proof size to less than half of the size in the original ZKB protocol. Moreover, our benchmarks show that this size reduction comes at no extra computational cost.

3.2.1 Modification 1: A modification of the decomposition

The ZKB paper omits the details of how the `Share` function samples from the three input tapes that it is given. Moreover, when we examined the accompanying source code, we found that the random tapes are not actually used in the `Share` function, but are instead sampled using independent randomness.

We therefore specify sampling as follows:
$(x_1, x_2, x_3) \leftarrow \texttt{Share}(x, k_1, k_2, k_3) =:$

$$\begin{aligned} x_1 &= R_1(0) \\ x_2 &= R_2(0) \\ x_3 &= x - x_1 - x_2 \end{aligned}$$

As before, R_i is a pseudorandom generator seeded with k_i.

We note that sampling in this manner preserves the 2-privacy of the decomposition. In particular, given only two of $\{(k_1, x_1), (k_2, x_2), (k_3, x_3)\}$, x remains uniformly distributed over the choice of the third unopened (k_i, x_i).

We specify the `Share` function in this manner as it will lead to more compact zero-knowledge proofs. Moving now to the ZKB 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. The expected reduction in proof size resulting from using the ZKB++ sampling technique instead of the technique used in ZKB is

$$r \times \frac{4}{3} \times |x| \text{ bits}$$

Given a $(2, 3)$-decomposition $\mathcal{D}$ for a function ϕ, the ZKB++ protocol is a zero knowledge proof system for relations of the form $R := \{(y, x) : y = \phi(x)\}$. We recall the details of ZKB++ in Figure 3.2.1.

The ZKB++ non-interactive proof system

For public ϕ and $y \in L_\phi$, the prover has $x : y = \phi(x)$. The prover and verifier have access to a Random Oracle $H(\cdot).t$ is the number of parallel iterations.

$p \leftarrow \texttt{Prove}(x)$:

1. For each iteration $r_i, i \in [1, t]$:
    (a) Sample random tapes $k_1^{(i)}, k_2^{(i)}, k_3^{(i)}$
    (b) Simulate the MPC protocol "in the head" to get an output view $\mathsf{View}_j^{(i)}$ and output share $y_j^{(i)}$ for each player. In particular, for each player P_j:

i.

$$\begin{aligned}(x_1^{(i)}, x_2^{(i)}, x_3^{(i)}) &\leftarrow \texttt{Share}(x, k_1^{(i)}, k_2^{(i)}, k_3^{(i)}) \\ &= (G(k_1^{(i)}), G(k_2^{(i)}), x \oplus G(k_1^{(i)}) \oplus G(k_2^{(i)}))\end{aligned}$$

ii. $\mathsf{View}_j^{(i)} = \texttt{Update}(\texttt{Update}(\cdots\texttt{Update}(x_j^{(i)}, x_{j+1}^{(i)}, k_j^{(i)}, k_{j+1}^{(i)})\ldots)\ldots)\ldots)$

iii. $y_j^{(i)} = \texttt{Output}(\mathsf{View}_j^{(i)})$

iv. Commit: $[C_j^{(i)}, D_j^{(i)}] = [H(k_j^{(i)}, \mathsf{View}_j^{(i)}), k_j^{(i)} || \mathsf{View}_j^{(i)}]$

(c) Denote $a^{(i)} = (y_1^{(i)}, y_2^{(i)}, y_3^{(i)}, C_1^{(i)}, C_2^{(i)}, C_3^{(i)})$

2. Compute the challenge: $e = H(a^{(1)}, a^{(2)}, \cdots, a^{(t)})$. Interpret the challenge such that for $i \in [1, t], e^{(i)} \in \{1, 2, 3\}$

3. For each iteration $r_i, i \in [1, t]$,

   (a) Denote $b^{(i)} = (y_{e^{(i)}+2}^{(i)}, C_{e^{(i)}+2}^{(i)})$

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

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

$b \leftarrow \texttt{Verify}(y, p)$:

1. For each iteration $r_i, i \in [1, t]$:

   (a) Run the MPC protocol to reconstruct the views, input and output shares that were not explicitly given as part of the proof p. In particular:

   i. $x_{e^{(i)}}^{(i)} = \begin{cases}\text{if } e^{(i)} = 1, & G(k_1^{(i)}) \\ \text{if } e^{(i)} = 2, & G(k_2^{(i)}) \\ \text{if } e^{(i)} = 3, & x_3^{(i)} \text{ which was explicitly given as part of } z^{(i)}\end{cases}$

   ii. $x_{e^{(i)}+1}^{(i)} = \begin{cases}\text{if } e^{(i)} = 1, & G(k_2^{(i)}) \\ \text{if } e^{(i)} = 2, & x_3^{(i)} \text{ which was explicitly given as part of } z^{(i)} \\ \text{if } e^{(i)} = 3, & G(k_1^{(i)})\end{cases}$

   iii. $\mathsf{View}_e^{(i)} = \texttt{Update}(\texttt{Update}(\cdots\texttt{Update}(x_e^{(i)}, x_{e+1}^{(i)}, k_e^{(i)}, k_{e+1}^{(i)})\ldots)\ldots)\ldots)$

iv. $y^{(i)}_{e^{(i)}} = \texttt{Output}(\mathsf{View}^{(i)}_{e^{(i)}})$

v. $y^{(i)}_{e^{(i)}+1} = \texttt{Output}(\mathsf{View}^{(i)}_{e^{(i)}+1})$ // Note that $\mathsf{View}^{(i)}_{e^{(i)}+1}$ was explicitly given as part of $z^{(i)}$

vi. $y^{(i)}_{e^{(i)}+2} = y \oplus y^{(i)}_{e^{(i)}} \oplus y^{(i)}_{e^{(i)}+1}$

(b) Compute the commitments for views $\mathsf{View}^{(i)}_{e^{(i)}}$ and $\mathsf{View}^{(i)}_{e^{(i)}}$. In particular, for $j \in \{e^{(i)}, e^{(i)}+1\}$:
Commit: $[C^{(i)}_j, D^{(i)}_j] = [H(k^{(i)}_j, \mathsf{View}^{(i)}_j), k^{(i)}_j || \mathsf{View}^{(i)}_j]$

(c) Denote $a'^{(i)} = (y^{(i)}_1, y^{(i)}_2, y^{(i)}_3, C^{(i)}_1, C^{(i)}_2, C^{(i)}_3)$ // Note that $y^{(i)}_{e^{(i)}+2}$ and $C^{(i)}_{e^{(i)}+2}$ were explicitly given as part of $z^{(i)}$

2. Compute the challenge: $e' = H(a'^{(1)}, a'^{(2)}, \cdots, a'^{(t)})$. If, $e' = e$, output Accept. Else, output Reject.

3.2.2 Modification 2: Not including input shares

Recall from Section 3.2.1 that we changed the way the input shares are generated. In particular, for two of the three views, the input share x_i is generated from the seed k_i. Therefore, when we open either of those two views, we do not need to include the input share as the verifier can recompute it.

The exact savings of this modification depends on the challenge, e. If the $e = 1$, then for both of the opened views we can derive the input share from the seed and do not need to send it explicitly. However, if $e = 2$ or $e = 3$, we still need to send one input share for the third view for which the input share cannot be derived from the seed. Since the challenge is generated uniformly at random from $\{1, 2, 3\}$, the expected number of input shares that we'll need to include for a single iteration is $\frac{2}{3}$.

3.2.3 Modification 3: Not including commitments

In the ZKB proofs, the commitments of all three views are sent to the verifier. However, this is unnecessary since for the two views that are opened, the verifier can recompute the commitment itself. Only for the third view that the verifier is not given does the prover need to explicitly send the commitment.

We stress that there is no lost security here as even when the prover sends the commitments, the verifier must check that the prover has sent the correct commitments by hashing the commitments to recompute the challenge. Here too, the verifier checks that the commitments that it computed are the same ones that were used by the prover by hashing them as part of the input to recompute the challenge.

There is also no extra computational cost in this approach – whereas the verifier now must recompute the commitments, in the original ZKB protocol, the verifier needed to verify the commitments in step 2a (see Figure 3.1.3). For the hash-based commitment scheme used by ZKB, the function to verify the commitment first recomputes the commitment and thus there is no extra computation.

3.2.4 Modification 4: 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. Further, each view included in the commitment must be well randomized for the security of the MPC protocol. Clearly, commitments without an extra randomizing value are computationally indistinguishable from those that have and explicit randomizing value.

3.2.5 Modification 5: Not including the output shares

In the ZKB proofs, as part of a, the output shares y_i are included in the proof. Moreover, for the two views that are opened, those output shares are included a second time.

First, we don't need to send two of the output shares twice. However, 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 sent as part of the proof, the output share can be recomputed from the remaining parts of the view. In particular, the output share is just the value on the output wires. However, given the random tapes together with the communicated bits from the binary multiplication gates, the verifier can recompute all wires for both views.

For the third view, recall that the `Reconstruct` function simply XORs 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:

$$y_{i+2} = y + y_i + y_{i+1}$$

Computational trade-off. While we would expect some computational cost from recomputing rather than sending the output shares, our benchmarks show that there is no additional computational cost incurred by this modification, perhaps because it is a small part of the overall verification. For the challenge view, View_e, the verifier anyway needs to recompute all of the wire values in order to do the verification, so there is no added cost.

For the second view, as well View_{e+1}, the verifier must recompute the wire values as well since the verifier will need to compute the bits that are "communicated" in the MPC, so there is effectively no cost.[*]

[*]We note that the reason that in the ZKB proofs we need to recompute the second view as well is

Finally, for the third view, the extra computational cost of recomputing the output share is just two additions in the ring, which is exactly the cost of a single call to `Reconstruct`.

However, in step 2a of the ZKB verification, the verifier had to call `Reconstruct` in order to verify that the three output shares given were correct (see Figure 3.1.3). But in our modification, the verifier no longer needs to perform this check as the derivation of the third share guarantees that it will reconstruct correctly. Thus, the verifier is adding one `Reconstruct` but saving one, and thus no cost is incurred.

We note that the outputs will be checked as the y_i's are hashed with H to determine the challenge. The verifier recomputes the challenge and if the y_i values used by the verifier do not match those used by the prover, the challenge will be different (by the collision resistance property of H), and the proof will fail.

3.2.6 Modification 6: Not including View_e

In step 2a of the proof, the verifier recomputes every wire in View_e and checks as he goes that the received values are correct. However we note that this is not necessary.

The verifier can recompute View_e given just the random tapes k_e, k_{e+1} and the wire values of View_{e+1}. But the verifier does not need to explicitly check that each wire value in View_e is computed correctly. Instead, the verifier will recompute the view, and check the commitments using the recomputed view. By the binding property of the commitment scheme, the commitments will only verify if the verifier has correctly recomputed every value stored in the view.

Notice that this modification reduces the computational time as the verifier does not need to perform step 2c, i.e, there is no need to check every wire as checking the commitment will check these wires for us. But more crucially, this modification reduces the proof size significantly. There is no need to send the AND wire values for View_e as we can recompute them and check their correctness. Indeed, for this view, the prover only needs to send the input wire value and nothing else.

because the values that we are storing in the view are not the values that are actually communicated. The bits that are communicated are the input wires to binary multiplication gates. While ZKB could have stored those directly, the cost would be two values per multiplication gates.

Instead, we send the output values of the binary multiplication gates. These values alone allow the verifier to recompute the values on all wires, and thus they can recompute the input wire values that are communicated. Doing it this way is more efficient space-wise as we only need to send 1 value per binary multiplication gate instead of 2. Yet it comes at the computational trade-off as the verifier now needs to recompute the second view to recover the communicated bits.

However ZKB was already making this trade-off (indeed they don't even discuss the reasoning or the alternative that we presented). Thus considering that they already were doing this, our modification comes at no cost as the re-computation of the view was being done regardless.

3.2.7 Putting it all together: ZKB++

We have proposed a series of modifications to the ZKB protocol, and present our new protocol in Figure 3.2.1.

Notice that in the new protocol, the prover explicitly sends the challenge e to the verifier. In the original ZKB protocol, the verifier was explicitly given all of the inputs to the challenge random oracle, so it could compute the challenge right away, and then check the proofs. However, in our protocol, the verifier is no longer explicitly given these inputs. Thus our verifier must first run the proof system, recompute the commitments, and then compute the challenge.

But this leads to a problem. There is no way for the verifier to know which order to has the commitments. The verifier knows the relative ordering of the commitments relative to the challenge (i.e., which is $e, e+1$, and $e+2$), but in order to check that the challenge was generated correctly it needs to be given e, and we therefore include e explicitly.

There are 3 possible challenges for each iteration, so the cost of sending e for an r iteration proof is

$$r \times \log_2(3)$$

Readers familiar with Schnorr proofs of knowledge of a discrete logarithm will note this is analogous to the choice in that protocol of sending (c, s) or (r, s) (where (r, c, s) is a Σ-protocol transcript as defined in Section 2). When r is larger than c, as is the case when when working in a finite field, this reduces the proof size.

3.2.8 ZKB++ proof size

Our notation is as before. The expected size of our new proofs is given by

$$\begin{aligned}|p| &= r \times [|c_i| + 2|k_i| + \frac{2}{3}|x_i| + b|w_i| + |e_i|]\\ &= r \times [c + 2\kappa + \frac{2}{3}\ell m + b\ell + \log_2(3)]\\ &= r \times [c + 2\kappa + \log_2(3) + \ell \times (\frac{2}{3} \times m + b)]\end{aligned}$$

3.2.9 Concrete Size Savings

In order to give some intuition as to how much smaller our proofs are in practice, we give a concrete example. In the ZKB paper, SHA-2 is used as an example. with $r = 137$ rounds[*]. The ring is $R = \mathbb{Z}_2$, so $\ell = 1$. The circuit used to evaluate SHA-2 had 23,296

[*]Our numbers are slightly higher than those reported in [10]. While the authors stated that that they used 137 rounds, their benchmarking code actually only ran 136 rounds.

binary multiplications (corresponding to 728 32-bit ANDs/ADDs).Thus $b = 23,296$. For commitments, SHA-256 was also used, so $c = 256$, and commitments were randomized with a 32-bit random value s [*]. SHA-256 has a 256-bit output, so $n = 256$,and in [10] it is computed it with input size $m = 256$. Finally, the seeds for the random tapes were 128 bits long, so $\kappa = 128$.

The size of the ZKB proof for these parameters is:

$$\begin{aligned}
|p| &= r \times [3c + 2\kappa + 2s + \ell \times (5n + 2m + 2b)] \\
&= 137 \times [3(256) + 2(128) + 2(32)1 \times (5(256) + 2(512) + 2(23296)) \\
&= 6,847,808 \text{ bits} \\
&= 855.976 \text{ kilobytes}
\end{aligned}$$

With our modifications, the expected proof size for the same function and security parameters becomes:

$$\begin{aligned}
|p| &= r \times [c + 2\kappa + \log_2(3) + \ell \times (\frac{2}{3} \times m + b)] \\
&= 137 \times [256 + 2(128) + \log_2(3) + 1 \times (\frac{2}{3} \times 256 + 23,296)] \\
&= 3,285,295 \text{ bits} \\
&= 410.662 \text{ kilobytes}
\end{aligned}$$

Notice that we have cut the proof size by more than half. In truth, we can even make the proof smaller. ZKB uses 256-bit commitments and 128-bit random tapes. Yet, the proof was only targeting 80-bit security, and so the 256 and 128 could be reduced to 80 and 160, respectively. For the sake of a fair comparison though, we used the original ZKB parameter choices and still showed that our modifications reduce the proof size significantly.

4 Non-Interactive Transforms and PQ Security

In this section we describe the Fiat-Shamir and Unruh constructions for transforming an interactive sigma protocol to a non-interactive one, and consider the security against quantum adversaries.

[*] We do not agree with the use of 32-bits for s. We believe that if a random nonce was needed for the commitments, 32-bits does not suffice, and as we showed, we don't believe that a random nonce is actually needed at all. We use $|s| = 32$ here simply to get an accurate comparison of the savings of our modifications with the exact parameters from [10].

4.1 Post-quantum security of ZKB++

In order to transform an interactive sigma protocol with the Fiat-Shamir transformation, we require that it has the special soundness property and is honest-verifier zero-knowledge (HVKZ). The ZKB paper [10] shows that both of these properties hold for the ZKB protocol, and they hold for ZKB++ as well by the same arguments.

The HVZK property reduces to:

1. The 2-privacy property (i.e. the existence of the 2-privacy simulator) of the underlying MPC scheme, which is information theoretic.

2. The hiding property of the commitment scheme (with which you are also committing to the third "garbage" view that will not be opened).

The 2-privacy property of the MPC scheme, in turn, relies on indistinguishability of a uniformly distributed value and a value of the form $\phi_e(c) = Z - R_{i+1}(c)$, where Z depends on the gate. Were $R_{i+1}(c)$ uniformly distributed, these would be identical, but in practice it is pseudorandomly generated. Thus, for the 2-privacy property to hold, we rely on the security of the pseudorandom generator.

Thus, the two properties that we need to show that ZKB++ is HVZK in the presence of a quantum adversary is the quantum security of the commitments and the security of the pseudorandom generator.

The special soundness property reduces to

1. Security of the underlying MPC

2. The binding property of the commitment scheme.

4.1.1 Post-quantum commitments

For the Unruh transform, we will have to assume that H, our hash function is a quantum random oracle. Although we can construct secure post-quantum commitments without random oracles (see [16]), since we must assume that H is a random oracle anyway, we use the result of [15] (Lemma 25) which shows that the canonical hash-based commitment scheme is *collapse-binding* – i.e., it is a secure commitment scheme against quantum adversaries (with security as defined in [15]).

To instantiate the random oracle, we use SHA-256 (recall our discussion of collision resistance in Section 2).

4.2 Fiat-Shamir

The Fiat-Shamir transform [9] is an approach for converting a sigma protocol into a non-interactive zero knowledge proof of knowledge. A sigma protocol consists of a transcript (r, c, s) as defined in Section 2. The corresponding non-interactive proof (r', c', s') would generate c' and s' as in the interactive case, but generate $e' = H(c')$, instead of receiving it from the verifier. This is known to be a secure NIZK in the random oracle model against standard (non-quantum) adversaries [9].

One simple approach to arguing post-quantum security is to assume that the Fiat-Shamir heuristic also works against quantum adversaries, as long as the underlying sigma protocol and the hash function used to instantiate the random oracle are quantum-secure. The paper [7] cites lattice-based signature schemes that make this assumption, sometimes implicitly. Given the general uncertainty of the capabilities of quantum attackers, we prefer to avoid this assumption, provided there is an efficient alternative.

4.2.1 ZKB++ number of rounds with Fiat-Shamir

To get soundness of $2^{-\sigma}$, ZKB++, like ZKB, requires

$$t = \sigma(\log_2 3 - 1)^{-1}$$

iterations.

This, however, is for classical security. For post-quantum security, we will need to double the number of rounds required to protect against Grover's algorithm.

To see why this is so, we can formulate an attacker 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. Then there are 3^t possible challenges that can come from hashing those N 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 therefore applies and allows the attacker to search the space in time $(\frac{3}{2})^{t/2}$.

To get soundness of $2^{-\sigma}$ then, which corresponds to a search time of 2^σ, we solve for t such that $2^\sigma = (\frac{3}{2})^{t/2}$ and we find

$$t = 2 \cdot \sigma(\log_2 3 - 1)^{-1}$$

This is exactly double the number of rounds required for a classical adversary. In order achieve 128-bit quantum security, we will need 438 iterations.

4.3 Unruh's transform

Unruh [14] presents an alternative to the Fiat-Shamir transform that is provably secure in the QROM. Indeed, Unruh even explicitly presents a construction for a signature scheme and proves its security. We use his approach to argue that with a few modifications, our signature scheme is also provably secure in this model. One interesting aspect is that, while on first observation Unruh's transform seems much more expensive than the standard Fiat-Shamir transform, we show how to make use of the structure of ZKB++ to reduce the cost significantly.

4.3.1 Unruh's transform: overview

At a high level, Unruh's transform works as follows: Given a sigma protocol with challenge space C an integer t, a statement x and a random permutation G, the prover will

1. Run the first phase of the sigma protocol t times to produce $r_1, \ldots, r_t$.
2. For each $i \in \{1, \ldots, t\}$, and for each $j \in C$, compute the response s_{ij} for r_i and challenge j. Compute $g_{ij} = G(s_{ij})$.
3. Compute $H(x, r_1, \ldots, r_t, g_{11}, \ldots, g_{t|C|})$ to obtain a set of indices $J_1, \ldots, J_t$.
4. Output $\pi = (r_1, \ldots, r_t, s_{1J_1}, \ldots, s_{tJ_t}, g_{11}, \ldots, g_{t|C|})$.

Similarly, the verifier will verify the hash, verify that the given s_{iJ_i} values match the corresponding g_{iJ_i} values, and that the s_{iJ_i} values are valid responses for the corresponding r_i values.

At a very high level, in Unruh's security analysis, zero knowledge follows from HVZK of the underlying sigma protocol - the simulator will just generate t transcripts and then program the random oracle to get the appropriate challenges. The proof of knowledge property is more complex, but very roughly, the argument is that any adversary who has non-trivial probability of producing an accepting proof will also have to output some g_{ij} for $j \neq J_i$ which is a correct response for a different challenge - then the extractor can invert G and get the second response, which by special soundness allows it to produce a witness.

To instantiate the protocol, Unruh shows that one does not need a random oracle that is actually a permutation. Instead, as long as the domain and range of the random function are the same, it can be used in place of a permutation, since it is indistinguishable from a random permutation.

4.3.2 Applying the Unruh transform to ZKB++: the direct approach

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

Since ZKB++ has three special soundness, we would need at least three responses for each iteration. Moreover, since there only are three possible challenges in ZKB++, we would run Unruh's transform with $C = \{1, 2, 3\}$ – i.e. every possible challenge and response. We would then proceed as follows

Let $G : \{0,1\}^{|s_{ij}|} \to \{0,1\}^{|s_{ij}|}$ be a hash function modelled as a random oracle.[*] Non-interactive ZKB++ proofs would then proceed as follows (using the notation of Section 4.3.1):

1. Run the first phase of ZKB++ t times to produce $r_1, \ldots, r_t$.
2. For each $i \in \{1, \ldots, t\}$, and for each $j \in 1, 2, 3$, compute the response s_{ij} for r_i and challenge j. Compute $g_{ij} = G(s_{ij})$.
3. Compute $H(x, r_1, \ldots, r_t, g_{11}, \ldots, g_{t3})$ to obtain a set of indices $J_1, \ldots, J_t$.
4. Output $\pi = (r_1, \ldots, r_t, s_{1J_1}, \ldots, s_{tJ_t}, g_{11}, \ldots, g_{t3})$.

While this works, it comes as a significant overhead in the size of the proof. Recall from Section 3.2.8 that the proof size for ZKB++ is:

$$\begin{aligned}
|p| &= t \times [|c_i| + 2|k_i| + \frac{2}{3}|x_i| + b|w_i| + |e_i|] \\
&= t \times [c + 2\kappa + \frac{2}{3}\ell m + b\ell + \log_2(3)] \\
&= t \times [c + 2\kappa + \log_2(3) + \ell \times (\frac{2}{3} \times m + b)]
\end{aligned}$$

where t is the number of rounds, c_i are commitments which are of size c, k_i are random tapes of length κ, the x values are the input shares which consist of m wires, the e values are the challenge bits, b is the number of binary-multiplication gates in the circuit, and each wire w_i can be expressed with ℓ bits.

For the Unruh transform, we'd have to additionally include $g_{11}, \ldots, g_{t3}$. Each g_{ij} is a permutation of an output share and there are $3t$ such values, so in particular the extra overhead would incur the cost of

[*]Actually, the size of the response changes depending on what the challenge is. If the challenge is 0, the response is slightly smaller as it doesn't 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.

$$
\begin{aligned}
3t \times |s_i j| &= 3t \times [2|k_i| + \frac{2}{3}|x_i| + b|w_i|] \\
&= 3t \times [2\kappa + \frac{2}{3}\ell m + b\ell] \\
&= 3t \times [2\kappa + \ell \times (\frac{2}{3} \times m + b)]
\end{aligned}
$$

And the total cost of an Unruh-ZKB++ proof would then be

$$
\begin{aligned}
& t \times [c + 2\kappa + \log_2(3) + \ell \times (\frac{2}{3} \times m + b)] + 3t \times [2\kappa + \ell \times (\frac{2}{3} \times m + b)] \\
= \;& t \times [c + 8\kappa + log_2(3) + \ell \times (\frac{8}{3}m + 4b)]
\end{aligned}
$$

Since for most functions, the size of the proof is dominated by $t \times \ell b$, this proof is roughly four times as large as the Fiat-Shamir version.

4.3.3 Modification 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*, s_{ij}, we can compute a permutation of each *view*, v_{ij}. For each $i \in \{1, \ldots, t\}$, and for each $j \in \{1, 2, 3\}$, the prover computes $g_{ij} = G(v_{ij})$.

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

4.3.4 Modification 2: omit re-computable values

Moreover, since G is a public function, we do not need to include $G(v_{ij})$ in the transcript if we have included v_{ij} 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.

4.3.5 Putting it together: new proof size

The combination of these two modifications yields a major reduction in proof size. For each of the t iterations of ZKB++, we include just a single extra $G(v)$ than we would in the Fiat-Shamir transform.

As G is a permutation, the per-iteration overhead of ZKB++/Unruh over ZKB++/Fiat-Shamir is the size of a single view. This overhead is less that one-third of the overhead that would be incurred from the naive application of Unruh as described in Section 4.3.2. In particular, the overhead of our optimized version is

$$\begin{aligned} t \times |v_{ij}| &= t \times [|k_i| + \frac{1}{3}|x_i| + b|w_i|] \\ &= t \times [\kappa + \frac{1}{3}\ell m + b\ell] \\ &= t \times [\kappa + \ell \times (\frac{1}{3} \times m + b)] \end{aligned}$$

Notice the term $\frac{1}{3}|x_i|$. Here x_i is the third input share that is not derivable from its random tape. When considering views, there is a $\frac{1}{3}$ chance that it will be included (as opposed to the $\frac{2}{3}$ probability when considering responses as each response contains two views.

And the total size of a ZKB++/Unruh proof would then be

$$\begin{aligned} & t \times [c + 2\kappa + \log_2(3) + \ell \times (\frac{2}{3} \times m + b)] + t \times [\kappa + \ell \times (\frac{1}{3} \times m + b)] \\ = \; & t \times [c + 3\kappa + log_2(3) + \ell \times (m + 2b)] \end{aligned}$$

The overhead varies per function. For the functions we examined in Section 5, the overhead ranges from 1.6 to 2x compared to the equivalent ZKB++/Fiat-Shamir proof.

4.3.6 Security argument of the modified Unruh transform

To argue zero knowledge, we can take the same approach as in Unruh: to simulate the proof we choose the set of challenges $J_1, \ldots, J_t$, run the MPC simulator to obtain views for each pair of dishonest parties J_i, J_{i+1}, honestly generate g_{iJ_i} and $g_{iJ_{i+1}}$ and the commitments to those views, and choose $g_{J_{i+2}}$ and the corresponding commitment at random. Then we program the random oracle to output $J_1, \ldots, J_t$ on the resulting tuple. The analysis follows exactly as in Unruh[15].

For the soundness argument, our protocol has two main differences from Unruh's general version: 1) the underlying protocol we use only has 3-special soundness, rather than the normal 2-special soundness, and 2) we have one commitment for each view, and one $G(v)$ for each view, rather than having a separate $G(view_i, view_{i+1})$ for each i.

As mentioned above, the core of Unruh's argument, which appears in Lemma 17 in [15], says that the probability that the adversary can find a proof such that the extractor cannot extract but the proof still verifies is negligible.

For our case, we can make that analysis as follows: For a given tuple of commitments $r_1 \dots r_t$, and G-values $g_{11}, g_{t|C|}$ that is queried to the random oracle either one of the following is true: 1) There is some i for which $(G^{-1}(g_{i1}),\, G^{-1}(g_{i2}))$, $(G^{-1}(g_{i2}),\, G^{-1}(g_{i3}))$, $(G^{-1}(g_{i3}),\, G^{-1}(g_{i1}))$, are valid responses for challenges $1, 2, 3$ respectively*, or 2) For all i at least one of these pairs is not a valid response. In particular that means that if that challenge is produced by the hash function, $\mathcal{A}$ will not be able to produce an accepting response. From that, we can argue that if the extractor cannot extract from a given tuple, then the probability (over the choose of a random RO) that there exists an accepting response for $\mathcal{A}$ to output is at most $(2/3)^t$. Then, we can rely on Unruh's Lemma 7, which tells us that given q_H queries, the probability that $\mathcal{A}$ produces a tuple from which we cannot extract but $\mathcal{A}$ can produce an accepting response is at most $2(q_H + 1)(2/3)^t$.

The rest of our argument can proceed exactly as in Unruh's proof.

4.3.7 Unruh's Transform with Constant Overhead?

We conjecture that we may be able to further reduce the overhead of Unruh to a fixed size that does not depend on the circuit being used. We leave this as a conjecture for now as it does not follow from Unruh's proof, and we have not yet proved it.

If we were to include just the hash using G of the seeds (and the third input share that is not derivable from its seed), it seems that this would be enough for the extractor to produce a witness.

Combining this with the previous optimizations, we only need to explicitly give the extractor a permutation of the input share of the third view. For the first two views, the views are communicated in the open, and the extractor can compute the permutation himself.

This would reduce the overhead when compared to FS from about 1.5x to 1.16x.

5 Our Signature Scheme

We now combine ZKB++ with the Fiat-Shamir and Unruh transforms to build a post-quantum signature scheme. Our construction uses a block cipher to define the relation between private and public keys (i.e., use ZKB++ to prove knowledge of a key used to create a public ciphertext). This is because practical block cipher constructions generally have

*In fact G is not exactly a permutation, but we ignore that here. We can make this formal exactly as in Unruh's proof, by considering the set of preimages.

much lower circuit-complexity than hash functions, and will therefore lead to significantly smaller signatures. We considered using a hash function as well. While the security of this option is straightforward (given a wide enough output), the resulting signatures would be much larger.

Let f_k be a pseudorandom permutation with key k, key size of n bits and block size of n bits, where n is large enough to be secure against a quantum adversary ($n = 256$ in our implementation).

$(pk, sk) \leftarrow$ `Key-Gen`(1^n)

1. Choose $k \in_R \{0,1\}^n$.
2. Choose $r \in_R \{0,1\}^n$.
3. Compute $y = f_k(r)$.
4. Output $(y, r), k$, where (y, r) is the public key and k is the private key.

$\sigma \leftarrow$ `Sign`$_k(m)$

1. Construct a ZKB++ proof of knowledge of k, where m is included when computing the challenge: $\pi \leftarrow$ `ZK-Prove`(k, y, r).
2. Output proof π.

$b \leftarrow$ `Verify`$_{pk}(\pi)$

1. Run ZKB++ verification: `ZK-Verify`(pk, π), again m included when computing the challenge.
2. Output 'valid' iff `ZK-Verify` outputs 'valid', and output 'invalid' otherwise.

Note: the rationale for using a random block r as input to f_k when creating the key pair is to improve security against multi-user key recovery attacks and generic time-memory trade-off attacks like [11].

For many block ciphers, the input key is first expanded to multiple round keys, and key expansion can be a non-trivial part of the total encryption cost. One might consider proving knowledge of the expanded key, however, this is insecure. Taking AES as an example, the attacker chooses all but the last round key at random, encrypts r with these keys, then finds the last round key that maps the intermediate ciphertext to y (which is possible since a single round of AES provides very little security). Therefore, it is important that we prove knowledge of the input key, rather than an expanded key.

5.0.8 Security of Key Generation

In this section, we show that a block cipher in which the block size and key size are equal, and in particular equal to the security parameter n can serve as our hard instance generator to generate keys for our signature scheme. While it is clear that a quantum preimage-resistant hash function has this property, we show that a block cipher has this property as well.

Definition 8 (Hard Instance Generators [see Definition 20 in [14]]) *We call an algorithm G a* **hard instance generator** *for a relation R iff*

1. $\Pr[(p,s) \in R : (p,s) \leftarrow G()]$*is overwhelming and*
2. *for any polynomial-time algorithm A, $Pr[(p,s') \in R : (p,s) \leftarrow G, s' \leftarrow A(p)]$ is negligible.*

Definition 9 (Pseudorandom permutation) *Let $F : \{0,1\}^k \times \{0,1\}^b \to \{0,1\}^n$ be a an efficiently computable keyed permutation. We say that F is a* **pseudorandom permutation** *if for all probabilistic polynomial time distinguishers D,*

$$|\Pr[D^{F_k,F_k^{-1}}(1^{|k|}) = 1] - \Pr[D^{f_n,f_n^{-1}}(1^{|k|}) = 1]|$$

is negligible where $k \leftarrow \mathcal{K}$ is chosen uniformly at random and f_n is chosen uniformly at random from the set of permutations of n-bit strings.

Construction 1 *Let $F : \mathcal{K} \times \mathcal{M} \to \mathcal{M}$ be a pseudorandom permutation.*

1. *Choose a value $r \in \mathcal{M}$ uniformly at random.*
2. *Choose a key $k \leftarrow \mathcal{K}$ uniformly at random.*
3. *Compute $y = F_k(x)$.*
4. *Output (y, r, k) ((y, r) is the instance and k is the witness).*

Theorem 1 *Construction 1 is a hard instance generator for relation $R := \{(y, r, k) : F_k(x) = y\}$ if F is a pseudorandom permutation with $|\mathcal{M}| \geq |\mathcal{K}|$.*

Proof. The first condition of Definition 8 clearly holds as a triple (y, r, k) output by Construction 1 will always be in R. Thus, we need only show that the second condition holds. In particular, we need to show that the following probability is negligible for any PPT algorithm A:

$$P_1 := \Pr[(y, r, k' = k) \in R : (y, r, k) \leftarrow G, k' \leftarrow A(y)]$$

We denote by $\texttt{keyset}(y)$ the set of keys $\mathcal{B}$ such that for all $k \in \mathcal{B}, F_k(r) = y$. If there is no such satisfying key, $\texttt{keyset}(y)$ returns the empty set. For an algorithm A, denote by t_y the probability that A will output a key k' on input y such that $F_{k'}(r) = y$. Then, using this notation we can rewrite P_1 as

$$P_1 := \sum_y \frac{|\texttt{keyset}(y)|}{|\mathcal{K}|} \cdot t_y$$

and we need to show that P_1 is negligible for any A.

First, we define, probability P_2, which is the probability that A will output the "correct key", by which we mean the same key that was chosen to generate y. Since the key was chosen uniformly at random, information-theoretically, there is no way for A to distinguish between the "correct key" and any other valid key (i.e. any k' for which $F_{k'}(r) = y$). Thus, the only strategy that A has is to output any valid key and with probability $\frac{1}{\texttt{keyset}(y)}$, the key that it outputs will be the "correct key". Thus, we have:

$$\begin{aligned} P_2 :&= \Pr[k = k' : (y, r, k) \leftarrow G, k' \leftarrow A(y)] \\ &= \sum_y \frac{|\texttt{keyset}(y)|}{|\mathcal{K}|} \cdot t_y \cdot \frac{1}{|\texttt{keyset}(y)|} \\ &= \frac{1}{|\mathcal{K}|} \sum_y t_y \end{aligned}$$

We now show that P_2 is negligible. Assume that there exists an A for which P_2 is equal to non-negligible ϵ. Then we can build a distinguisher D that distinguishes between F and a random permutation as follows:

$D^{\mathcal{O}}(1^{|k|})$

1. $y' \leftarrow \mathcal{O}(r)$. Queries the oracle on r and receive response y'.
2. Invoke A on input y'.
   (a) if $\perp \leftarrow A(y')$, output 0.
   (b) if $k^* \leftarrow A(y')$, check that this is the "correct key" as follows
      i. First check that $F_{k^*}(r) = y$. If not, output 0. Else, continue
      ii. Next, choose a value $q \leftarrow \mathcal{M}$ uniformly at random and query on that value – i.e. query for $z \leftarrow \mathcal{O}(q)$.
      iii. Check that $z = F_{k^*}(q)$. If it does not, output 0. If it does, output 1.

Now, let's analyze the output of D. Whenever A outputs the "correct key", D will output 1. Moreover, A will output the correct key with probability ϵ. Thus, if D's oracle is a pseudorandom function – i.e. if $\mathcal{O} = F$, then with probability at least ϵ, D will output 1. To see that this is true notice that when $\mathcal{O} = F$, the key for F is chosen from the same distribution as it is in Construction 1, and thus A's success probability on outputting the "correct key" will be exactly ϵ.[*]

If, however, $\mathcal{O}$ is a random permutation – i.e. $\mathcal{O} = f_n$, then D will only output 1 in the event that $f_n(q) = F_{k^*}(q)$. In step (iii), once we have chosen a key k^*, the probability of the random function agreeing with F_{k^*} on q is $\delta = \frac{1}{|\mathcal{M}|-1}$, which is negligible in $|k|$ since $|\mathcal{M}| \geq |\mathcal{K}| = 2^{|k|}$.

Thus, we have built a good distinguisher since:

$$|\Pr[D^{F_k, F_k^{-1}}(1^{|k|}) = 1] - \Pr[D^{f_n, f_n^{-1}}(1^{|k|}) = 1]| \geq \epsilon - \delta = \mathtt{non-negligible}$$

This contradicts our assumption that F is a pseudorandom permutation, and we therefore conclude that P_2 is negligible.

We now show that $|P_1 - P_2|$ is negligible. Once again, consider an algorithm A that on input y outputs a key k' such that $F_{k'}(r) = y$with probability t_y. Consider the following two games.

Game 1. A key $k \leftarrow \mathcal{K}$ is chosen uniformly at random and $y = F_k(r)$ is given to the adversary. The adversary wins if it can produce a key K' such that $F_{K'}(r) = y$. The probability of A succeeding at this game is exactly P_1:

$$\sum_y \frac{|\mathtt{keyset}(y)|}{|\mathcal{K}|} \cdot t_y$$

Game 2. $y \leftarrow \mathcal{M}$ is chosen uniformly at random and given to the adversary. The adversary wins if it can output a key k' such that $F_{k'}(r) = y$. The difference between this game and the previous one is that now we choose y uniformly irrespective of the keys. Thus all y's will be chosen with equal probability no matter how many keys (if any) map r to y. The success probability of A in this game is

$$P_3 := \frac{1}{|\mathcal{M}|} \sum_y t_y$$

Now if you could distinguish between Game 1 and Game 2, you could build an algorithm D that distinguishes F from a random permutation. D simply queries its oracle at r, and send the response y to A. If the oracle is a pseudorandom permutation, then the success

[*]It is possible that when $\mathcal{O} = F$, D will output 1 with probability greater than ϵ –i.e. if A outputs the wrong key that happens to agree with the "correct key" on the queried values, but for the sake of our argument it suffices to show that it outputs 1 with probability at least ϵ.

probability will be exactly the same as Game 1, namely P_1. If it is a random function, the success probability is exactly the same as Game 2, namely P_3. Thus, by Definition 9, we know that $|P_1 - P_3|$ is negligible.

Since $|\mathcal{K}| \leq |\mathcal{M}|$, then $P_2 \leq P_3$, and in particular, when $|\mathcal{K}| = |\mathcal{M}|$, $P_2 = P_3$. We thus have that $|P_1 - P_2|$ is negligible. Since we have shown that both P_2 and $|P_1 - P_2|$ are negligible, it follows that P_1 is negligible as well.□

$$|\Pr[D^{F_k, F_k^{-1}}(1^{|k|}) = 1] - \Pr[D^{f_n, f_n^{-1}}(1^{|k|}) = 1]| \geq \epsilon - \delta = \mathtt{non-negligible}$$

This contradicts our assumption that F is a pseudorandom permutation, and we therefore conclude that P_2 is negligible.

We now show that $|P_1 - P_2|$ is negligible. Once again, consider an algorithm A that on input y outputs a key k' such that $F_{k'}(r) = y$with probability t_y. Consider the following two games.
Game 1. A key $k \leftarrow \mathcal{K}$ is chosen uniformly at random from and $y = F_k(r)$ is given to the adversary. The adversary wins if it can produce a key K' such that $F_{K?}(r) = y$. The probability of A succeeding at this game is exactly P_1:

$$\sum_y \frac{|\mathtt{keyset}(y)|}{|\mathcal{K}|} \cdot t_y$$

Game 2. $y \leftarrow \mathcal{M}$ is chosen uniformly at random and given to the adversary. The adversary wins if it can output a key k' such that $F_{k'}(r) = y$. The difference between this game and the previous one is that now we choose y uniformly irrespective of the keys. Thus all y's will be chosen with equal probability no matter how many keys (if any) map r to y. The success probability of A in this game is

$$P_3 := \frac{1}{|\mathcal{M}|} \sum_y t_y$$

Now if you could distinguish between Game 1 and Game 2, you could build an algorithm D that distinguishes F from a random permutation. D simply queries its oracle at x, and send the response y to A. If the oracle is a pseudorandom permutation, then the success
probability will be exactly the same as Game 1, namely P_1. If it is a random function, the success probability is exactly the same as Game 2, namely P_3. Thus, by Definition 9, we know that $|P_1 - P_3|$ is negligible.

Since $|\mathcal{K}| \leq |\mathcal{M}|$, then $P_2 \leq P_3$, and in particular, when $|\mathcal{K}| = |\mathcal{M}|$, $P_2 = P_3$. We thus have that $|P_1 - P_2|$ is negligible. Since we have shown that both P_2 and $|P_1 - P_2|$ are negligible, it follows that P_1 is negligible as well.□

NIZK	# iter.	Security	Proof size	Prover	Verifier	Total
ZKB	137	80-bit classical	855,976	73,178,518	70,572,692	143,751,210
ZKB++	137	80-bit classical	413,379	72,530,004	37,229,857	109,759,861
ZKB	219	128-bit classical	1,368,312	117,037,363	103,751,608	220,788,971
ZKB++	219	128-bit classical	660,503	116,012,659	59,410,728	175,423,387
ZKB	438	256-bit classical	2,736,624	238,248,747	203,683,630	441,932,377
ZKB++	438	256-bit classical	1,321,966	233,322,809	119,235,511	352,558,320

Table 1: Comparison of ZKB and ZKB++ proving knowledge of a SHA-256 preimage. The proof size is given in bytes, and the running times are given in CPU cycles. The running times are averages taken over 500 runs. The last two rows also target 128-bit PQ security. Only the Fiat-Shamir transform was used in this table.

5.1 Implementation and Benchmarks

We benchmarked ZKB++ and compared it to ZKB, and used it with LowMC to implement our signature scheme.

All of our code is written in C, and our benchmarks are taken on a desktop with a 4-core Intel Core i7-6700 3.40GHz CPU running Ubuntu 14.04. For the running time comparison, we compiled with GCC using the -O3 and -March=native flags for optimization. We give all of our running time benchmarks in units of CPU cycles.

The current benchmarks focus on *signature size*, since CPU cost is expected to be practical. In any case, neither our implementation, nor the one of Giacomelli et al. [10] is highly optimized. We plan to create a highly optimized implementation for one of our LowMC parameter sets. These optimizations should be independent of the FS or Unruh transform, since they will concentrate on the MPC component of ZKB++.

5.1.1 ZKB vs. ZKB++

In Table 1 we benchmark both the size and running times of ZKB++ and compare it to ZKB. Our proof size are more than a factor of 2 smaller, and we also saw increased running times – particularly for the verifier.

5.1.2 Signature scheme implementation

As the first implementation of our scheme, we wanted to build a tool that would most easily allow us to test our scheme with different block-ciphers and hash functions. We therefore built a fully modular circuit-based implementation. As input, our implementation took a boolean-circuit text file that followed the standard format used in [13]. Our implementation then runs the signature scheme using the function specified in the circuit file. To verify a signature, our code similarly takes a circuit file and runs the verification algorithm using the given function. By decoupling the underlying symmetric function from MPC aspects of the signature scheme, our implementation allows one to "plug-in" any function they want without writing any code.

We benchmarked our implementation for AES and two configurations of LowMC. For the circuit files, we obtained an AES-circuit from [13].

In order to run our code with LowMC, we needed to obtain a binary circuit for it. As part of the ABY framework [8], circuit code was made available for LowMC. We began with that code and modified it to obtain compact circuits for the LowMC parameters that we benchmarked.

While the circuit-based implementation allows us to easily experiment with different functions for which we have circuit files, it suffers in efficiency since we are doing all of our operations as bitwise operations. We therefore created faster implementations for specific candidates, and we have benchmarked those here as well.[*]

For each candidate, we show the results both using plain Fiat-Shamir, as well as using Unruh's transformation. Note that, as expected, our modified Unruh scheme adds a fixed number of bytes that does not depend on the number of AND gates in the underlying symmetric function.

5.1.3 Parameters

For our post quantum choices, we targeted 128-bit post quantum security, and therefore chose our primitives to achieve 256-classical security. Based on Theorem 1, it was sufficient to have a block cipher with both a block size and a key size of 256. Moreover, for the base ZKB protocol, we need 256-bit classical soundness, and thus require 438 rounds based on our analysis in Section 4.2.1.

For LowMC, we require 256 bit blocks and cipher size. LowMC has two more parameters as well: data complexity and number of s-boxes. Given these parameters, we used the python script provided with the LowMC paper to determine the number of LowMC rounds required for security.

The data complexity parameter refers to the number of input/output pairs that an attacker will be given access to for a single key. In our signature scheme, the attacker only ever sees one such pair. However, in the simulation in the proof of Theorem 1, the attacker sees one more pair. Thus, the total data complexity is 2 pairs. In ZKB, the data complexity parameter is given as the base-2 logarithm of the number of pairs, and thus our data complexity is 1.

We were still free to choose the number of s-boxes. There is a signature size/efficiency trade-off since as we increase the number of s-boxes, the number of ANDs in the circuit increases, but the overall circuit size decreases. Thus, for our signature scheme, as we increase the number of s-boxes, the size of the signature will increase, but the time to sign and verify will decrease.

[*]Runtime benchmarks for the LowMC candidates are in preparation and will be added to a revision of this paper. For now our focus is signature size.

Table 2: Signature size comparison

Transform	**Primitive**	**ZKB iter.**	**Security**	**Signature size**
Fiat-Shamir	LowMC-256-256-1-243	438	128-bit PQ	91,670
Unruh	LowMC-256-256-1-243	438	128-bit PQ	150,490
Fiat-Shamir	LowMC-256-256-60-9	438	128-bit PQ	140,832
Unruh	LowMC-356-256-60-9	438	128-bit PQ	247,914
Fiat-Shamir	SHA-256	438	128-bit PQ	1,331,632
Unruh	SHA-256	438	128-bit PQ	2,630,414
Fiat-Shamir	AES-128	438	comparison only*	419,226
Unruh	AES-128	438	comparison only	805,602

Table 3: Comparison of signature size (in bytes) for various parameter choices, at the 128-bit PQ security level. The AES and SHA primitives are included for comparison only; for SHA the resulting signatures would be too large for practical use, and AES-128 does not provide sufficient security. The column "ZKB iter" is the number of parallel rounds for the ZKB protocol at the security level, see Section 4.2.1 for details.

We benchmark the endpoints of this trade-off here: the smallest signatures will result from choosing a single s-box and requires 243 rounds for security (we call this configuration LowMC-256-256-1-243). The smallest circuit will result from choosing 60 s-boxes as this requires only 9 rounds (LowMC-256-256-60-9). One may also choose 30 or 40 s-boxes as these are points that offer a tradeoff between these two extremes.

5.1.4 Signature size comparison

In Table 2, we compare the sizes for signatures using different underlying functions, and both the Fiat-Shamir and Unruh transforms are included.

References

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

[2] Mihir Bellare and Shafi Goldwasser. New paradigms for digital signatures and message authentication based on non-interative zero knowledge proofs. In Gilles Brassard,

*For 128-bit PQ security using an AES-variant, one would need to use Rijndael 256-256, but we did not have a circuit for this function.

editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 194–211, Santa Barbara, CA, USA, August 20–24, 1990. Springer, Heidelberg, Germany.

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

[4] Dan Boneh, Özgür Dagdelen, Marc Fischlin, Anja Lehmann, Christian Schaffner, and Mark Zhandry. Random oracles in a quantum world. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 41–69, Seoul, South Korea, December 4–8, 2011. Springer, Heidelberg, Germany.

[5] 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, Campinas, Brazil, April 20–24, 1998. Springer, Heidelberg, Germany.

[6] Jan Camenisch, Rafik Chaabouni, and abhi shelat. Efficient protocols for set membership and range proofs. In Josef Pieprzyk, editor, *ASIACRYPT 2008*, volume 5350 of *LNCS*, pages 234–252, Melbourne, Australia, December 7–11, 2008. Springer, Heidelberg, Germany.

[7] Özgür Dagdelen, Marc Fischlin, and Tommaso Gagliardoni. The Fiat-Shamir transformation in a quantum world. In Kazue Sako and Palash Sarkar, editors, *ASIACRYPT 2013, Part II*, volume 8270 of *LNCS*, pages 62–81, Bengalore, India, December 1–5, 2013. Springer, Heidelberg, Germany.

[8] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *NDSS 2015*, San Diego, CA, USA, February 8–11, 2015. The Internet Society.

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

[10] 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`.

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

[12] 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.

[13] Stefan Tillich and Nigel Smart. Circuits of basic functions suitable for mpc and fhe. `https://www.cs.bris.ac.uk/Research/CryptographySecurity/MPC/`.

[14] Dominique Unruh. Quantum proofs of knowledge. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 135–152, Cambridge, UK, April 15–19, 2012. Springer, Heidelberg, Germany.

[15] 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, Sofia, Bulgaria, April 26–30, 2015. Springer, Heidelberg, Germany.

[16] Dominique Unruh. Computationally binding quantum commitments. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 497–527, Vienna, Austria, May 8–12, 2016. Springer, Heidelberg, Germany.