Symmetric Cryptography from Alternating Moduli: Cryptanalysis, Distributed Protocols, and Applications

Abstract

Abstract goes here.

1 Introduction

2 Preliminaries

Notation. We start with some basic notation. For a positive integer k, $[k]$ denotes the set $\{1, \ldots, k\}$. $\mathbb{Z}_p$ denotes the ring of integers modulo p. We use bold uppercase letters (e.g., $\mathbf{A}, \mathbf{K}$) to denote matrices. We use $\mathbf{0}^l$ and $\mathbf{1}^l$ to denote the all zeros and the all ones vector respectively (of length l), and drop l when sufficiently clear. For distributed protocols with N parties, we use $\mathcal{P} = \{\mathrm{P}_1, \ldots, \mathrm{P}_N\}$ to denote the set of parties. For a value x in group $\mathbb{G}$, we use $[\![x]\!]$ to denote an additive sharing of x (in $\mathbb{G}$) among the protocol parties. When $\mathbb{G}' = \mathbb{G}^l$ is a product group (e.g., $\mathbb{Z}_p^l$), for $x \in \mathbb{G}'$, we may also say that $[\![x]\!]$ is a sharing *over* $\mathbb{G}$, similar to the standard practice of calling x a vector over $\mathbb{G}$.

3 Candidate Constructions

In this section, we introduce our suite of candidate constructions for a number of cryptographic primitives: weak pseudo-random function families (wPRF), one-way functions (OWF), pseudo-random generators (PRG), and cryptographic commitment schemes. Our constructions are based on similar interplays between mod-2 and mod-3 linear mappings.

3.1 Basic structure

Given the wide range of candidates we propose, we find it useful to have a clean way to describe the operations that are performed in our candidate constructions. For this, we take inspiration from the basic formalism of the function secret sharing (FSS) approach to MPC with preprocessing, first introduced by Boyle, Gilboa, and Ishai [BGI19]. Abstractly, the key technique here is to represent an MPC functionality as a circuit, where each gate represents an operation to be performed in the distributed protocol. Inputs and outputs of each gate are secret shared and the gate operation is "split" using a function secret sharing (FSS) scheme. To evaluate the circuit in a distributed fashion, the dealer first shares a random mask for each input wire in the circuit, and possibly some more correlated randomness. Now, to compute a gate, the masked input is first revealed to all parties, who can then locally compute shares of the output wire or shares of the masked output.

While we find it useful to use the formalism from [BGI19] for representing the circuit to be computed, we do not explicitly require the FSS formalism for splitting the functionality of each gate. The individual operations are quite straightforward, and we instead chose to directly provide the distributed protocols that compute them. Further, by doing so, our protocols can make better use of correlated randomness to reduce the overall protocol cost as compared to the general techniques in [BGI19].

Circuit gates. We make use of just four basic operations, or "gates," which we detail below. All our constructions can be succinctly represented using just these gates. In Section 5, we will provide distributed protocols to compute them. To cleanly describe both our candidates constructions, and their distributed protocols, the gates we describe here depart from the formalism in [BGI19] in that the input values are not secret shared. The distributed protocols will instead take in secret shared inputs as necessary.

- **Mod-p Public Linear Gate.** For a prime p, given a public matrix $\mathbf{A} \in \mathbb{Z}_p^{s\times l}$, the gate $\mathsf{Lin}_p^{\mathbf{A}}(\cdot)$ takes as input $x \in \mathbb{Z}_p^l$ and outputs $y = \mathbf{A}x \in \mathbb{Z}_p^s$.
- **Mod-p Bilinear Gate.** For a prime p, and positive integers s and l, the gate $\mathsf{BL}_p^{s,l}(\cdot,\cdot)$ takes as input a matrix $\mathbf{K} \in \mathbb{Z}_p^{s\times l}$ and a vector $x \in \mathbb{Z}_p^l$ and outputs $y = \mathbf{K}x \in \mathbb{Z}_p^s$. When clear from context, we will drop the superscript and simply write $\mathsf{BL}_p(\mathbf{K}, x)$.
- $\mathbb{Z}_2 \to \mathbb{Z}_3$ **conversion.** For a positive integer l, the gate $\mathsf{Convert}_{23}^l(\cdot)$ takes as input a vector $x \in \mathbb{Z}_2^l$ and returns its equivalent representation x^* in $\mathbb{Z}_3^l$. When clear from context, we will drop the superscript and simply write $\mathsf{Convert}_{23}(x)$.
- $\mathbb{Z}_3 \to \mathbb{Z}_2$ **conversion.** For a positive integer l, the gate $\mathsf{Convert}_{32}^l(\cdot)$ takes as input a vector $x \in \mathbb{Z}_3^l$ and computes its mapping x^* in $\mathbb{Z}_2^l$. For this, each $\mathbb{Z}_3$ element in x is computed

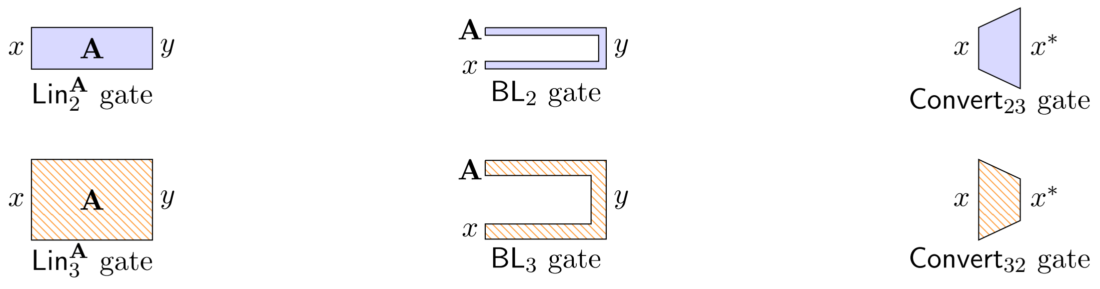


Figure 1: Pictorial representations of the circuit gates.

modulo 2 to get the corresponding $\mathbb{Z}_2$ element in the output x^*. Specifically, each 0 and 2 are mapped to 0 while each 1 is mapped to 1. When clear from context, we will drop the superscript and simply write $\mathsf{Convert}_{32}(x)$.

Note that the difference between the Lin and the BL gates is seen in the context of their distributed protocols. For Lin, the matrix $\mathbf{A}$ will be publicly available to all parties, while the input x will be secret shared. On the other hand, for BL, the both the key $\mathbf{K}$ and the input x will be secret shared. Also note that although the $\mathsf{Convert}_{23}$ gate is effectively a no-op in a centralized evaluation, in the distributed setting, the gate will be used to convert an additive sharing over $\mathbb{Z}_2$ to an additive sharing over $\mathbb{Z}_3$.

As described previously, for the linear mappings, we focus only on constructions that use mod-2 and mod-3 mappings. In Figure 1, we provide a pictorial representation for each circuit gate. We will connect these pieces together to also provide clean visual representations for all our constructions.

Construction styles. The candidate constructions we introduce follow one of two broad styles, which we detail below.

- **(p,q)-constructions.** For distinct primes p, q, the (p,q)-constructions have the following structure: On an input x over $\mathbb{Z}_p$, first a linear mod$p$ mapping is applied, followed by a linear mod$q$ mapping. Note that after the mod$p$ mapping, the input is first reinterpreted as a vector over q. For unkeyed primitives (e.g., OWF), both mappings are public, while for keyed primitives (e.g., wPRF), the key is used for the first linear mapping. The construction is parameterized by positive integers n, m, t (functions of the security parameter λ) denoting the length of the input vector (over $\mathbb{Z}_p$), the length of the intermediate vector, and the length of the output vector (over $\mathbb{Z}_q$) respectively. The two linear mappings can be represented by matrices $\mathbf{A} \in \mathbb{Z}_p^{m\times n}$ and $\mathbf{B} \in \mathbb{Z}_q^{t\times m}$. For keyed primitives, the key $\mathbf{K} \in \mathbb{Z}_p^{m\times n}$ will be used instead of $\mathbf{A}$.

  In this paper, we will analyze this style of constructions for $(p,q) = (2,3)$ and $(3,2)$. As a shorthand, we will denote these by 23-constructions and 32-constructions (e.g., 23-wPRF).

- **LPN-style-constructions.** [Mahimna: todo]

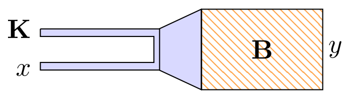
- [Mahimna: Also add any other constructions.]

23-constructions

Parameters. Let λ be the security parameter and define parameters n, m, t as functions of λ such that $m \geq n, m \geq t$.
Public values. Let $\mathbf{A} \in \mathbb{Z}_2^{m \times n}$ and $\mathbf{B} \in \mathbb{Z}_3^{t \times m}$ be fixed public matrices chosen uniformly at random.

Construction 3.1 (Mod-2/Mod-3 wPRF Candidate [BIP+18])**.** The wPRF candidate is a family of functions $\mathsf{F}_\lambda : \mathbb{Z}_2^{m \times n} \times \mathbb{Z}_2^n \to \mathbb{Z}_3^t$ with key-space $\mathcal{K}_\lambda = \mathbb{Z}_2^{m \times n}$, input space $\mathcal{X}_\lambda = \mathbb{Z}_2^n$ and output space $\mathcal{Y}_\lambda = \mathbb{Z}_3^t$. For a key $\mathbf{K} \in \mathcal{K}_\lambda$, we define $\mathsf{F}_\mathbf{K}(x) = \mathsf{F}_\lambda(\mathbf{K}, x)$ as follows:

1. On input $x \in \mathbb{Z}_2^n$, first compute $w = \mathsf{BL}_2(\mathbf{K}, x) = \mathbf{K}x$.
2. Output $y = \mathsf{Lin}_3^\mathbf{B}\left(\mathsf{Convert}_{(2,3)}(w)\right)$. That is, view w as a vector over $\mathbb{Z}_3$ and then output $y = \mathbf{B}w$.



Construction 3.2 (Mod-2/Mod-3 OWF Candidate)**.** The OWF candidate is a function $\mathsf{F}_\lambda : \mathbb{Z}_2^n \to \mathbb{Z}_3^t$ with input space $\mathcal{X}_\lambda = \mathbb{Z}_2^n$ and output space $\mathcal{Y}_\lambda = \mathbb{Z}_3^t$. We define $\mathsf{F}(x) = \mathsf{F}_\lambda(x)$ as follows:

1. On input $x \in \mathbb{Z}_2^n$, first compute $w = \mathsf{Lin}_2^\mathbf{A}(x) = \mathbf{A}x$.
2. Output $y = \mathsf{Lin}_3^\mathbf{B}\left(\mathsf{Convert}_{(2,3)}(w)\right)$. That is, view w as a vector over $\mathbb{Z}_3$ and then output $y = \mathbf{B}w$.

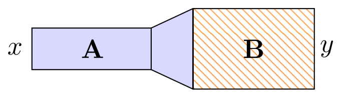


Figure 2: 23-constructions

4 Cryptanalysis

4.1 Choosing concrete parameters

5 Distributed Protocols

We now describe efficient protocols to compute our candidate constructions in several interesting distributed settings.

5.1 Technical Overview

Recall that all our constructions can be succinctly represented using four basic gates. The main strategy now will be to evaluate each of these gates in a distributed manner. These gate evaluation subprotocols can then be easily composed to evaluate the candidate constructions.

We begin with distributed protocols to evaluate each of the four gates. Abstractly, the goal of a gate protocol is to convert shares of the inputs to shares of the outputs (or shares of the masked output). To make our formalism cleaner, the gate protocols, by themselves, will involve no communication. Instead, they can additionally take in masked versions of the inputs, and possibly some additional correlated randomness. When composing gate protocols, whenever a masked input is needed, the parties will exchange their local shares to publicly reveal the masked value.

Notation. For a value v in a group $\mathbb{G}$, we use $\tilde{v}$ to denote a random mask value sampled from the same group, and $\hat{v} = v + \tilde{v}$ (where $+$ is the group operation for $\mathbb{G}$) to denote v masked by $\tilde{v}$. We use the $+$ operator quite liberally and unless specified, it denotes the group operation (e.g., component-wise addition mod p for $\mathbb{Z}_p^l$) for the summands.

For a value v, we use $\llbracket v \rrbracket$ to denote an (additive) sharing of v (in $\mathbb{G}$) among all protocol participants, and $\llbracket v \rrbracket^{(i)}$ to denote the share of the i^{th} party. When clear from context (e.g., a local protocol for P_i), we will often drop the superscript.

For a protocol $\boldsymbol{\pi}$, we use the notation $\boldsymbol{\pi}(a_1, \ldots, a_k \mid b_1, \ldots, b_l)$ to denote that the values $a_1, \ldots, a_k$ are provided publicly to all parties in the protocol, while the values $b_1, \ldots, b_l$ are secret shared among the parties.

Local share operations. Given public values $a_1, \ldots, a_k$, it is straightforward for the protocol parties to compute a sharing $\llbracket f(a_1, \ldots, a_k) \rrbracket$ for a function f (for example, P_1 computes the function as its share, and all other parties set their share to 0).

5.1.1 Distributed Computation of Circuit Gates

We provide detailed (local) protocols to compute each circuit gate in this section. The description of each gate protocol is also summarized in Table 1.

Linear gate protocol $\boldsymbol{\pi}_{\mathsf{Lin}}^{\mathbf{A},p}$. The linear gate is the easiest to evaluate. Here, each party is provided with the matrix $\mathbf{A}$, and a share of the input x (over $\mathbb{Z}_p$). The goal is to compute shares of the output $y = \mathbf{A}x$. For the protocol $\boldsymbol{\pi}_{\mathsf{Lin}}^{\mathbf{A},p}(\mathbf{A} \mid x)$, each party P_i computes its share $\llbracket y \rrbracket = \mathbf{A} \llbracket x \rrbracket$. Note that $\mathbf{A}x = \sum_{\mathcal{P}} \mathbf{A} \llbracket x \rrbracket$.

[Mahimna: Is this sufficiently clear or should we use something like $\llbracket x \rrbracket^{(i)}$ everywhere?]

Protocol	Public Inputs	Shared Inputs	Output Shares (over base group $\mathbb{G}$)
$\boldsymbol{\pi}_{\mathsf{Lin}}^{\mathbf{A},p}$	$\mathbf{A}$	x	$y = \mathbf{A}x$
$\boldsymbol{\pi}_{\mathsf{BL}}^{p}$	$\hat{\mathbf{K}}, \hat{x}$	$\tilde{\mathbf{K}}, \tilde{x}, \tilde{\mathbf{K}}\tilde{x} + \tilde{y}$	$\hat{y} = \mathbf{K}x + \tilde{y}$
$\boldsymbol{\pi}_{\mathsf{Convert}}^{(2,3)}$	$\hat{x}$ (over $\mathbb{Z}_2$)	$r = \tilde{x}$ (over $\mathbb{Z}_3$)	$x^* = x$ (over $\mathbb{Z}_3$)
$\boldsymbol{\pi}_{\mathsf{Convert}}^{(3,2)}$	$\hat{x}$ (over $\mathbb{Z}_3$)	$u = \tilde{x} \bmod 2$ (over $\mathbb{Z}_2$) $v = (\tilde{x} + \mathbf{1} \bmod 3) \bmod 2$ (over $\mathbb{Z}_2$)	$x^* = x \bmod 2$ (over $\mathbb{Z}_2$)

Table 1: Summary of circuit gate protocols

Bilinear gate protocol $\boldsymbol{\pi}_{\mathsf{BL}}^{p}$. For the bilinear gate, given shares of $\mathbf{K}$ and x (over $\mathbb{Z}_p$), the goal is to compute shares of the masked value $\hat{y} = \mathbf{K}x + \tilde{y}$. In the protocol, the values $\hat{\mathbf{K}}, \hat{x}$ will be publicly provided to all parties. Further, each party will also be given shares of $\tilde{\mathbf{K}}, \tilde{x}$ and $\tilde{\mathbf{K}}\tilde{x} + \tilde{y}$. Now, the evaluate $\boldsymbol{\pi}_{\mathsf{BL}}^{p}(\hat{\mathbf{K}}, \hat{x} \mid \tilde{\mathbf{K}}, \tilde{x}, \tilde{\mathbf{K}}\tilde{x} + \tilde{y})$, each party P_i computes its share of $\hat{y}$ as:

$$\llbracket \hat{y} \rrbracket = \left\llbracket \hat{\mathbf{K}}\hat{x} \right\rrbracket - \hat{\mathbf{K}} \llbracket \tilde{x} \rrbracket - \left\llbracket \tilde{\mathbf{K}} \right\rrbracket \hat{x} + \left\llbracket \tilde{\mathbf{K}}\tilde{x} + \tilde{y} \right\rrbracket$$

Note that this works since:

$$\begin{aligned}\sum_{\mathcal{P}} \llbracket \hat{y} \rrbracket &= \hat{\mathbf{K}}\hat{x} - \hat{\mathbf{K}}\tilde{x} - \tilde{\mathbf{K}}\hat{x} + \tilde{\mathbf{K}}\tilde{x} + \tilde{y} \\ &= (\mathbf{K} + \tilde{\mathbf{K}})x - \tilde{\mathbf{K}}(x + \tilde{x}) + (\tilde{\mathbf{K}}\tilde{x} + \tilde{y}) \\ &= \mathbf{K}x + \tilde{y}\end{aligned}$$

$\mathbb{Z}_2 \to \mathbb{Z}_3$ conversion protocol $\boldsymbol{\pi}_{\mathsf{Convert}}^{(2,3)}$. For the $\mathbb{Z}_2$ to $\mathbb{Z}_3$ conversion gate, given a masked input $\hat{x} = x \oplus \tilde{x}$ (i.e., over $\mathbb{Z}_2$) publicly, the goal is to output shares of $x^* = x$ over $\mathbb{Z}_3$. For this, the parties will also be given shares of $r = \tilde{x}$ over $\mathbb{Z}_3$. Note that since $\hat{x}$ is given publicly, it can also be thought to be over $\mathbb{Z}_3$. Now, to evaluate $\boldsymbol{\pi}_{\mathsf{Convert}}^{(2,3)}(\hat{x} \mid r)$, each party proceeds as follows:

$$\llbracket x^* \rrbracket = \llbracket \hat{x} \rrbracket + \llbracket r \rrbracket + (\hat{x} \odot \llbracket r \rrbracket) \quad \bmod 3$$

where $\odot$ denotes the Hammard (component-wise) product modulo 3. Here, addition is also done over $\mathbb{Z}_3$.

To see why this works, suppose that $\hat{x} \in \mathbb{Z}_2^l$. Consider any position $j \in [l]$, and denote by using a subscript j, the j^{th} position in a vector. Note that now, the position j of the output can be written as:

$$\llbracket x^* \rrbracket_j = \llbracket \hat{x} \rrbracket_j + \llbracket r \rrbracket_j + (\hat{x} \llbracket r \rrbracket_j \bmod 3) \quad \bmod 3$$

Consider two cases:

- If $\hat{x}_j = 0$, then $\tilde{x}_j = x_j$. Therefore, $\sum_{\mathcal{P}} \llbracket x^* \rrbracket_j = 0 + \tilde{x}_j = x_j$.
- If $\hat{x}_j = 1$, then $x_j = 1 - \tilde{x}_j$. Therefore, $\sum_{\mathcal{P}} \llbracket x^* \rrbracket_j = 1 + 2\tilde{x}_j \bmod 3$. If $\tilde{x}_j = 0$, this evaluates to $1 = x_j$, while if $\tilde{x}_j = 1$, it evaluates to $0 = 1 - \tilde{x}_j = x_j$

In other words, in all cases, each component of the sum (mod 3) of shares $\llbracket x^* \rrbracket$ is the same as the corresponding component of x. Therefore, $\sum_{\mathcal{P}} \llbracket x^* \rrbracket \pmod 3 = x$ will hold.

$\mathbb{Z}_3 \to \mathbb{Z}_2$ conversion protocol $\pi^{(3,2)}_{\mathsf{Convert}}$. For the $\mathbb{Z}_3$ to $\mathbb{Z}_2$ conversion gate, given a masked input $\hat{x} \in \mathbb{Z}_3^l; \hat{x} = x + \tilde{x} \bmod 3$, the goal is to output shares of x^* (over $\mathbb{Z}_2$) where $x^* = x \bmod 2$. For this, each party is also given shares (over $\mathbb{Z}_2$) of two vectors: $u = \tilde{x} \bmod 2$ and $v = (\tilde{x} + \mathbf{1} \bmod 3) \bmod 2$. Now, each party computes its share of x^* as follows: For each position $j \in [l]$,

$$[\![x^*]\!]_j = \begin{cases} 1 - [\![u]\!]_j - [\![v]\!]_j & \text{if } \hat{x}_j = 0 \\ [\![v]\!]_j & \text{if } \hat{x}_j = 1 \\ [\![u]\!]_j & \text{if } \hat{x}_j = 2 \end{cases}$$

To see why this works, consider three cases:

- If $\hat{x}_j = 0$, then $\sum_{\mathcal{P}} [\![x^*]\!]_j \bmod 2 = 1 - u_j - v_j$. This evaluates to 1 only when $\tilde{x}_j = 2$, and is exactly the case when x_j is also 1.
- $\hat{x}_j = 1$, then $\sum_{\mathcal{P}} [\![x^*]\!]_j \bmod 2 = v_j = (\tilde{x}_j + 1 \bmod 3) \bmod 2)$. This evaluates to 1 only when $\tilde{x}_j = 0$, and is exactly the case when x_j is also 1.
- $\hat{x}_j = 2$, then $\sum_{\mathcal{P}} [\![x^*]\!]_j \bmod 2 = u_j$. This evaluates to 1 only when $\tilde{x}_j = 1$, and is exactly the case when x_j is also 1.

Consequently, $\sum_{\mathcal{P}} [\![x^*]\!] \bmod 2 = x \bmod 2$ holds.

5.2 Distributed Evaluation in the Preprocessing Model

Equipped with our technical overview, we proceed to describe our distributed protocols in the preprocessing model. We focus primarily on the 2-party semi-honest setting but comment that our protocols can easily be generalized to n parties.

5.2.1 2-party wPRF evaluation.

We start with a 2-party distributed protocol to evaluate our primary mod-2/mod-3 wPRF candidate from Construction 3.1. In this setting, the two parties hold additive shares of a key $\mathbf{K} \in \mathbb{Z}_2^{m \times n}$, and an input $x \in \mathbb{Z}_2^n$. The goal is to compute an additive sharing of the wPRF output $y = \mathsf{LinMap}_{\mathbf{G}}(\mathbf{K}x)$ where $\mathbf{G} \in \mathbb{Z}_3^{t \times m}$ is a publicly known matrix.

[Mahimna: Work in progress]

n-party distributed evaluation.

5.3 2-party Public Input Evaluation

5.4 3-party Distributed Evaluation

5.5 Oblivious PRF Evaluation

6 Implementation and Evaluation

We implemented our 2-party 23-wPRF (Construction 3.1) and 23-OPRF (Construction 3.2) in C++. For the constructions, we used the parameters $n = m = 256$ and $t = 81$. In other words, the implemented 23-constructions use a circulant matrix in $\mathbb{Z}_2^{256\times 256}$ as the key, take as input a vector in $\mathbb{Z}_2^{256}$ and output a vector in $\mathbb{Z}_3^{81}$. The correlated randomness was implemented as if provided by a trusted third party. See Section **??** for concretely efficient protocols for securely generating the correlated randomness, which we did not implement but give efficiency estimates based on prior works. [Yuval: Add pointer when section is written.]

6.1 Optimizations

We start with a centralized implementation of the 23-wPRF. We find optimizations that provide 5x better performance over a naïve implementation, which we detail below. Later, in Table 3, we compare the performance gain from these optimizations.

6.2 Representing Z_2 vector - bit packing

The dark matter function manipulates elements of Z_2 and Z_3. However, machine words can hold up to 64 bits and we use this to represent many elements in just a few machine words, using bit slicing. Namely, we used each machine word as a vector of 64 bits and applied operations to this bit-vector in a SIMD manner. Since in our implementation each key $\mathbf{K}$ is of size $m \times n = 256 \times 256$ and each input is a vector of size $n = 256$, this results in packing each key into $\dfrac{256}{64} = 4$ words and each input vector into 4 words. This may result in time saving of up to $\times 64$ of the run-time.

6.3 Representing Z_3 vector - bit slicing

The last part of the dark matter function takes an intermediate vector viewed as a vector of Z_3 elements, which is then multiplied by a matrix R in Z_3 to produce the output vector in Z_3. We tried different methods for implementing this matrix-vector multiplication over Z_3.:

Bit slicing is implemented by representing a vector over Z_3 as two binary vectors - one LSB's and one MSB's. The operations over this Z_3 vector - addition, subtraction, multiplication and MUX - were implemented in a bitwise manner as shown in table 2. We took advantage of the fact that when representing each vector as MSB and LSB, negation (which is the same as multiplying by two) is the same as swapping the most and least significant bits.

Lookup table for matrix multiplication. Even with bitslicing, implementing the matrix-vector multiplication column by column via the opertions from Table 1 is still rather slow. To get better performence, we capitalized on the fact that the random matrix R is fixed, and we can therefore set up R-dependent tables and use table lookups in the implementation of the multiply-by-R operation.

Specifically, we partition the matrix R (which has $m = 256$ columns) into sixteen slices of 16 columns each, denoted $R_1, \ldots, R_{16}$. For each of these small matrices R_i, we then build a table with 2^{16} entries, holding the result of multiplying R_i by every vector from $\{0,1\}^{16}$. Namely, let $R_{i,0}, R_{i,1}, \ldots, R_{i,15}$ be the sixteen columns of the matrix R_i. The table for R_i is then defined as

Table 2: Operations in Z_3. input: l_1, m_1 - LSB and MSB of first trinary number, l_2, m_2 - LSB and MSB of second trinary number, s - selection bit for MUX

Operations	**Methods**
Addition	$t := (l_1 \wedge m_2) \oplus (l_2 \wedge m_1)$ $m_{\text{out}} := (l_1 \wedge l_2) \oplus t$ $l_{\text{out}} := m_1 \wedge m_2) \oplus t$
Subtraction	$t := (l_1 \wedge l_2) \oplus (m_2 \wedge m_1);$ $m_{\text{out}} := (l_1 \wedge m_2) \oplus t;$ $l_{\text{out}} := (m_1 \wedge l_2) \oplus t;$
Multiplication	$m_{\text{out}} := (l_1 \vee m_2) \wedge (m_1 \vee l_2);$ $l_{\text{out}} := (l_1 \vee l_2) \wedge (m_1 \vee m_2);$
MUX	$m_{\text{out}} := (m_2 \vee s) \wedge (m_1 \vee (\bar{s}));$ $l_{\text{out}}[i] := (l_2 \vee s) \wedge (l_1 \vee (\bar{s}));$

follows: For each index $0 \leq j < 2^{16}$, let $\vec{J} = (j_0, j_1, \ldots, j_{15})$ be the 0-1 vector holding the bits in the binary representation of j, then we have

$$T_i[j] = R_i \times \vec{J} = \sum_{k=0}^{15} R_{i,k} \cdot j_k \mod 3.$$

Recalling that R is Z_3 matrix of dimension 81×256, every entry in each table T_i therefore holds an 81-vector over Z_3. Specifically, it holds a packed-Z_3 element with four words (two for the MSBs and two for the LSBs of this 81-element vector).

Note, however, that T_i can only be used directly to multiply R_i by *0-1 vectors*. To use T_i when multiplying R_i by a $\{0,1,2\}$ vector, multiply R_i separately by the MBSs and the LSBs of that vector, and then subtract one from the other using the operations from Table 1. To multiply a dimension-256 $\{0,1,2\}$ vector by the matrix R, we partition it into sixteen vectors of dimension 16, use the approach above to multiply each one of them by the corresponding R_i, then use the operations from Table 1 to add them all together. Hence we have

```
Multiply-by-$R$(input: $\vec{V} \in \{0,1,2\}^{256}$:
 1. $Acc := 0$
 2. For $i = 0, \ldots, 15$
 3.     Let $\vec{M}_i, \vec{L}_i \in \{0,1\}^{16}$ be the MSB, LSB vectors (resp.) of $\vec{V}_i = \vec{V}[16i, \ldots, 16i+15]$
 4.     Set the indexes $m := \sum_{k=0}^{15} 2^i \cdot \vec{M}[i]$ and $\ell := \sum_{k=0}^{15} 2^i \cdot \vec{L}[i]$
 5.     $Acc := Acc + T_i[\ell] - T_i[\ell] \bmod 3$
 6. Output $Acc$
```

We note that a slightly faster implementation could be obtained by braking the matrix into (say) 26 slices of upto 10 columns each, and directly identify each 10-vector over $\{0,1,2\}$ with an index $i < 3^{10} = 59049$. This implementation would have only 26 lookup operations instead of 32 in the algorithm above, so we expect it to be about 20% faster. On the other hand it would have almost twice the table size of the implementation from above.

6.4 Performance Benchmarks

Experimental setup We ran all our experiments on a t2.medium AWS EC2 instance with 4GiB RAM (architecture: x86-64 Intel(R) Xeon(R) CPU E5-2686 v4 @ 2.30GHz) running on Ubuntu 18.04. The performance benchmarks and timing results we provide are averaged over 1000 runs. For the distributed construction benchmarks, both parties were run on the same instance. We separately report the computational runtime for the parties, and provide an estimate of the communication costs.

Communication vs. Computation: from the table we can estimate the break even point for each protocol. Note that synchronization is needed between the parties, though, so the assumption this is independent is an estimate. For the wPRF protocol **??** presented in this paper, the break even point is achieved at a network speed 63 Mbps, in which case the overall communication and computation times will be the same. For the oPRF protocol, the break even point will be achieved with a network of speed 146 Mbps. From here we see that for the oPRF protocol, for an existing 25 Mbps connection, communication will dominate the total run-time, while for a high speed network of 1 Gbps, the computation time will dominate the overall run-time.

[Yuval: Communication costs in bits can be computed analytically. Maybe talk about estimated breakeven points: (1) minimal network speed in which computation dominates communication, and (2) minimal network speed where we estimate our OPRF to be faster than the DDH-based alternative.]

Optimization results. We start with benchmarks for the different optimization techniques detailed in Section 6.1. Table 3 contains timing results for our centralized implementation of the 23-wPRF construction using these optimizations.

Optimization			Runtime(μ sec)	Evaluations / sec
Packing	Bit Slicing	Lookup Table		
✓			26.84	37K
✓	✓		18.5	65K
✓	✓	✓	6.08	165K

Table 3: Centralized 23-wPRF benchmarks using different optimization techniques. Packing was done into 64-bit sized words (for both $\mathbb{Z}_2$ and $\mathbb{Z}_3$ vectors). For the lookup table optimization, an ($m = 256$)-column matrix was partitioned into 16 slices of 16 columns each. A lookup table containing 81×2^{20} elements in $\mathbb{Z}_3$[Yuval: Specify units in Runtime column. Memory Usage column: not sure what this means, and in any case the first two rows seem too small to be meaningful (e.g., the amount of memory used by the program for maintaining variables may be bigger). I suggest removing this column, and instead specifying the lookup table size in the caption. Alternatively, we can keep this column and change the title to "lookup table size", keeping the first two rows empty. Regarding this size, what does 2^{20} count? Bits? Machine words? $\mathbb{Z}_3$ elements?]

[Mahimna: It would be nice to get benchmarks for performance gain using each optimization independently. If possible, it might also be nice to get the numbers for different sized lookup tables. We could also estimate this analytically.]

Distributed wPRF evaluation. We consider two 2-party weak PRF protocols. The first protocol is described in [BIP+18] and the second one is our construction as described in **??**. Since the goal of the implementation was timing the local *online*run-time, the implementation used a trusted party to generate the pre-processed variables.

Protocol	**Eval/sec**	**Runtime(μ sec)**
Fully Distributed wPRF [BIP+18]	36K	28.02
Fully Distributed wPRF	82K	12.12

Table 4: Optimized protocols runtime. These protocols utilize both the bit-packing and lookup table optimization

OPRF evaluation. In Table 5, we provide performance benchmarks for our 23-OPRF construction while also comparing it with other constructions. For our timing results, we report both the server and client runtimes (averages over 1000 runs). For each construction, we also include the size of the preprocessed correlated randomness, and the online communication cost. All constructions are parameterized appropriately to provide 128-bit security.

For the comparison with the DDH-based OPRF construction in [NPR99], we use the libsodium library [Lib] for the elliptic curve scalar multiplication operation. We use the Curve25519 elliptic curve, which has a 256-bit key size, and provides 128 bits of security. [Yuval: I actually think that the Naor-Pinkas-Reingold99 paper cited in the TCC '18 paper refers to a different notion of distributing a PRF, namely where the PRF key is distributed between two or more servers and the input is public. Let's try to figure out the source of the simple DDH-based OPRF protocol. Can ask Stas/Hugo if needed.]

Protocol		Runtime		Preprocessing	Communication(bits)	
		Server	Client		Server	Client
23-OPRF	Key Update	0.65	-	1835	593	512
	Evaluation	9.45	9.13			
DDH-based OPRF		57.38	28.69	-	256	256

Table 5: Comparison of protocols for (semi-honest) OPRF evaluation in the preprocessing model. Runtimes are given in microseconds (μs). [Yuval: What does "parallel" mean? Should be explained in the caption. Our output size has roughly 128 bits (81 is the number of $\mathbb{Z}_3$ elements), but it is actually meaningless because the output size can be easily extended or decreased. So let's drop this column. For preprocessing and communication, I suggest using concrete numbers as opposed to general expressions that refer to different values of n.]
[Mahimna: Include more comparisons as necessary]

7 Applications

7.1 A Signature Scheme

Here we describe a signature scheme using the $(2,3)$-OWF. Abstractly, a signature scheme can be built from any OWF that has an MPC protocol to evaluatue it, by setting the public key to $y = F(x)$ for a random secret x, and then proving knowledge of x, using a proof system based on the MPC-in-the-head paradigm [**STOC:IKOS07**]. In addition to assuming the OWF is secure, the only other assumption require is a secure hash function. As no additional number theoretic assumptions are required, these type of signatures are often proposed as secure post-quantum schemes.

Concretely, our design follows the Picnic signature scheme [**CCS:CDGORR17**], specifically the variant instantiated with the KKW proof system [**CCS:KatKolWan18**] (named Picnic2 and Picnic3). We chose to use the KKW, rather than ZKB++ proof system since our MPC protocol to evaluate the $(2,3)$-OWF is most efficient with a pre-processing phase, and KKW generally produces shorter signatures. We replace the OWF in Picnic, the LowMC block cipher [**EC:ARSTZ15**], with the $(2,3)$-OWF, and make the corresponding changes to the MPC protocol.

[Greg: Need some motivation here for why this is interesting. Rough ideas:

- No existing signature scheme based on the $(2,3)$-OWF or similar assumption.
- Alternative structure of the $(2,3)$-OWF may be easier to analyze than LowMC, which follows a more traditional block cipher design.
- The $(2,3)$-OWF is conceptually much simpler requires far less precomputed constants, and has a simpler implementation.
- Potential advantages in implementation performance

]

$(2,3)$-OWF Description Recall that the $(2,3)$-OWF is defined as $y = F(x)$ where $x \in \mathbb{Z}_2^n$ and $y \in \mathbb{Z}_3^t$, and is computed as follows:

1. Compute $w = \mathbf{A}x \in \mathbb{Z}_2^m$, where $\mathbf{A}$ is a public, randomly chosen matrix of full rank in $\mathbb{Z}_2^{m\times n}$
2. Let $z \in \mathbb{Z}_3^m$ be w, where entries are intepreted as values mod 3
3. Compute and output $y = \mathbf{B}z$, where $\mathbf{B} \in \mathbb{Z}_3^{m\times t}$ is public and randomly chosen as $\mathbf{A}$ was.

An N-party protocol There will be N parties for our MPC protocol, each holding a secret share of x, who jointly compute $y = F(x)$. The protocol is N-private, meaning that up to $N-1$ parties may be malicious, and the secret input remains private. Put another way, given the views of $N-1$ parties we can simulate the remaining party's view, to prove that the $N-1$ parties have no information about the remaining party's share.

The preprocessing phase is similar to that in Picnic. Each party has a random tape that they can use to sample a secret sharing of a uniformly random value (e.g., a scalar, vector, or a matrix with terms in $\mathbb{Z}_2$ or $\mathbb{Z}_3$). Each party samples their share $[r]$ and the shared value is implicitly defined as $r = \sum_{i=1}^{N} [r]_i$. We use $[r]_i$ to denote party i's share of r, or simply $[r]$ when the context makes the party's index clear.

We must also be able to create a sharing mod 3, of a secret shared value mod 2. Let $w' \in \mathbb{Z}_2$ be secret shared. Then to establish shares of $r = w' \pmod 3$, the first $N-1$ parties sample a share $[r]$ from their random tapes. The N-th party's share is chosen by the prover, so that the sum of the shares is r. We refer to the last party's share as an *auxiliary value*, since it's provided by the prover as part of pre-processing. For efficiency, the random tape for party i is generated by a random seed, denoted seed_i, using a PRG. The state of the first $N-1$ parties after pre-processing is a seed value used to generate the random tape, and for the N-th party the state is the seed value plus the list of auxiliary values, denoted aux.

After pre-processing, the parties enter the *online* phase of the protocol. The prover computes $\hat{x} = x + x'$, where x' is a random value, established during preprocessing so that each party has a share $[x']$. The parties can then compute the OWF using the homomorphic properties of the secret sharing, and a share conversion gadget (to convert shares mod 2 to mod 3, used when computing z) setup during preprocessing that we describe below. During the online phase, parties broadcast values to all other parties and we write msgs_i to denote the broadcast messages of party i.

7.1.1 MPC protocol details

Preprocessing phase Preprocessing establishes random seeds of all parties and shares of

1. x': a random vector in $\mathbb{Z}_2^n$, //Sampled from random tapes
2. w': a random vector in $\mathbb{Z}_2^m$, //Sampled from random tapes
3. r: a sharing of w' mod 3, shares in $\mathbb{Z}_3^m$, //Tapes + one aux value
4. $\bar{r}$: a sharing of $1 - w'$ mod 3, shares in $\mathbb{Z}_3^m$. //Computed from shares of r

The shares of $\bar{r}$ are computed from shares of r as follows (all arithmetic in $\mathbb{Z}_3^m$): the first party computes $[\bar{r}] = 1 - [r]$, then the remaining parties compute $[\bar{r}] = -[r]$. Then observe that

$$\sum_{i=1}^{N} [\bar{r}]_i = 1 - [r]_1 - \ldots - [r]_N = 1 - \sum_{i=1}^{N} [r]_i = 1 - r$$

as required.

Online phase The public input to the online phase is $\hat{x} = x + x'$.

1. The parties locally compute $[u] = \mathbf{A}[x'] - [w']$ and broadcast it, then sum the received values to get $u = \mathbf{A}x' - w'$. Then they locally compute $\hat{w} = \mathbf{A}\hat{x} - u = \mathbf{A}x + w'$, where $\hat{w} \in \mathbb{Z}_2^m$. Each party broadcasts m bits in this step.
2. Let z be a vector in $\mathbb{Z}_3^m$ and let z_i denote the i-th component. Each party defines

$$[z_i] = \begin{cases} [r_i] & \text{if } \hat{w}_i = 0 \qquad \text{//Note that } [r_i] = [w_i'] \\ [\bar{r}_i] & \text{if } \hat{w}_i = 1 \qquad \text{//Note that } [\bar{r}_i] = [1 - w_i'] \end{cases}$$

then localy computes $[y] = \mathbf{B}[z]$. All parties broadcast $[y]$ and reconstruct the output $y \in \mathbb{Z}_3^t$. In this step each party broadcasts t values in $\mathbb{Z}_3$.

Correctness The protocol correctly computes the $(2,3)$-OWF. The first step computes $w = \mathbf{A}x$, while keeping it masked with a random w', updating the public value $\hat{x} = x + x'$ with $\hat{w} = w + w'$. The second step is where the bits of w are cast from $\mathbb{Z}_2$ to $\mathbb{Z}_3$. The parties have sharings of w' and $1 - w' \mod 3$. Now, the key observation is that when $\hat{w} = 0$, then w and w' are the same, and when $\hat{w} = 1$, w and w' are different. So in the first case we set the shares of $z = w \mod 3$ to the shares of $[w'] \mod 3$, and when $\hat{w} = 1$, we set the shares of z to the complement of w'.

Communication Costs Here we quantify the cost of communication for the aux values and the broadcast msgs of one party, as this will directly contribute to the signature size in the following section. Let ℓ_3 be the bitlength of an element in $\mathbb{Z}_3$; the direct encoding has $\ell_3 = 2$, but with compression we can reudce ℓ_3 to as little as $\log_2(3) \approx 1.58$. [Greg: TODO: crossref. I'm assuming the details of this will be somewhere in the paper? I don't know them :)] The size of the aux information is $m\ell_3$, the MPC input value has size n bits, and the broadcast values have size $m + t\ell_3$ bits (per party). The total in bits is thus

$$|\mathsf{MPC}(n,m,t)| = m(\ell_3 + 1) + n + t\ell_3 \tag{1}$$

which is $2.58m + n + 1.58t$ when $\ell_3 = 1.58$. For the parameters $(n,m,t) = (256, 256, 81)$ the total is 1045 bits for $(n,m,t) = (256, 256, 160)$ the total is 1170 bits, and for $(n,m,t) = (128, 400, 81)$ the total is 1288 bits. This compares favorably to Picnic at the same security level, which communicates 1032 bits for the aux and msgs when LowMC uses a full S-box layer, and 1200 bits when LowMC uses a partial S-box layer [**TCHES:KalZav20**].

7.1.2 Signature Scheme Details

Given the MPC protocol above, we can compute the values $\hat{x}$, aux and msgs for the $(2,3)$-OWF and neatly drop it into the KKW proof system used in Picnic. The signature generation and verification algorithms for the $(2,3)$-OWF signature scheme are given in Fig. 3.

Parameters Let κ be a security parameter. The $(2,3)$-OWF parameters are denoted (n,m,t). The KKW parameters (N, M, τ) denote the number of parties N, the total number of MPC instances M, and the number τ of MPC instances where the verifier checks the online phase of simulation. We use a cryptographic hash function $\mathsf{H} : \{0,1\}^* \to \{0,1\}^{2\kappa}$ for computing commitments, and the function Expand takes as input a random 2κ-bit string and derives a challenge having the form $(\mathcal{C}, \mathcal{P})$ where $\mathcal{C}$ is a subset of $[M]$ of size τ, and $\mathcal{P}$ is a list of length τ, with entries in $[N]$. The challenge $(\mathcal{C}, \mathcal{P})$ defines τ pairs (c, p_c) where c is the index of an MPC instance for which the verifier will check the online phase, and p_c is the index of the party that will remain unopened.

Key generation The signer chooses a random $x \in \mathbb{Z}_2^n$ as a secret key, and a random seed $s_A \in \{0,1\}^\kappa$ such that s_A expands to a matrix $\mathbf{A} \in \mathbb{Z}_2^{m \times n}$ that is full rank (using a suitable cryptographic function, such as the SHAKE extendable output function [KCP16]). We use a unique $\mathbf{A}$ per signer in order to avoid multi-target attacks against F. Compute $y = F(x)$ and set (y, s_A) as the public key. [Greg: We could generate the public matrix $\mathbf{B}$ from s_A as well, or we could fix it once for all signers. Is there a security reason to go with per-signer $\mathbf{B}$? Question for Itai.]

Signature generation and verification Here we give an overview of signature generation, for details see Fig. 3. The structure of the signature follows a three-move Σ-protocol. In the commit phase, the prover simulates the preprocessing phase for all M MPC instances, and commits to the output (the seeds and auxiliary values). She then simulates the online phase for all M instances and commits to the inputs, and broadcast messages. Then a challenge is computed by hashing all commitments, together with the message to be signed. The challenge selects τ of the M MPC instances. The verifier will check the simulation of the online phase for these instances, by recomputing all values as the prover did for $N-1$ of the parties, and for remaining unopened party, the prover will provide msgs and a commitment to the seed so that the verifier may complete the simulation and recompute all commitments. For the $M-\tau$ instances not chosen by the challege, the verifier will check the preprocessing phase only – here the prover provides the random seeds of all N parties and the verifier can recompute the prover's commitment to the preprocessing, in particular the verifier checks that the auxiliary values are correct.

Optimizations and simplifications For ease of presentation, Fig. 3 omits some optimizations that are essential for efficiency, but are not unique to the $(2,3)$-signature schemes, they are exactly as in Picnic. All random seeds in a signature are derived from a single random root seed, using a binary tree construction. First we derive M initial seeds, once for each MPC instance, then from from the initial seed we derive the N per-party seeds. This allows the signer to reveal the seeds of $N-1$ parties by revealing only $\log_2(N)$ intermediate seeds, similarly, the initial seeds for $M-\tau$ of M instances may be revealed by communicating only $(\tau)\log_2(M/\tau)$ κ-bit seeds.

For the commitments $h'^{(k)}$ to the online execution, τ are recomputed by the verifier, and the prover provides the missing $M-\tau$. Here we compute the $h'^{(k)}$ as the leaves of a Merkle tree, so that the prover can provide the missing commitments by sending only $\tau\log_2(M/\tau)$ 2κ-bit digests.

Finally, we omit a random salt, included in each signature, as well as counter inputs to the hash functions to prevent multi-target attacks [**EC:DinNad19**]. Also, hashing the public key when computing the challenge, and prefixing the inputs to H in each use for domain separation should also be done, as in [Tea20].

Parameter selection and signature size The size of the signature in bits is:

$$\underbrace{\kappa\tau\log_2\left(\frac{M}{\tau}\right)}_{\text{initial seeds}} + \underbrace{2\kappa\tau\log_2\left(\frac{M}{\tau}\right)}_{\text{Merkle tree commitments}} + \tau\left(\underbrace{\kappa\log_2 N}_{\text{per-party seeds}} + \underbrace{|\mathsf{MPC}(n,m,t)|}_{\text{one MPC instance, Eq. (1)}}\right)$$

and we note that the direct contribution of OWF choice is limited to $|\mathsf{MPC}(n,m,t)|$[1]. However, the size of this term can impact the choice of (N,M,τ). The Picnic parameters (N,M,τ) must be chosen so that the soundness error,

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

[1]The size $|\mathsf{MPC}(n,m,t)|$ is a slight overestimate since for $1/N$ instances we don't have to send aux, if the last party is unopened. In Table 6 our estimates include this, but it's a very small difference as τ is quite small.

OWF Params (n, m, t)	KKW params (N, M, τ)	Signature size (KB)
$(128, 400, 81)$	$(16, 150, 51)$	15.06
	$(16, 168, 45)$	14.03
	$(16, 218, 38)$	12.99
	$(16, 250, 36)$	12.78
	$(16, 303, 34)$	12.66
	$(16, 352, 33)$	12.70
Picnic3-L1	$(16, 250, 36)$	12.60
$(128, 400, 81)$	$(64, 149, 46)$	15.54
	$(64, 209, 34)$	13.00
	$(64, 246, 31)$	12.41
	$(64, 343, 27)$	11.69
Picnic2-L1	$(64, 343, 27)$	12.36

Table 6: Signature sizes for Picnic using the $(2, 3)$-OWF, compared to Picnic using LowMC.

is less than $2^{-\kappa}$. There are two time/size tradeoffs, both exponential (i.e., a small decrease in size costs a large increase in time). The first is increasing N, which reduces τ slightly, but requires a large amount of computation to implement the MPC simulation for all parties (in particular the hashing required to derive seeds, compute commitments and compute random tapes is the most expensive, and independent of the OWF). So we fix $N = 16$. Next we can reduce τ, but only by increasing M significantly, intuitively, when there are fewer only instances checked by the verifier, we need greater assurance in each one, and so we must audit more preprocessing instances.

By searching the parameter space for fixed N and various options for M, τ, we get a curve, and choose from the combinations in the "sweet spot", near the bend of the curve with moderate computation costs. Table 6 gives some options. [Greg: TODO: highlight one row as being the one we'd choose]

Random notes [Greg: These should be worked into the text as appropriate]

- OWF vs. PRF: for the OWF I don't think there is any performance advantage to using compressed matrices (circulant matrices). So better to use fully random ones.
- The parameters above need review. I'm pretty sure $(128, 400, 81)$ is supported by the analysis in Itai's document but the others may not be.
- Optionally we may compare the NIST L5 level (AES-256 equivalent, i.e., 256-bit classical, 128-bit quantum secure), in case the security of the (2,3)-OWF scales better than LowMC.

7.2 Distributed Picnic

[Greg: Still no good ideas on how to do this]

$(2,3)$-OWF Signatures

Inputs Both signer and verifier have F, $y = F(x)$, the message to be signed Msg, and the signer has the secret key x. The parameters of the protocol (M, N, τ) are described in the text.

Commit For each MPC instance $k \in [M]$, the signer does the following.

1. Choose uniform $\mathsf{seed}^{(k)}$ and use to generate values $(\mathsf{seed}_i^{(k)})_{i \in [N]}$, and compute $\mathsf{aux}^{(k)}$ as described in the text. For $i = 1, \dots N-1$, let $\mathsf{state}_i^{(k)} = \mathsf{seed}_i^{(k)}$ and let $\mathsf{state}_N^{(k)} = \mathsf{seed}_N^{(k)} || \mathsf{aux}^{(k)}$.
2. Commit to the preprocessing phase:
$$\mathsf{com}_i^{(k)} = \mathsf{H}(\mathsf{state}_i^{(k)}) \text{ for all } i \in [N], \quad h^{(k)} = \mathsf{H}(\mathsf{com}_1^{(k)}, \dots, \mathsf{com}_N^{(k)}).$$
3. Compute MPC input $\hat{x}^{(k)} = x + x'^{(k)}$ based on the secret key x and the random values $x'^{(k)}$ defined by preprocessing.
4. Simulate the online phase of the MPC protocol, producing $(\mathsf{msgs}_i^{(k)})_{i \in [N]}$.
5. Commit to the online phase: $h'^{(k)} = \mathsf{H}(\hat{x}^{(k)}, \mathsf{msgs}_1^{(k)}, \dots, \mathsf{msgs}_N^{(k)})$.

Challenge The signer computes $\mathsf{ch} = \mathsf{H}(h_1, \dots h_M, h'_1, \dots, h'_M, \mathsf{Msg})$, then expands ch to the challenge $(\mathcal{C}, \mathcal{P}) := \mathsf{Expand}(\mathsf{ch})$, as described in the text.

Signature output The signature σ on Msg is

$$\sigma = (\mathsf{ch}, ((\mathsf{seed}^{(k)}, h^{(k)})_{k \notin \mathcal{C}}, (\mathsf{com}_{p_k}^{(k)}, (\mathsf{state}_i^{(k)})_{i \neq p_k}, \hat{x}^{(k)}, \mathsf{msgs}_{p_k}^{(k)})_{k \in \mathcal{C}})_{k \in [M]})$$

Verification The verifier parses σ as above, and does the following.

1. Check the preprocessing phase. For each $k \in [M]$:
   (a) If $k \in \mathcal{C}$: for all $i \in [N]$ such that $i \neq p_k$, the verifier uses $\mathsf{state}_i^{(k)}$ to compute $\mathsf{com}_i^{(k)}$ as the signer did, then computes $h'^{(k)} = \mathsf{H}(\mathsf{com}_1^{(k)}, \dots, \mathsf{com}_N^{(k)})$ using the value $\mathsf{com}_{p_k}^{(k)}$ from σ.
   (b) If $k \notin \mathcal{C}$: the verifier uses $\mathsf{seed}^{(k)}$ to compute $h'^{(k)}$ as the signer did.
2. Check the online phase:
   (a) For each $k \in \mathcal{C}$ the verifier simulates the online phase using $(\mathsf{state}_i^{(k)})_{i \neq p_k}$, masked witness $\hat{x}$ and $\mathsf{msgs}_{p_k}^{(k)}$ to compute $(\mathsf{msgs}_i)_{i \neq p_k}$. Then compute $h^{(k)}$ as the signer did. The verifier outputs 'invalid' if the output of the MPC simulation is not equal to y.
3. The verifier computes $\mathsf{ch}' = \mathsf{H}(h_1, \dots h_M, h'_1, \dots, h'_M, \mathsf{Msg})$ and outputs 'valid' if $\mathsf{ch}' = \mathsf{ch}$ and 'invalid' otherwise.

Figure 3: Picnic-like signature scheme using the $(2,3)$-OWF and the KKW proof sytem.

References

[BGI19] Elette Boyle, Niv Gilboa, and Yuval Ishai. “Secure Computation with Preprocessing via Function Secret Sharing”. In: *TCC*. 2019, pp. 341–371.

[BIP+18] Dan Boneh, Yuval Ishai, Alain Passelègue, Amit Sahai, and David J. Wu. “Exploring Crypto Dark Matter: New Simple PRF Candidates and Their Applications”. In: *TCC*. 2018, pp. 699–729.

[KCP16] J. Kelsey, S. J. Chang, and R. Perlner. *SHA-3 derived functions: cSHAKE KMAC TupleHash and ParallelHash*. National Institute for Standards and Technology, Special Publication 800-185. 2016. URL: `https://doi.org/10.6028/NIST.SP.800-185`.

[Lib] *libsodium 1.0.18-stable*. `https://libsodium.gitbook.io/doc/`. Online; December 31 2020. 2020.

[NPR99] Moni Naor, Benny Pinkas, and Omer Reingold. “Distributed pseudo-random functions and KDCs.” In: *EUROCRYPT*. 1999, pp. 327–346.

[Tea20] The Picnic Design Team. *The Picnic Signature Algorithm Specification*. Version 3.0, Available at `https://microsoft.github.io/Picnic/`. 2020.