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 a vector x, by $x \bmod p$, we mean that each element in x is taken modulo p. We use $x \xleftarrow{\$} \mathcal{X}$ to denote sampling uniformly at random from domain $\mathcal{X}$.

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, and $[\![x]\!]^{(i)}$ 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. 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 $\mathbb{G}$.

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.

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 all 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 [**boyle2019-fss-preprocess**]. 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 an FSS scheme [**boyle2015-fss**; **boyle2016-fss-extension**]. 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 [**boyle2019-fss-preprocess**] 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 [**boyle2019-fss-preprocess**].

Circuit gates. We make use of just four types of 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 [**boyle2019-fss-preprocess**] in that the input values of the circuit itself are not secret shared. The MPC protocols will instead take in secret shared inputs as necessary and evaluate the circuit in a distributed fashion. We denote by Gates, the set comprising of these gates.

- **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 $\mathsf{Convert}_{(2,3)}^l(\cdot)$ 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 $\mathsf{Convert}_{(2,3)}(x)$.

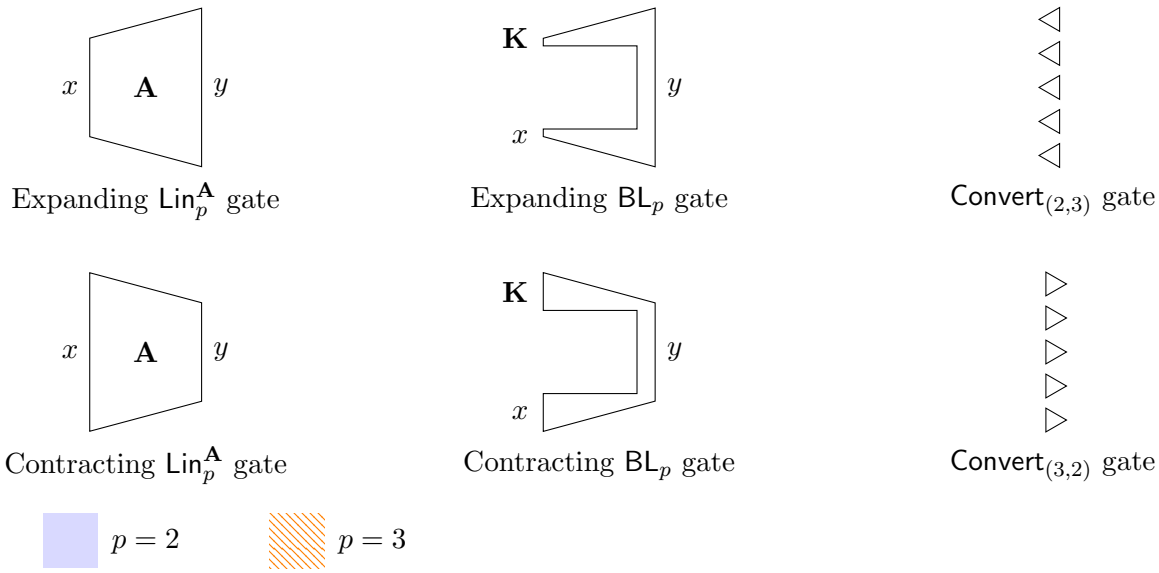


Figure 1: Pictorial representations of the circuit gates. For the linear and bilinear gates, expanding means that the length of the output vector is greater than or equal to the length of the input vector, while contracting means that the output vector is smaller than the input vector. Additionally, for $p = 2$, the gates are shaded in violet, and for $p = 3$, the gates contain diagonal orange lines. [Mahimna: We use $n = m$ for many constructions. Should we call that length preserving (although the general construction has $m \geq n$) or should be rename expanding to something like non-contracting?]

- $\mathbb{Z}_3 \to \mathbb{Z}_2$ **conversion.** For a positive integer l, the gate $\mathsf{Convert}^l_{(3,2)}(\cdot)$ 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 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 $\mathsf{Convert}_{(3,2)}(x)$.

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, both the key $\mathbf{K}$ and the input x will be secret shared. We call this gate *bilinear* because its output is linear in both of its inputs. Also note that although the $\mathsf{Convert}_{(2,3)}$ 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 (bi)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 $\mathbb{Z}_q$. 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 $(3, 2)$.

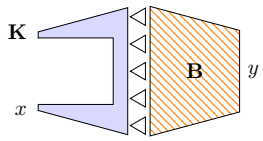
- **LPN-style-constructions.** [Mahimna: todo]
- [Mahimna: Also add any other constructions.]

$(2,3)$**-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 [**boneh2018-darkmatter**])**.** 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 $y = \mathbf{B}w$.

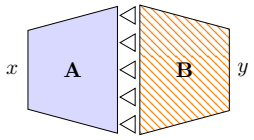


Figure 2: $(2,3)$-constructions

Circuit description of constructions. We can represent our constructions by circuits consisting of the gates described previously. This approach is similar to that of [**boyle2019-fss-preprocess**]. We formally define computation circuit representations for our constructions in Definition 3.3.

Definition 3.3 (Computation circuit)**.** A computation circuit C with input space $\mathbb{G}^{\mathsf{in}} = \prod \mathbb{G}_i^{\mathsf{in}}$ and output space $\mathbb{G}^{\mathsf{out}} = \prod \mathbb{G}_i^{\mathsf{out}}$ is a (labeled) directed acyclic graph $(\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ denotes the set of vertices and $\mathcal{E}$ denotes the set of edges according to the following:

- Each source vertex corresponds to exactly one $\mathbb{G}_i^{\mathsf{in}}$ and vice versa. The label for the vertex is the identity function on the corresponding $\mathbb{G}_i^{\mathsf{in}}$. Each sink vertex corresponds to exactly one $\mathbb{G}_i^{\mathsf{out}}$ and vice versa. The label for the vertex is the identity function on the corresponding $\mathbb{G}_i^{\mathsf{out}}$. Each non-source $V \in \mathcal{V}$ is labeled with a gate $\mathcal{G}_V \in \mathsf{Gates}$ that computes the function $\mathcal{G}_V : \mathbb{G}_V^{\mathsf{in}} \to \mathbb{G}_V^{\mathsf{out}}$.
- For an edge (V_a, V_b), let $\mathbb{G}_{V_a}^{\mathsf{out}} = \prod \mathbb{G}_{V_a,i}^{\mathsf{out}}$ and $\mathbb{G}_{V_b}^{\mathsf{in}} = \prod \mathbb{G}_{V_b,i}^{\mathsf{in}}$. Then, there exists indices j and k such that $\mathbb{G}_{V_a,j}^{\mathsf{out}} = \mathbb{G}_{V_b,k}^{\mathsf{in}}$. Further, for each input $\mathbb{G}_{V_b,i}^{\mathsf{in}}$ for V_b, there is some edge (V_c, V_b) that satisfies the above.
- The evaluation of the gate for vertex V on input $x \in \mathbb{G}_V^{\mathsf{in}}$ is defined as $y = \mathcal{G}_V(x)$. The evaluation of the circuit C, denoted by $\mathsf{Eval}_C(x)$, where $x \in \mathbb{G}^{\mathsf{in}}$ is the value $y \in \mathbb{G}^{\mathsf{out}}$, that is obtained by recursively evaluating each gate function in the circuit.

Let $\mathsf{F} = \{\mathsf{F}_\lambda\}_{\lambda \in \mathbb{N}}$ denote a family of functions $\mathsf{F}_\lambda : \mathcal{X}_\lambda \to \mathcal{Y}_\lambda$. We say that $\{C_\lambda\}_{\lambda \in \mathbb{N}}$ is a family of computation circuits for F if all C_λ have the same topological structure, and for all $\lambda \in \mathbb{N}$, $\mathsf{F}_\lambda(x) = \mathsf{Eval}_C(x)$ for all $x \in \mathcal{X}_\lambda$.

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. This section is structured as follows. First, in Section **??**, we provide a technical overview for our protocol design. This includes computation protocols for each of our circuit gates as well as a generic way to compose them to obtain fully distributed protocols in the preprocessing model for all our constructions. Section **??** quantifies this approach by providing concrete communication and preprocessing costs for distributed evaluations for our constructions. In Section **??**, for our $(2, 3)$-wPRF construction, we provide a 3-party protocol that does not require any preprocessing. Finally, in Section **??**, we provide two OPRF protocols for our $(2, 3)$-wPRF construction in the preprocessing model. All provided protocols are for the semi-honest setting.

[Mahimna: many of the subsections will undergo many changes but I wanted to get at least a first draft of the content written.]

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. This choice also prevents redoing the same communication when the masked value is already available from earlier gate evaluations.

Protocol notation and considerations. 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. When P_i knows the values $(a_1, \ldots, a_k)$, and has shares $[\![b_1]\!]^{(i)}, \ldots, [\![b_l]\!])$, we use the notation $\boldsymbol{\pi}(a_1, \ldots, a_k \mid [\![b_1]\!]^{(i)}, \ldots, [\![b_l]\!]^{(i)})$ to denote that P_i runs the protocol with its local inputs.

Given public values $a_1, \ldots, a_k$, it is straightforward for the protocol parties to compute a sharing $[\![f(a_1, \ldots, a_k)]\!]$ 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

[Yuval: In the following, I suggest to switch to a more structured format. For instance: an itemized list for each subprotocol, where the first item describes the functionality, the second describes the correlated randomness, and the third describes the actual protocol. It will also be good to have a table that summarizes the online and communication and correlated randomness (in bits) for each subprotocol.]

Protocol	Public Inputs	Shared Inputs	Shared Correlated Randomness	Output Shares (over base group $\mathbb{G}$)
$\boldsymbol{\pi}_{\mathsf{Lin}}^{\mathbf{A},p}$	$\mathbf{A}$	x	-	$y = \mathbf{A}x$ (over $\mathbb{Z}_p$)
$\boldsymbol{\pi}_{\mathsf{BL}}^{p}$	$\hat{\mathbf{K}}, \hat{x}$	-	$\tilde{\mathbf{K}}, \tilde{x}, \tilde{\mathbf{K}}\tilde{x}$	$y = \mathbf{K}x$ (over $\mathbb{Z}_p$)
$\boldsymbol{\pi}_{\mathsf{Convert}}^{(2,3)}$	$\hat{x}$ (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 the circuit gate protocols

We provide detailed (local) protocols to compute each circuit gate in this section. The description of inputs (including shared correlated randomness) and outputs for 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, and follows from the standard linear homomorphism of additive secret sharing.

- **Functionality**: Each party is provided with the matrix $\mathbf{A}$ and shares of the input x (over $\mathbb{Z}_p$). The goal is to compute shares of the output $y = \mathbf{A}x$.
- **Preprocessing**: None required.
- **Protocol details**: For the protocol $\boldsymbol{\pi}_{\mathsf{Lin}}^{\mathbf{A},p}(\mathbf{A} \mid x)$, each party P_i computes its output share as $[\![y]\!]^{(i)} = \mathbf{A}\,[\![x]\!]^{(i)}$. Note that this works because $\mathbf{A}x = \sum_{\mathrm{P}_i \in \mathcal{P}} \mathbf{A}\,[\![x]\!]^{(i)}$ as a direct consequence of the linear homomorphism of additive shares.

Bilinear gate protocol $\boldsymbol{\pi}_{\mathsf{BL}}^{p}$. The bilinear gate protocol is essentially a generalization of Beaver's multiplication triples [**boyle2019-fss-preprocess**; **beaver1991-triples**] that computes the multiplication of two shared inputs. For Beaver's protocol, to compute a sharing of $y = ab$ given shares of a and b (all sharings are over a ring $\mathcal{R}$), the protocol parties are provided shares of a randomly sampled triple of the form $(\tilde{a}, \tilde{b}, \tilde{a}\tilde{b})$ in the preprocessing stage. Beaver's protocol first reconstructs the masked inputs $\hat{a}$ and $\hat{b}$ after which local computation is enough to produce shares of the output. For our bilinear gate protocol, we assume that all parties are already provided with the masked inputs (to move the communication outside of the gate protocol), along with correlated randomness similar to a Beaver triple.

- **Functionality**: Abstractly, the goal of the bilinear gate protocol is to compute shares of the output $y = \mathbf{K}x$ given shares of both inputs $\mathbf{K}$ and x. For our purpose however, the masked inputs will have already been reconstructed beforehand, i.e., each party is provided with $\hat{\mathbf{K}}$ and $\hat{x}$ publicly, along with shares of correlated randomness similar to a Beaver triple (see below).
- **Preprocessing**: Each party is provided shares of $\tilde{\mathbf{K}}, \tilde{x}$, and $\tilde{\mathbf{K}}\tilde{x}$ as correlated randomness.

- **Protocol details**: For the protocol $\boldsymbol{\pi}^{p}_{\mathsf{BL}}(\hat{\mathbf{K}}, \hat{x} \mid \tilde{\mathbf{K}}, \tilde{x}, \tilde{\mathbf{K}}\tilde{x})$, each party P_i computes its share of $\hat{y}$ as:

$$[\![\hat{y}]\!]^{(i)} = \left[\!\left[\hat{\mathbf{K}}\hat{x}\right]\!\right]^{(i)} - \hat{\mathbf{K}}\,[\![\tilde{x}]\!]^{(i)} - \left[\!\left[\tilde{\mathbf{K}}\right]\!\right]^{(i)}\hat{x} + \left[\!\left[\tilde{\mathbf{K}}\tilde{x}\right]\!\right]^{(i)}$$

Correctness. Note that this works since:

$$\begin{aligned}\sum_{\mathrm{P}_i \in \mathcal{P}} [\![\hat{y}]\!]^{(i)} &= \hat{\mathbf{K}}\hat{x} - \hat{\mathbf{K}}\tilde{x} - \tilde{\mathbf{K}}\hat{x} + \tilde{\mathbf{K}}\tilde{x} \\ &= (\mathbf{K} + \tilde{\mathbf{K}})x - \tilde{\mathbf{K}}(x + \tilde{x}) + \tilde{\mathbf{K}}\tilde{x} \\ &= \mathbf{K}x\end{aligned}$$

Since the output of the bilinear gate will usually feed into a conversion gate which requires the input to be already masked, as an optimization, we can have the bilinear gate itself compute shares of the masked output, i.e., $\hat{y} = \mathbf{K}x + \tilde{y}$. This can be done by providing the correlated randomness $\tilde{\mathbf{K}}\tilde{x} + \tilde{y}$ instead of $\tilde{\mathbf{K}}\tilde{x}$. The upshot of this optimization is that one fewer piece of correlated randomness will be required.

$\mathbb{Z}_2 \to \mathbb{Z}_3$ conversion protocol $\boldsymbol{\pi}^{(2,3)}_{\mathsf{Convert}}$.

- **Functionality**: Abstractly, the goal of the $\mathbb{Z}_2 \to \mathbb{Z}_3$ conversion protocol is to convert a sharing of x over $\mathbb{Z}_2$ to a sharing of the same $x^* = x$, but now over $\mathbb{Z}_3$. For our purpose, the parties will be provided the masked input $\hat{x} = x \oplus \tilde{x}$ (i.e., masking is over $\mathbb{Z}_2$) directly along with correlated randomness that shares $\tilde{x}$ over $\mathbb{Z}_3$.

- **Preprocessing**: Each party is also provided with shares of the mask $r = \tilde{x}$ over $\mathbb{Z}_3$ as correlated randomness.

- **Protocol details**: For the protocol $\boldsymbol{\pi}^{(2,3)}_{\mathsf{Convert}}(\hat{x} \mid r)$, each party proceeds as follows:

$$[\![x^*]\!]^{(i)} = [\![\hat{x}]\!]^{(i)} + [\![r]\!]^{(i)} + (\hat{x} \odot [\![r]\!]^{(i)}) \mod 3$$

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

  Correctness. 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:

$$[\![x^*]\!]^{(i)}_j = [\![\hat{x}]\!]^{(i)}_j + [\![r]\!]^{(i)}_j + (\hat{x}\,[\![r]\!]^{(i)}_j \bmod 3) \mod 3$$

  Consider two cases:

  - If $\hat{x}_j = 0$, then $\tilde{x}_j = x_j$. Therefore, $\sum_{\mathrm{P}_i \in \mathcal{P}} [\![x^*]\!]^{(i)}_j = 0 + \tilde{x}_j = x_j$.
  - If $\hat{x}_j = 1$, then $x_j = 1 - \tilde{x}_j$. Therefore, $\sum_{\mathrm{P}_i \in \mathcal{P}} [\![x^*]\!]^{(i)}_j = 1 + 2\tilde{x}_j \bmod 3$. 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 $(\bmod 3)$ of shares $[\![x^*]\!]^{(i)}$ is the same as the corresponding component of x. Therefore, $\sum_{\mathrm{P}_i \in \mathcal{P}} [\![x^*]\!]^{(i)} \,(\bmod 3) = x$ will hold.

$\mathbb{Z}_3 \to \mathbb{Z}_2$ conversion protocol $\boldsymbol{\pi}^{(3,2)}_{\mathsf{Convert}}$.

- **Functionality**: Abstractly, the goal of the protocol is to convert a sharing of x over $\mathbb{Z}_3$ to a sharing of $x^* = x \bmod 2$ over $\mathbb{Z}_2$. For our purpose, the parties will be provided with the masked input $\hat{x} = x + \tilde{x} \bmod 3$ directly, along with correlated randomness over $\mathbb{Z}_3$ (see below).

- **Preprocessing**: Each party is also given shares (over $\mathbb{Z}_2$) of two vectors: $u = \tilde{x} \bmod 2$ and $v = (\tilde{x} + \mathbf{1} \bmod 3) \bmod 2$ as correlated randomness.

- **Protocol details**: For the protocol $\boldsymbol{\pi}^{(3,2)}_{\mathsf{Convert}}(\hat{x} \mid u, v)$, each party computes its share of x^* as follows: For each position $j \in [l]$,

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

  Correctness. To see why this works, consider three cases:

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

  Consequently, $\sum_{\mathrm{P}_i \in \mathcal{P}} [\![x^*]\!]^{(i)} \bmod 2 = x \bmod 2$ holds.

5.1.2 Composing Gate Protocols

[Mahimna: Still need to mention how this can be adapted from prior work] We now describe a general technique to evaluate circuit composed of the previously specified gates in a distributed fashion. We provide details for the semi-honest fully distributed setting (with preprocessing), where all inputs are secret shared between all parties initially. While the technique will also work for other settings (e.g., OPRF, public input), the concrete communication costs will be significantly worse than more specially designed protocols. For these settings, we will provide more efficient protocols than provided by this general technique.

Consider a circuit C (Definition 3.3) with input space $\mathbb{G}^{\mathsf{in}} = \prod \mathbb{G}^{\mathsf{in}}_i$. To evaluate C with input $(x_1, \ldots, x_l) \in \mathbb{G}^{\mathsf{in}}$, in the fully distributed setting, all parties are given additive shares for each x_i. Now, the distributed evaluation of C proceeds as follows:

- All vertices at the same depth in C are evaluated simultaneously, starting from the source vertices that contain the inputs of the computation.

- The evaluation of a (non-source) vertex in the graph of C is done by each party running the corresponding gate protocol locally on their share of the inputs.

- For an edge (V_a, V_b), suppose that the output of V_a is used as one of the inputs of V_b. If the gate protocol corresponding to $\mathcal{G}_V$ requires this input to be masked (e.g., the bilinear gate protocol), then before evaluating V_b, each party first masks its share of the output. Now, all parties simultaneously reveal their shares to publicly reveal the masked value. The masking values are provided to the parties in the preprocessing phase. The same value also need not be masked multiple times if it is required for multiple gates.

- The required output shares of the distributed evaluation are given by the evaluation of the sink vertices in the circuit.

Communication cost. Since the gate protocols themselves are locally computable, the communication cost during a distributed evaluation of a circuit comes solely from the public reconstructions of masked values required for gate protocols. For example, before feeding the output x of a $\mathsf{Lin}_2^{\mathbf{A}}$ gate into a $\mathsf{Convert}_{(2,3)}$ gate, in the distributed evaluation, all parties will first mask their shares of x to obtain shares of $\hat{x}$. Then, the parties will exchange messages to reconstruct the $\hat{x}$ value required for $\boldsymbol{\pi}^{(2,3)}_{\mathsf{Convert}}$.

Consider N parties taking part in the distributed evaluation. To reconstruct an l-bit value $\hat{x}$ that is additively shared among the parties, one of the following can be done.

- Each party sends its share of $\hat{x}$ to each other party. Now, all parties can compute $\hat{x}$ locally. This requires only 1 online round but has a communication cost of $(N-1)l$ bits per party. Each party sends $N-1$ messages. The simplest case for this is when $n = 2$, in which case, both parties can simultaneously exchange their shares, and add the two shares locally to reconstruct $\hat{x}$. This requires 1 online round, and has a communication cost of 1 message and l bits per party.

- All parties can send their share to a designated party, say P_1, who computes $\hat{x}$ and sends it back to everyone. This requires 2 rounds and has a communication cost of $(N-1)l$ bits for P_1 and l bits each for other parties. Here, P_1 sends $N-1$ messages while all other parties send a single message.

It is also straightforward to parallelize the communication to reduce the number of rounds. For this, suppose that we call an edge (V_a, V_b) *communication-requiring* if the output of the protocol for V_a needs to be masked before it is input into the protocol for V_b. Now, define the communication-depth of a vertex V as the maximum number of communication-requiring edges in the path from a source vertex to V. Now, instead of evaluating vertices with the same depth simultaneously, we will evaluate vertices with the same *communication-depth* together before the next communication round. By doing so, we can reduce the total number of rounds to the maximum communication-depth.

Correlated randomness. [Mahimna: todo] [Mahimna: I was thinking, here we mention a generic way to calculate randomness naively. There are two more things that would need to go somewhere. (1) Compressing randomness using e.g., PRG; (2) Generating the randomness. @yuval: What do you think?]

Primitive	Construction	Preprocessing (bits)	Communication		
			Size (bits)	Rounds	Messages
wPRF	23-wPRF 32-wPRF				
OWF	23-OWF 32-OWF				

Table 2: Preprocessing and communication cost for fully distributed evaluation of each of our candidate constructions. [Mahimna: to be filled. I think here, we should report the final numbers after compression. We should also decide whether to report the concrete numbers, numbers as functions of n, m, t or both. We can also report numbers for other constructions in literature here]

5.2 Distributed Evaluation in the Preprocessing Model

Equipped with our technical overview, it is simple to construct distributed protocols (with preprocessing) for our candidate wPRF and OWF constructions. By distributed evaluation, we mean that all inputs are secret shared between all parties and the protocol provides parties with a sharing of the output. As a concrete example, we provide the complete details of a 2-party distributed evaluation protocol for our $(2,3)$-wPRF candidate. In Table 2, for each our constructions, we provide the amount of preprocessed randomness required as well as the communication cost for the distributed protocol.

5.2.1 2-Party Protocol for $(2,3)$-wPRF.

We detail a 2-party semi-honest protocol for evaluating our $(2,3)$-wPRF candidate (Construction 3.1). In this setting, two parties, denoted by P_1 and P_2 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 $y = \mathsf{Lin}_3^{\mathbf{B}}(\mathbf{K}x)$ where $\mathbf{B} \in \mathbb{Z}_3^{t \times m}$ is a publicly known matrix.

Preprocessing. Our protocol requires preprocessed randomness as follows. A dealer randomly samples masks $\tilde{\mathbf{K}}$ and $\tilde{x}$ for $\mathbf{K}$ and x respectively. It also randomly samples $\tilde{w} \in \mathbb{Z}_2^m$ as mask for the intermediate output. Let $r \in \mathbb{Z}_3^m; r = \tilde{w}$ (viewed over $\mathbb{Z}_3$). The dealer creates additive sharings for $\tilde{\mathbf{K}}, \tilde{x}, \tilde{\mathbf{K}}\tilde{x}, \tilde{w}$, and $\tilde{r}$. Each $\mathrm{P}_{i \in \{1,2\}}$ is now provided $\left[\!\left[\tilde{\mathbf{K}}\right]\!\right]^{(i)}, [\![\tilde{x}]\!]^{(i)}, \left[\!\left[\tilde{\mathbf{K}}\tilde{x}\right]\!\right]^{(i)}, [\![\tilde{w}]\!]^{(i)}$, and $[\![r]\!]^{(i)}$ as preprocessing.

Protocol details. The distributed protocol proceeds as follows:

- Each party P_i computes masks their key and input shares as

$$\left[\!\left[\hat{\mathbf{K}}\right]\!\right]^{(i)} = [\![\mathbf{K}]\!]^{(i)} + \left[\!\left[\tilde{\mathbf{K}}\right]\!\right]^{(i)}$$
$$[\![\hat{x}]\!]^{(i)} = [\![x]\!]^{(i)} + [\![\tilde{x}]\!]^{(i)}$$

  using its given randomness. The shares are then exchanged simultaneously by both parties to reconstruct $\hat{\mathbf{K}}$ and $\hat{x}$.

- Each party P_i now locally runs $\boldsymbol{\pi}^2_{\mathsf{BL}}\left(\hat{\mathbf{K}}, \hat{x} \mid \left[\!\left[\tilde{\mathbf{K}}\right]\!\right]^{(i)}, [\![\tilde{x}]\!]^{(i)}, \left[\!\left[\tilde{\mathbf{K}}\tilde{x}\right]\!\right]^{(i)}\right)$ and adds to it its share of $\tilde{w}$ to obtain its share (over $\mathbb{Z}_2$) of $\hat{w} = \mathbf{K}x + \tilde{w}$. The shares are then exchanged simultaneously by both parties to reconstruct $\hat{w}$.
- Each party P_i now locally runs $\boldsymbol{\pi}^{(2,3)}_{\mathsf{Convert}}(\hat{w} \mid [\![r]\!]^{(i)})$ to obtain its shares of $w^* = w$ (over $\mathbb{Z}_3$).
- Finally, each party P_i obtains its share of the final output y (over $\mathbb{Z}_3$) by running $\boldsymbol{\pi}^{\mathbf{B},3}_{\mathsf{Lin}}(\mathbf{B} \mid [\![w^*]\!]^{(i)})$.

Cost analysis. The distributed protocol takes 2 communication rounds in total, with both parties sending a message in each round. When $\mathbf{K}$ is a circulant matrix (i.e., it can be represented by n bits), each party communicates $2n$ bits in the first round and m bits in the second round.

Public input setting. [Mahimna: todo]

5.3 3-party Distributed Evaluation

5.4 Oblivious PRF Evaluation in the Preprocessing Model

[Mahimna: This section still needs to undergo a lot of changes.] Our distributed evaluation protocols from Section **??** can be used directly for semi-honest PRF evaluation in the preprocessing model. Recall that in the OPRF setting, one party (called the "server") holds the key $\mathbf{K}$ and the other party (called the "client") holds the input x. The goal of the protocol is to have the client learn the output of the PRF for key $\mathbf{K}$ and input x, while the server learns nothing. In the semi-honest setting, both parties can first use the distributed protocol to obtain shares of the PRF output. The server can then send its share to the client so that only the client learns the final output. Such an OPRF protocol would require one extra round over the corresponding distributed PRF protocol.

We can however construct much better protocols whose efficiency rivals that of existing DDH-based OPRF protocols. Here, we provide two concrete efficient protocols for evaluating the $(2, 3)$-wPRF candidate (Construction 3.1) in the OPRF setting. Both protocols take 3 rounds and involve 2 messages from the server to the client and 1 message from the client to the server. The first server message however, is only required when the key needs to be changed (or re-masked). We call this the key-update phase. Now, when the masked key is already known, our protocols are optimal in the sense that they require only a single message from the client followed by a single message from the server. Since OPRF applications usually involve reusing the same key for many PRF invocations, in such a *multi-input* setting, our protocols are comparable to other 2-round OPRF protocols in literature (e.g., DDH-based).

In Section 6, we compare our protocols with common OPRFs protocols in terms of both computation and communication. A key observation was that in comparison to common OPRF protocols, our protocols are much faster to compute but require preprocessing as well as slightly more communication.

Oblivious PRF Protocol 1. This protocol is in spirit similar to the distributed evaluation for the $(2, 3)$-wPRF construction. Since $\mathbf{K}$ is known to the server, and x is known to the client, both parties do not need to exchange their shares to reconstruct the masked values $\hat{\mathbf{K}}$ and $\hat{x}$; the party

that holds a value can mask it locally and send it to the other party. This allows us to decouple the server's message that masks its PRF key from the rest of the evaluation. To update the key, the server can simply send $\hat{\mathbf{K}} = \mathbf{K} + \tilde{\mathbf{K}}$ to the client. Many PRF evaluations can now be done using the same $\hat{\mathbf{K}}$.

The protocol requires the following preprocessed randomness. The mask $\tilde{\mathbf{K}}$ is given to the server only when the key-update phase needs to be run. For PRF evaluations, the dealer samples $w \in \mathbb{Z}_2^m$ at random and provides the server and client $\mathbb{Z}_2$ shares of $\tilde{w}$ along with $\mathbb{Z}_3$ shares of $r = \tilde{w}$. Additionally, the dealer also generates an OLE correlation pair $(\tilde{\mathbf{K}}, \tilde{v})$ and $(\tilde{x}, \hat{v})$ such that $\tilde{\mathbf{K}} \in \mathbb{Z}_2^{m \times n}$ is a random circulant matrix that is same for all correlations, $\tilde{v} \xleftarrow{\$} \mathbb{Z}_2^m$, $\tilde{x} \xleftarrow{\$} \mathbb{Z}_2^n$, and $\hat{v} = \tilde{\mathbf{K}}\tilde{x} + \tilde{v}$. The server is given $(\tilde{\mathbf{K}}, \tilde{v})$ while the client is given $(\tilde{x}, \hat{v})$. Note that we simply use OLE correlations and do not make use of an actual OLE protocol. In practice, if the key-update phase is run after every k evaluations (where k is known), the OLE correlations for all evaluations can be preprocessed at the beginning. [Mahimna: need to write about / point to another section for generating the OLE correlations.]

Assuming that the masked key $\hat{\mathbf{K}}$ is known to the client, for an input x, the evaluation protocol now proceeds as follows:

- The client computes $\hat{x} = x + \tilde{x}$ and $[\![\hat{w}]\!]^{(2)} = -\hat{\mathbf{K}}\tilde{x} + \hat{v} + [\![\tilde{w}]\!]^{(2)}$ and sends both $\hat{x}$ and $[\![\hat{w}]\!]^{(2)}$ to the server.
- The server first computes $[\![\hat{w}]\!]^{(1)} = \mathbf{K}\hat{x} - \tilde{v} + [\![\tilde{w}]\!]^{(1)}$, and adds to it the client's share to reconstruct $\hat{w}$. Identical to the distributed protocol, the server now runs $\boldsymbol{\pi}_{\mathsf{Convert}}^{(2,3)}$ followed by $\boldsymbol{\pi}_{\mathsf{Lin}}^{\mathbf{B},3}$ to obtain its share $[\![y]\!]^{(1)}$ of the PRF output. Finally, it sends both $\hat{w}$ and $[\![y]\!]^{(1)}$ to the client.
- The client also runs $\boldsymbol{\pi}_{\mathsf{Convert}}^{(2,3)}$ followed by $\boldsymbol{\pi}_{\mathsf{Lin}}^{\mathbf{B},3}$ to obtain its share $[\![y]\!]^{(2)}$ of the PRF output. It can now use the server's share to reconstruct the PRF output y.

For evaluating a client input, the protocol takes 2 rounds and involves a single message in each direction. The client sends $2n$ bits while the server sends m bits and t $\mathbb{Z}_3$ elements. For our parameters ($n = m = 256, t = 81$), and with proper $\mathbb{Z}_3$ packing, this amounts to roughly 897 bits of total online communication. To update $\tilde{\mathbf{K}}$, the server sends a 256-bit message to the client.

Oblivious PRF Protocol 2. For the second protocol, the server masks the PRF in a different way; a multiplicative mask is used instead of an additive one. For simplicity, suppose that $n = m$ and that $\mathbf{K}$ is a random full-rank circulant matrix. Then to mask $\mathbf{K}$, the server computes $\bar{\mathbf{K}} = \mathbf{R}\mathbf{K}$ using a random matrix $\mathbf{R}$ that is also full-rank and circulant. $\mathbf{R}$ will be provided as preprocessing to the server, and can be reused for multiple PRF evaluations. The server will send $\bar{\mathbf{K}}$ to the client in the key-update phase. Note that since the product of two circulant matrices is also circulant, this message is only n bits. Additionally, since the product $\mathbf{RK}$ is essentially a convolution, it can be efficiently computed in $\Theta(n \log n)$ asymptotic runtime using the fast Fourier transform (FFT) algorithm.

The protocol requires the following preprocessed randomness. The mask $\mathbf{R}$ is given to the server only when the key-update phase needs to be run. For PRF evaluations, similar to the first protocol,

the dealer samples $w \xleftarrow{\$} \mathbb{Z}_2^m$ and provides the server and client $\mathbb{Z}_2$ shares of $\tilde{w}$ along with $\mathbb{Z}_3$ shares of $r = \tilde{w}$. Additionally, the dealer gives $\tilde{u} \xleftarrow{\$} \mathbb{Z}_2^m$ to the client and $\tilde{v} = \mathbf{R}^{-1}\tilde{u} + \tilde{w}$ to the server. Assuming that the masked key $\bar{\mathbf{K}}$ is known to the client, for an input x, the evaluation protocol now proceeds as follows:

- The client computes $\hat{u} = \bar{\mathbf{K}}x + \tilde{u}$ and sends it to the server.
- The server first computes $\mathbf{R}^{-1}\hat{u} + \tilde{v} = \mathbf{R}^{-1}(\mathbf{RK}x + \tilde{u}) + (\mathbf{R}^{-1}\tilde{u} + \tilde{w}) = \hat{w} \mod 2$. Identical to the distributed protocol, the server now runs $\boldsymbol{\pi}^{(2,3)}_{\mathsf{Convert}}$ followed by $\boldsymbol{\pi}^{\mathbf{B},3}_{\mathsf{Lin}}$ to obtain its share $[\![y]\!]^{(1)}$ of the PRF output. Finally, it sends both $\hat{w}$ and $[\![y]\!]^{(1)}$ to the client.
- The client also runs $\boldsymbol{\pi}^{(2,3)}_{\mathsf{Convert}}$ followed by $\boldsymbol{\pi}^{\mathbf{B},3}_{\mathsf{Lin}}$ to obtain its share $[\![y]\!]^{(2)}$ of the PRF output. It can now use the server's share to reconstruct the PRF output y.

For evaluating a client input, this protocol also takes 2 rounds and involves a single message in each direction. The client sends n bits while the server sends m bits and t $\mathbb{Z}_3$ elements. This is n fewer bits of communication as compared to the first protocol. The key-update phase is slower however, since it involves a convolution rather than a simple vector addition. For our parameters $(n = m = 256, t = 81)$, and with proper $\mathbb{Z}_3$ packing, this amounts to roughly 641 bits of total online communication. To update $\bar{\mathbf{K}}$, the server sends a 256-bit message to the client.

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 $\mathbb{Z}_2^{256\times256}$ 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 4, we compare the performance gain from these optimizations.

Bit packing for $\mathbb{Z}_2$ vectors. Instead of representing each element in a $\mathbb{Z}_2$ vector separately, we pack several elements into a machine word and operate on them together in an SIMD manner. For our architecture with 64-bit machine words, we can pack a vector in $\mathbb{Z}_2^{256}$ (e.g., the input x) into 4 words. Since the key $\mathbf{K}$ is circulant and can be represented with $n = 256$ bits, it can also be represented by 4 words. This results in a theoretical $\times64$ maximum speedup in run-time for operations involving $\mathbf{K}$ and x.

Bit slicing for $\mathbb{Z}_3$ vectors. We represent each element in $\mathbb{Z}_3$ using the two bits from its binary representation. For $z \in \mathbb{Z}_3$, the two bits are the least significant bit (LSB) $l_z = z \bmod 2$, and the most significant bit (MSB) h_z which is 1 if $z = 2$ and 0 otherwise. $\mathbb{Z}_3$ vectors are now also represented by two binary vectors, one containing the MSBs, and one containing the LSBs. Operations involving a $\mathbb{Z}_3$ vector are done on these binary vectors instead. In Table **??**, we specify how we perform common operations on $\mathbb{Z}_3$ elements using our bit slicing approach. We also take advantage of the bit packing optimization when operating on the binary vectors.

Operation	Result MSB	Result LSB
$z_1 + z_2 \bmod 3$	$(l_1 \vee l_2) \oplus (l_1 \vee h_2) \oplus (l_2 \vee h_1)$	$(h_1 \vee h_2) \oplus (l_1 \vee h_2) \oplus (l_2 \vee h_1)$
$-z_1 \bmod 3$	l_1	h_1
$z_1 z_2 \bmod 3$	$(l_1 \wedge l_2) \oplus (h_1 \wedge h_2)$	$(l_1 \wedge h_2) \oplus (h_1 \wedge l_2)$
$\mathrm{MUX}(s; z_1, z_2)$	$(h_2 \wedge s) \vee (h_1 \wedge \neg s)$	$(l_2 \wedge s) \vee (l_1 \wedge \neg s)$

Table 3: Operations in $\mathbb{Z}_3$. z_1 and z_2 are elements in $\mathbb{Z}_3$ with (MSB, LSB) $= (h_1, l_1)$ and (h_2, l_2) respectively. For a bit s, the operation $\mathrm{MUX}(s; z_1, z_2)$ outputs z_1 when $s = 0$ and z_2 when $s = 1$. [Mahimna: I'm debating whether we need to have this table at all. Once we specify that we split $\mathbb{Z}_3$ elements into 2 bits, for any $\mathbb{Z}_3$ operation, the corresponding operation on the bits can easily be computed using a truth table.]

Lookup table for matrix multiplication. Recall that the $(2,3)$-wPRF evaluation contains a mod-3 linear map using a public matrix $\mathbf{B} \in \mathbb{Z}_3^{81\times 256}$. Specifically, it computes the matrix-vector product $\mathbf{B}w$ where $w \in \mathbb{Z}_3^{256}$. Since $\mathbf{B}$ is known prior to evaluation, we can use a lookup table to speedup the multiplication by $\mathbf{B}$. The same preprocessing can also by reused for multiple evaluations of the wPRF.

For this, we partition $\mathbf{B}$, which has $m = 256$ columns, into 16 slices of 16 columns each. These matrices, denoted by $\mathbf{B}_1, \ldots, \mathbf{B}_{16}$, are all in $\mathbb{Z}_3^{81\times 16}$. Now, for each $\mathbf{B}_i$, we will effectively build a lookup table for its multiplication with any $\mathbb{Z}_3$ vector of length 16. A point to note here is that since we represent $\mathbb{Z}_3$ vectors by two binary vectors (from the bit slicing optimization), it is sufficient to preprocess multiplications (modulo 3) for binary vectors of length 16. To multiply $\mathbf{B}_i$ by a vector in $\mathbb{Z}_3^{16}$, we can first multiply it separately by the corresponding MSB and LSB vectors, and then subtract the former from the later modulo 3. This works since for $z_1, z_2 \in \mathbb{Z}_3$ the multiplication $z_1 z_2 \bmod 3$ can be given by $z_1(2 \cdot h_2 + l_2) \bmod 3 = z_1 l_2 - z_1 h_2 \bmod 3$ where h_2, l_2 are the MSB and LSB of z_2 respectively. Now, to multiply $\mathbf{B}$ by $v \in \mathbb{Z}_3^{256}$, we first evaluate all multiplications of the form $\mathbf{B}_i v_i$ where v_i is the $\mathbb{Z}_3^{16}$ vector denoting the i^{th} slice of v if it was split into 16-element chunks. Then, multiplication by $\mathbf{B}$ is given by $\mathbf{B}v = \sum_{i\in[16]} \mathbf{B}_i v_i \mod 3$.

In general, a $\mathbf{B} \in \mathbb{Z}_3^{t\times m}$ can be partitioned into m/c partitions with c columns each (assume c divides m for simplicity), and would require a total multiplication lookup table size of $(m/c) \cdot 2^c \cdot t$ $\mathbb{Z}_3$ elements. Multiplying $\mathbf{B}$ by $v \in \mathbb{Z}_3^m$ requires $2(m/c)$ lookup table accesses (one each for MSB and LSB of v per partition of $\mathbf{B}$) and an addition of $(m/c - 1)$ $\mathbb{Z}_3^t$ vectors.

For our parameters, this results in a table size of roughly 135MB with proper $\mathbb{Z}_3$ packing. We chose $c = 16$ as a compromise between the size of the lookup table and increased computational efficiency.

Lookup table for matrix multiplication. [Mahimna: I added sufficient details from this into the earlier paragraph. If that looks good, then we can remove this] 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 follows: For each index $0 \le j < 2^{16}$, 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.

6.2 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: The proposed OPRF hits communication bottleneck for low bandwidth network since number of bits exchanged is almost two times as that in DDH-based OPRF[**LibSodium**]. Further calculation shows that the minimal network speed in which computation dominates communication is 7.86 Mbps. For network speed greater than 7.86Mbps, the proposed OPRF outperforms the DDH-based OPRF. [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 4 contains timing results for our centralized implementation of the 23-wPRF construction using these optimizations.

Distributed wPRF evaluation. We consider two 2-party weak PRF protocols. The first protocol is described in [**boneh2018-darkmatter**] 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.

Optimization			Runtime(μ sec)	Evaluations / sec
Packing	Bit Slicing	Lookup Table		
			156.41	6K
✓			26.84	37K
✓	✓		18.5	65K
✓	✓	✓	6.08	165K

Table 4: 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.]

Protocol	**Eval/sec**	**Runtime(μ sec)**
Fully Distributed wPRF [boneh2018-darkmatter]	36K	28.02
Fully Distributed wPRF	82K	12.12

Table 5: Optimized protocols runtime. These protocols utilize both the bit-packing and lookup table optimization

OPRF evaluation. In Table 6, 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 [**naor1999-oprf**], we use the libsodium library [**LibSodium**] 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(μ sec) Server	Client	Parallel	Preprocessing	Communication(bits) Server	Client
23-OPRF	Key Update	0.65	-	9.48	1835	593	512
	Evaluation	9.45	9.13				
23-OPRF*	Key Update	3.16	-	-	768	512	256
	Evaluation	4.84	8.84				
DDH-based OPRF		57.38	28.69		-	256	256

Table 6: Comparison of protocols for (semi-honest) OPRF evaluation in the preprocessing model. Runtimes are given in microseconds (μs). Server and client can run in parallel after exchange of bits at the end of each phase. Parallel column reports timing in such scenario. [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 $\tilde{w} \in \mathbb{Z}_2$ be secret shared. Then to establish shares of $r = \tilde{w} \pmod 3$, 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 $\hat{x} = x + \tilde{x}$, where $\tilde{x}$ is a random value, established during preprocessing so that each party has a share $[\tilde{x}]$. 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. $\tilde{x}$: a random vector in $\mathbb{Z}_2^n$, //Sampled from random tapes
2. $\tilde{w}$: the vector $\mathbf{A}\tilde{x}$ in $\mathbb{Z}_2^m$,
3. r: a sharing of $\tilde{w} \mod 3$, shares in $\mathbb{Z}_3^m$, //Tapes + one aux value
4. $\overline{r}$: a sharing of $1 - \tilde{w} \mod 3$, shares in $\mathbb{Z}_3^m$. //Computed from shares of r

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

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

as required.

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

1. The parties locally compute $\hat{w} \in \mathbb{Z}_2^m$ as $\hat{w} = \mathbf{A}\hat{x}$ (since both $\hat{x}$ and $\mathbf{A}$ are public).
2. Let z be a vector in $\mathbb{Z}_3^m$ and let z_i denote the i-th component. Each party defines

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

   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 $w = \mathbf{A}x$, updating the public value $\hat{x} = x + x'$ with $\hat{w} = w + \tilde{w}$. The second step is where the bits of w are cast from $\mathbb{Z}_2$ to $\mathbb{Z}_3$. The parties have sharings of $\tilde{w}$ and $1 - \tilde{w} \bmod 3$ (we focus on a single bit here, for simplicity). The key observation is that when $\hat{w} = 0$, then w and $\tilde{w}$ are the same, and when $\hat{w} = 1$, w and $\tilde{w}$ are different. So in the first case we set the shares of $z = w \mod 3$ to the shares of $[\tilde{w}] \mod 3$, and when $\hat{w} = 1$, we set the shares of z to the complement of $\tilde{w}$.

Communication Costs Here we quantify the cost of communication for the MPC inputs, 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 $\log_2(3) \approx 1.58$. [1] The size of the aux information is $m\ell_3$, the MPC input value has size n bits, and the broadcast values have size $t\ell_3$ bits (per party). The total in bits is thus

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

For the parameters $(n, m, t) = (128, 453, 81)$ the total is 972 bits (L1 security) and when $(n, m, t) = 256, 906, 162)$ the total is 1943 bits (L5 security). This compares favorably to Picnic at the same security level, which communicates 1161–1328 bits at L1 and 2295–2536 bits at L5, depending on whether LowMC uses a full or 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.

[1] To compress a vector $v \in \mathbb{Z}_3^n$, convert it to the integer it represents: $V = \sum_{i=0}^{n} v_i^i$ and output the binary representation of V. [Greg: TODO: If this appears elsewhere in the paper, just xref to it]

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 $s \in \{0,1\}^\kappa$ such that s expands to matrices $\mathbf{A} \in \mathbb{Z}_2^{m\times n}$ and $\mathbf{B} \in \mathbb{Z}_3^{m\times t}$ that is full rank (using a suitable cryptographic function, such as the SHAKE extendable output function [**sp800-185**]). We use unique $\mathbf{A}$ and $\mathbf{B}$ per signer in order to avoid multi-target attacks against F. Compute $y = F(x)$ and set (y,s) as the public key. We specify the scheme using random matrices, however, it could also be instantiated with circulant matrices, which may improve the performance of sign and verify operations.

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 [**picnic-spec**].

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 $|\mathsf{MPC}(n,m,t)|$[2]. 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,

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

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 7 gives some options. The row with is nearly the same as for Picnic, and we highlight the signature size of the $(2, 3)$-OWF. Signatures using the $(2, 3)$-OWF are five to fifteen percent shorter than Picnic using LowMC.

7.2 Distributed Picnic

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

[2]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 7 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)
$(128, 453, 81)$	$(16, 150, 51)$	13.30
	$(16, 168, 45)$	12.48
	$(16, 250, 36)$	**11.54**
	$(16, 303, 34)$	11.49
Picnic3-L1	$(16, 250, 36)$	12.60
$(128, 453, 81)$	$(64, 151, 45)$	13.59
	$(64, 209, 34)$	11.70
	$(64, 343, 27)$	**10.66**
	$(64, 515, 24)$	10.35
Picnic2-L1	$(64, 343, 27)$	12.36
$(256, 906, 162)$	$(16, 324, 92)$	50.19
	$(16, 400, 79)$	47.08
	$(16, 604, 68)$	**45.82**
Picnic3-L5	$(16, 604, 68)$	48.72
$(256, 906, 162)$	$(64, 322, 82)$	51.23
	$(64, 518, 60)$	44.04
	$(64, 604, 57)$	**43.45**
Picnic2-L5	$(64, 604, 58)$	46.18

Table 7: Signature size estimates for Picnic using the $(2, 3)$-OWF, compared to Picnic using LowMC. The first half of the table shows security level L1 with $N = 16$ and $N = 64$ parties, and the second half shows level L5.

$(2,3)$-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 $\hat{x}^{(k)} = x + \tilde{x}^{(k)}$ based on the secret key x and the random values $\tilde{x}^{(k)}$ 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.