A Maliciously-Secure Post-Quantum OPRF from Crypto Dark Matter

Diego F. Aranha, Aron van Baarsen, Adam Blatchley Hansen, Kent Nielsen, and Peter Scholl

Aarhus University, Aarhus, Denmark
{dfaranha,aron.van.baarsen,peter.scholl}@cs.au.dk

Abstract. We construct protocols for oblivious pseudorandom functions (OPRFs) based on alternating moduli assumptions in the "Crypto Dark Matter" paradigm (Boneh et al, TCC 2016). Prior OPRFs based on this type of assumption were only secure against a semi-honest adversary. We show how to obtain maliciously secure protocols, by leveraging new cut-and-choose techniques for generating correlated randomness based on vector oblivious linear evaluation (VOLE), which allow efficient conversions between different moduli in zero-knowledge and secure two-party computation.

Compared with the state-of-the-art GOLD OPRF (Yang et al, S&P 2025), our construction has a faster online phase in all settings, as well as overall better efficiency in the small-batch setting. Furthermore, our construction supports obtaining a secret-shared output, and can be extended to handle secret-shared inputs. This opens up additional applications in variants of private set intersection and secure database operations.

Table of Contents

A Maliciously-Secure Post-Quantum OPRF from Crypto Dark Matter 1
Diego F. Aranha, Aron van Baarsen, Adam Blatchley Hansen, Kent Nielsen, and Peter Scholl
1 Introduction 3
2 Technical Overview 5
3 Preliminaries 8
3.1 (Oblivious) Pseudorandom Functions 8
3.2 VOLE and VOLE-based ZK 9
3.3 BDOZ-style 2PC 11
4 PRF Construction 13
4.1 From Weak to Strong PRF 14
5 Building Blocks for Correlated Randomness 15
5.1 Local Doubly-Authenticated Bits 15
5.2 Authenticated OLE 20
5.3 Doubly-Authenticated Bits 20
5.4 Authenticated VOLE 21
6 OPRF Protocols 23
6.1 Malicious Protocol 23
6.2 Half-Malicious Protocol 26
6.3 Semi-Honest Protocol 30
6.4 Shared-Input Shared-Output Protocol 30
6.5 (Towards) Verifiable OPRF 34
7 Implementation and Benchmarks 37
7.1 Setup 37
7.2 Benchmarks 38
7.3 Comparison to Other OPRF 38
7.4 Potential Improvement Based on Recent Work 39
A Additional Functionalities 44
A.1 Randomness 44
A.2 Secure Equality Check 44
B Authenticated OLE 45
C Full Proofs 52
C.1 Local Doubly-Authenticated Bits 52
C.2 Doubly Authenticated Bits 54
C.3 Authenticated VOLE 55
C.4 Maliciously-Secure OPRF 58
C.5 Half-Half-Authenticated Bits 63

1 Introduction

An oblivious pseudorandom function (OPRF) is a cryptographic primitive which allows two parties, a sender and a receiver, to evaluate a PRF on the receiver's inputs using a key of the sender's choice, while the sender remains oblivious to the receiver's inputs and the receiver remains oblivious to the sender's key. OPRFs are a versatile primitive having applications such as password-authenticated key-exchange (PAKE), anonymous tokens, and private set intersection (PSI). However, the number of post-quantum secure OPRF constructions is limited, and of those, only a handful satisfy malicious security, while often suffering from poor performance.

We aim to fill this gap by constructing a maliciously-secure post-quantum OPRF with practical efficiency. Our construction is based on the "Crypto Dark Matter" PRF from Boneh et al. [15]. Since its introduction, this PRF construction has seen many follow-up works proposing improvements and corresponding OPRF protocols. However, these protocols have either been limited to semi-honest security [2, 15, 25] or suffer from high performance overheads while also relying on heuristic assumptions and game-based security definitions [3]. Simultaneously, some of these variants have recently seen attacks weakening their security [7, 22]. For these reasons, we chose to go with a conservative variant of the original PRF construction by Boneh et al. [15] and prove our construction satisfies standard simulation-based security against malicious adversaries. Additionally, we provide an implementation to demonstrate its practical efficiency.

Our protocol leverages recent developments in the silent generation of vector linear oblivious evaluation (VOLE) correlations [17, 18] to allow for efficient two-party evaluation of our PRF candidate, and simultaneously leverages recent developments in VOLE-based zero-knowledge (ZK) to protect against malicious parties. Along the way, we present an improved protocol for generating doubly authenticated bits [50] over small fields in the two-party setting. We consider several variants of our OPRF protocol for different application settings, including a semi-honest version, half-malicious versions where only the client is corrupted, as well as a version with secret-shared inputs and outputs.

Applications. Oblivious pseudorandom functions are a key ingredient in protocols for password-authenticated key exchange [40], anonymous credentials (as in PrivacyPass [24]) and many private set intersection protocols. Our construction can additionally allow both the inputs and outputs to be in secret-shared form, also called a *distributed* OPRF. This can be used for private database join operations on secret-shared data [43, 46], as well as advanced forms of private set intersection such as circuit PSI [54, 58].

Related Work. Despite recent progress, designing efficient post-quantum OPRFs in the fully malicious setting with respect to both latency and communication costs is still an open problem. There are multiple promising constructions in the semi-honest or malicious-client settings based on a range of assumptions [4, 16, 25, 27, 30, 33, 37, 52], but they tend to become impractical when lifted to fully malicious due to various performance penalties, mostly from zero-knowledge proofs. An overview of results can be found at [35].

One of the first proposals for a lattice-based verifiable OPRF was a feasibility result by Albrecht et al. [4], a round-optimal protocol based on the hardness of Ring Learning with Errors (RLWE) and one-dimensional Short Integer Solution (1D-SIS) problems. Adapting to the fully malicious setting was estimated to require additional $>$ 128GB of communication for the lattice-based zero-knowledge proofs. A follow-up result by Albrecht and Gur [5] reduced the overhead to around 316kB per query by adopting more efficient zero-knowledge proofs, among other improvements removing superpolynomial factors. Another lattice-based construction is LeOPaRd by Esgin et

al. [29], which introduces a new family of lattice problems (interactive Module LWE) to obtain a low communication per-query cost of 161KB, but under the limitation of 2^{32} queries per fixed client-side inputs.

Post-quantum OPRFs have also been constructed with isogenies. The first two constructions by Boneh, Kogan and Woo [16] respectively employ augmentable commitments, and extend the Naor-Reingold PRF [47] to the group action setting. While the semi-honest versions were remarkably communication-efficient, costs increased to respectively 2MB and 424KB in the malicious setting. A more recent proposal by Basso [9] achieves round optimality with parallel evaluations, but with communication again in the MBs.

In the line of more generic constructions, Beullens et al. [14] propose a framework for two-hash OPRFs, and provide an instantiation based on the Legendre PRF [52], using lattice-based oblivious transfer and VOLE-based zero-knowledge proofs. Their implementation requires around 748KB of communication and 185ms for a single evaluation.

In terms of performance, the current state of the art is GOLD [59], a post-quantum OPRF based on the power-residue PRF, which is a generalization of the Legendre PRF. It requires an overhead of 993KB[1] (resp. 793KB) for a single evaluation, but amortizing across large batches reduces the communication cost to only 1.9KB (resp. 0.1KB) in the fully malicious (resp. malicious-client) setting. A recent work of Doerner et al. [27] showed that in the malicious-client setting, the cost of a single (non-batched) evaluation can be reduced to just 1.14KB in the OT-hybrid model. Setting up the base OTs with post-quantum security would involve an additional one-time cost of $\approx$ 768KB [59].

As in the present paper, there has been a series of works constructing OPRFs from oblivious evaluations of the "Crypto Dark Matter" PRF. Alamati et al. [2] proposed a construction in the semi-honest model, however they do not extend to the fully malicious security setting. Albrecht et al. [3] achieves a verifiable OPRF against malicious adversaries by combining a fully homomorphic evaluation (FHE) of the PRF with a cut-and-choose heuristic for the client to detect cheating. The resulting construction requires 10.0KB for a semi-honest evaluation for a batch of 64 queries, and estimated as 160kB in the malicious case. The low communication cost comes at high server computational cost, however, due to the demanding FHE evaluation. Post-publication analysis of these constructions has degraded their security levels to the point of practical attacks being feasible [7, 22], both for variants of the inner PRF proposed in [2], and for the verifiability extension from [3].

Our techniques build on a line of work in efficient MPC [17, 31, 50] and zero-knowledge proofs [1, 11, 56]. The recent work of [1] also presents protocols for zero-knowledge modulus conversion with low communication, based on local PRGs, however, their setup protocol does not support small fields. We could potentially combine our approach with theirs to save some communication, at the cost of an additional assumption and more computation.

Our Contributions. Our main contributions are:

- We present the first efficient post-quantum OPRF based on the Crypto Dark Matter PRF that achieves simulation-based security against fully malicious adversaries. This includes extensions to one-sided malicious security and secret-shared input/output.
- We provide a full implementation in Rust, demonstrating competitive performance across different settings.
- We present new building blocks for MPC based on VOLE, including improved protocols for generating doubly-authenticated bits over small fields in the two-party setting, which may be of independent interest.

[1] We use the convention that 1KB=10^3B, whereas [59] uses 1KB=2^{10}B

2 Technical Overview

Dark-Matter PRF. The *"crypto dark matter"* or *"alternating moduli"* paradigm, initiated by Boneh et al. [15], constructs PRF candidates by mixing linear operations over small moduli. Their original weak PRF construction is given by

$$F_{\mathsf{weak}}^{\text{BIPSW}}(\boldsymbol{A}, \boldsymbol{x}) := \boldsymbol{B} \cdot_3 (\boldsymbol{A} \cdot_2 \boldsymbol{x}),$$

with key $\boldsymbol{A} \in \mathbb{F}_2^{m \times n}$, input $\boldsymbol{x} \in \mathbb{F}_2^n$ and $\boldsymbol{B} \in \mathbb{F}_3^{t \times m}$ a random public matrix. Here we denote $\boldsymbol{A} \cdot_2 \boldsymbol{x} := \boldsymbol{A} \cdot \boldsymbol{x} \bmod 2$ and $\boldsymbol{B} \cdot_3 \boldsymbol{y} := \boldsymbol{B} \cdot \boldsymbol{y} \bmod 3$. The efficiency of the PRF can be improved by choosing the key $\boldsymbol{A}$ to be a structured matrix.

To securely compute $F_{\mathsf{weak}}^{\text{BIPSW}}$ in a setting where one party has $\boldsymbol{x}$ and the other party has $\boldsymbol{A}$, a typical MPC-based approach is to first obtain secret shares over $\mathbb{F}_2$ of the matrix-vector product $\boldsymbol{Ax}$, and then perform an interactive share conversion protocol to convert these to $\mathbb{F}_3$ shares. Multiplication by $\boldsymbol{B}$ can then be done locally on the shares, before the server sends its share of the result to the client.

In the semi-honest setting, the matrix multiplication can be done using n oblivious transfers (OTs) on m-bit strings, which can also be preprocessed using correlated randomness. However, the $O(n^2)$ computational cost can still be significant. To avoid this, prior works have proposed to optimize this using either a structured matrix $\boldsymbol{A}$ such as a circulant matrix [15, 25], or replacing $\boldsymbol{Ax}$ with a component-wise product $\boldsymbol{k} \odot \boldsymbol{x}$ [2], which can be implemented with just n instances of bit-OT. The share conversion step is more involved. In the semi-honest setting, various methods based on either oblivious transfer [2] or forms of correlated randomness [15, 25] have been proposed.

Our Weak-PRF Variant. We propose to instantiate the matrix-vector product $\boldsymbol{x} \mapsto \boldsymbol{Ax}$ as a finite field multiplication by a fixed element of $\mathbb{F}_{2^n}$. This allows us to securely compute shares of $\boldsymbol{Ax}$ using a *single* output of a vector-OLE protocol over the field. This is conceptually simpler (and potentially faster) than the n OTs used in previous, semi-honest approaches, however, our main motivation is that it turns out to be easier to work with for malicious security. As an added benefit, we also avoid the security weaknesses associated with variants based on circulant matrices [21, 38] or component-wise multiplication [7, 53], thanks to the more rigid structure of the finite field multiplication matrix.

Formally, let $\mathsf{vec}_\alpha : \mathbb{F}_{2^n} \to \mathbb{F}_2^n$ be the map sending an element $x = \sum_{i=0}^{n-1} x_i \cdot \alpha^i$ to the vector $\boldsymbol{x} = (x_0, \ldots, x_{n-1}) \in \mathbb{F}_2^n$, where $\{1, \alpha, \ldots, \alpha^{n-1}\}$ is an $\mathbb{F}_2$-basis of $\mathbb{F}_{2^n}$, and let $\mathsf{elt}_\alpha : \mathbb{F}_2^n \to \mathbb{F}_{2^n}$ be the converse map.

The weak PRF F'_k is parametrized by a random matrix $\boldsymbol{B} \in \mathbb{F}_3^{t \times n}$. For a key $k \in \mathbb{F}_{2^n}$ and input $x \in \mathbb{F}_{2^n}$, we define

$$F'_k(x) = \boldsymbol{B} \cdot_3 \mathsf{vec}_\alpha(k \cdot x). \tag{1}$$

Secure Evaluation of the Weak PRF. At a high level, our approach towards evaluating F'_k with malicious security is to use authenticated secret-sharing based on information-theoretic MACs [13, 48], combined with carefully designed protocols for generating correlated randomness in a preprocessing phase. We prove simulation-based security of all our building blocks and PRF in a simplified version of the universal composability (UC) framework [19].

For a secret value $x \in \mathbb{F}_q$ held by party P_0, let $M_x \in \mathbb{F}_{q^r}$ denote P_0's MAC on x over a suitably large extension field where $q^r \approx 2^\kappa$. P_1 gets a secret, global key $\Delta_1 \in \mathbb{F}_{q^r}$, as well as the local key $K_x = M_x + x \cdot \Delta_1$ for this MAC. This is a linearly homomorphic MAC, which can be efficiently

generated using a vector oblivious linear evaluation (VOLE) protocol such as [55]. We use these MACs both to ensure reliable openings of secret-shared values, as well as *prove* properties about authenticated values in zero knowledge, using VOLE-based zero-knowledge proofs [26, 57].

As a first step, suppose the parties have preprocessed correlated randomness in the form of authenticated shares of $s \cdot r_i \in \mathbb{F}_{2^n}$, where the server holds s, the client holds the r_i's. This can be seen as a form of *authenticated* random VOLE. In a batch setting where the server's key k is fixed, one of these correlations can easily be used to create an authenticated sharing of $k \cdot x$, by having each party send a correction value.

Next, to convert these to authenticated sharings over $\mathbb{F}_3$, we rely on the tool of *doubly-authenticated bits* [6, 50] (or, daBits), that is, random bits $b_j \in \{0, 1\}$, which are secret-shared (with MACs) twice: once over $\mathbb{F}_2$, and again over $\mathbb{F}_3$. Given n daBits, the parties can open $k \cdot x + \mathsf{elt}(\boldsymbol{b}) \in \mathbb{F}_{2^n}$, and then use the shares of $\boldsymbol{b} \in \mathbb{F}_3^n$ to locally obtain a share of each bit of $k \cdot x$ over $\mathbb{F}_3$. The parties can then locally apply $\boldsymbol{B}$, and reconstruct the result as desired.

Instantiating the Building Blocks. The main challenge is to find efficient protocols for creating the correlated randomness, in particular, doubly-authenticated bits in $\mathbb{F}_2/\mathbb{F}_3$. We outline our constructions of these building blocks below.

Authenticated VOLE. The authenticated VOLE functionality can be realized by using a standard VOLE functionality together with several consistency checks. First, the parties use a VOLE to obtain (unauthenticated) secret shares of $\boldsymbol{v} + \boldsymbol{w} = \gamma \cdot \boldsymbol{y}$, where the receiver holds $\boldsymbol{w}$, $\boldsymbol{y}$ and the sender holds γ, $\boldsymbol{v}$.

To authenticate $\boldsymbol{w}$ under the sender's MAC key Δ_1, the parties use a VOLE to get shares $\bar{v} + \bar{w} = (\Delta_1 \cdot \gamma) \cdot \boldsymbol{y}$, which gives the commitment $\bar{\boldsymbol{w}} = \Delta_1 \cdot \boldsymbol{w} + (\Delta_1 \cdot \boldsymbol{v} - \bar{\boldsymbol{v}})$. To check the correctness of this commitment, the receiver additionally commits to $\boldsymbol{w}$ under the key Δ_1 using a VOLE and the parties check if these two commitments to $\boldsymbol{w}$ are consistent. To make sure the sender uses consistent keys $\gamma, \Delta_1, \Delta_1 \cdot \gamma$ for the three VOLES, the receiver appends a random field element to each of the values they commit to.

To authenticate $\boldsymbol{v}$ under the receiver's MAC key Δ_0, the parties use a VOLE to get shares $\gamma \cdot (\Delta_0 \cdot \boldsymbol{y}) = \tilde{\boldsymbol{v}} + \tilde{\boldsymbol{w}}$, which gives the commitment $\tilde{\boldsymbol{v}} = \Delta_0 \cdot \boldsymbol{v} + (\Delta_0 \cdot \boldsymbol{w} - \tilde{\boldsymbol{w}})$. To check the correctness of this commitment, the receiver commits to Δ_0 under the sender's key γ and proves in zero-knowledge that the commitment to $\Delta_0 \cdot \boldsymbol{y}$ is consistent with the commitments to Δ_0 and $\boldsymbol{y}$. To make sure the receiver actually commits to their MAC key Δ_0, the sender commits to γ under Δ_0 and the parties check consistency between the two commitments.

DaBits in $\mathbb{F}_2/\mathbb{F}_p$. The state-of-the-art protocol for generating daBits is [6]; they observed that given secret shares of a random bit in $\mathbb{F}_p$, two parties can locally derive a secret-sharing of the same bit in $\mathbb{F}_2$. If the parties then authenticate these $\mathbb{F}_2$ shares, they obtain a daBit. To prevent malicious parties from authenticating the wrong shares, they run a simple consistency check by taking random linear combinations over $\mathbb{F}_p$, and checking the least significant bit is consistent with the same combination over $\mathbb{F}_2$. Unfortunately, this check inherently requires a large value of p to avoid overflow, so it does not work when $p = 3$. An alternative approach [50] uses cut-and-choose techniques instead of linear combinations, and works for any p, but is much less efficient, requiring around 10kB of communication per daBit.

We present a different approach to daBit generation, first creating *local daBits*, where the underlying authenticated bits are known to one party, with a lightweight consistency check to verify correctness. Then, to obtain a daBit, we securely combine two local daBits (known to different

parties) by emulating an XOR operation modulo p, using a single, authenticated OLE correlation. This idea is similar in spirit to the approach for generating edaBits[2] from [28], except we develop new techniques tailored to the daBit setting.

Generating Local daBits. Our main technical contribution is a new method for verifying correctness of a batch of local daBits. A standard, bucketing technique (see e.g. [31]) to check a batch of $\ell B + C$ daBits is to randomly open C to check correctness (ensuring that not too many are "bad"), and then shuffle the remaining ones into ℓ buckets of size B. Doing pairwise correctness checks within each bucket allows extracting one correct daBit per bucket, with high probability. While this technique is reasonably efficient, it inherently has a factor B overhead, where $B = O(\lambda / \log \ell)$, and in practice can be as small as 3, but only for very large batch sizes of more than 1 million.

We show that, with a different correctness check using linear codes and a novel combinatorial analysis, it's actually possible to extract $\approx B/2$ correct daBits per bucket, where we use slightly larger buckets depending on the choice of code (e.g., $B = 12$ in our implementation), obtaining constant overhead. This reduces communication by around 30–50%, while also allowing smaller batch sizes than the standard approach. Our technique may be independently useful for checking other forms of correlated randomness, particularly in a ZK setting.

Authenticated OLE. As mentioned above, to go from *local* daBits to *secret-shared* daBits, we rely on authenticated OLE correlations over $\mathbb{F}_3$. These are similar to authenticated multiplication triples, although slightly simpler to generate. Our protocol is based on the $\mathbb{F}_2$ triple generation protocol from [34], adapted to our setting.

Upgrading to a Strong OPRF. To complete our OPRF protocol, we need to upgrade security from a *weak* OPRF, where the client's input x must be uniformly random, to a *strong* OPRF supporting arbitrary (even maliciously chosen) inputs. Here, we use the transformation from [15], which upgrades a dark matter-style weak PRF into a strong one, by encoding the input with a linear code over $\mathbb{F}_3$ and mapping the result into $\mathbb{F}_2$, before applying the weak PRF. To prove that the encoding is done correctly, we can use local daBits to do an $\mathbb{F}_2/\mathbb{F}_3$ conversion, and then use the parity check matrix to check the encoded input is a codeword.

Comparison with Generic MPC. When comparing our specialized approach for evaluating our newly proposed dark-matter PRF variant with using an off-the-shelf approach like MP-SPDZ [41], the biggest differences (in order of impact) are the following:

(1) The protocol of [50] would be used to generate secret-shared daBits instead of our new approach of proving correctness of local daBits before combining them;
(2) Instead of proving that the input is encoded correctly for the transformation from a weak to a strong PRF, the encoding would be done inside MPC;
(3) The key-input multiplications would be computed using authenticated multiplication triples instead of our authenticated VOLE protocol.

Our first optimization saves a factor $B(B-1)$ multiplication triples over $\mathbb{F}_3$ (where $B \geq 3$) by proving correctness locally, and our new consistency check sacrifices significantly less candidate daBits, which additionally saves around a factor 2 in communication. The second optimization saves

[2] In edaBits, the $\mathbb{F}_p$ sharings are created as a single integer sharing, instead of bitwise. This requires a large p, so does not apply to $p = 3$.

about a factor 4 in the number of $\mathbb{F}_3$-multiplication triples needed (similar to Section 6.4). The third optimization replaces the use of a fresh multiplication triple to compute each key-input multiplication with an authenticated VOLE correlation to compute a batch of key-input multiplications. Since generating authenticated OLE/multiplication triples constitutes the bulk of the communication and computation (as discussed in Section 7.4), we expect our approach to save at least an order of magnitude in communication and computation in the preprocessing phase. In the online phase we expect our approach to use about half as much communication and computation due to our use of VOLE-based zero-knowledge proofs and correlations as opposed to secret-sharing based techniques.

Overview of Paper. We present preliminaries and our notation for VOLE, VOLE-based zero-knowledge, and secure two-party computation in Section 3, followed by our Dark Matter PRF variant in Section 4. Section 5 details our building blocks of doubly-authenticated bits and authenticated (V)OLE. Finally, we present our main OPRF protocols in Section 6 and evaluate our implementation in Section 7.

3 Preliminaries

Notation. We denote by $\lambda \in \mathbb{N}$ the computational security parameter and $\kappa \in \mathbb{N}$ the statistical security parameter. For $m \in \mathbb{N}$, we commonly write $[m]$ for the set of integers $\{1, \ldots, m\}$ and for $a, b \in \mathbb{N}$ with $a \leq b$ we denote $[a, b]$ for the set $\{a, \ldots, b\}$. We denote the pointwise product of two vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{F}_q^\ell$ as $\boldsymbol{x} \odot \boldsymbol{y} := (x_1 \cdot y_1, \ldots, x_\ell \cdot y_\ell)$ and will use the shorthand $\boldsymbol{x}^2 := \boldsymbol{x} \odot \boldsymbol{x}$. For a finite field $\mathbb{F}_q$ and an extension field $\mathbb{F}_{q^r}$ of degree r with $\mathbb{F}_q$-basis $\{1, \alpha, \ldots, \alpha^{r-1}\}$ we denote $\mathsf{elt}_\alpha : \mathbb{F}_q^r \to \mathbb{F}_{q^r}$ for the map sending $\boldsymbol{x} = (x_0, \ldots, x_{r-1})$ to $\sum_{i=0}^{r-1} x_i \cdot \alpha^i$, and denote $\mathsf{vec}_\alpha : \mathbb{F}_{q^r} \to \mathbb{F}_q^r$ for the converse map.

3.1 (Oblivious) Pseudorandom Functions

We use the standard concepts of a (weak or strong) pseudorandom function family (PRF), $\{F_\lambda\}_{\lambda \in \mathbb{N}}$. In a strong PRF, or PRF, the function outputs should be pseudorandom even if an adversary can query the function on arbitrary inputs. In a weak PRF, the adversary is only given random input/output pairs.

Definition 1 (Weak/Strong Pseudorandom Function (wPRF/PRF)). *Let $\mathcal{K} = \{\mathcal{K}_\lambda\}_{\lambda \in \mathbb{N}}$, $\mathcal{X} = \{\mathcal{X}_\lambda\}_{\lambda \in \mathbb{N}}$ and $\mathcal{Y} = \{\mathcal{Y}_\lambda\}_{\lambda \in \mathbb{N}}$ be collections of finite sets indexed by the security parameter λ. Let $\{F_\lambda\}_{\lambda \in \mathbb{N}}$ be an efficiently-computable family of functions $F_\lambda : \mathcal{K}_\lambda \times \mathcal{X}_\lambda \to \mathcal{Y}_\lambda$.*

*Then we say that $\{F_\lambda\}_{\lambda \in \mathbb{N}}$ is an (ℓ, t, ε)-**weak pseudorandom function (wPRF)** if for any adversary $\mathcal{A}$ running in time $\leq t(\lambda)$, it holds that*

$$\left|\Pr\left[\mathcal{A}\left(1^\lambda, \{(x_i, F_\lambda(k, x_i))\}_{i \in [\ell(\lambda)]}\right)\right] - \Pr\left[\mathcal{A}\left(1^\lambda, \{(x_i, f_\lambda(x_i))\}_{i \in [\ell(\lambda)]}\right)\right]\right| \leq \varepsilon(\lambda),$$

where $k \xleftarrow{\$} \mathcal{K}_\lambda$, $f_\lambda \xleftarrow{\$} \{f : \mathcal{X}_\lambda \to \mathcal{Y}_\lambda\}$ and $x_1, \ldots, x_{\ell(\lambda)} \xleftarrow{\$} \mathcal{X}_\lambda$.

*We say that $\{F_\lambda\}_{\lambda \in \mathbb{N}}$ is a (t, ε)-**strong pseudorandom function (PRF)** if for any adversary $\mathcal{A}$ running in time $\leq t(\lambda)$, it holds that*

$$\left|\Pr\left[\mathcal{A}^{F_\lambda(k, \cdot)}(1^\lambda) = 1\right] - \Pr\left[\mathcal{A}^{f_\lambda(\cdot)}(1^\lambda) = 1\right]\right| \leq \varepsilon(\lambda),$$

where $k \xleftarrow{\$} \mathcal{K}_\lambda$ and $f_\lambda \xleftarrow{\$} \{f : \mathcal{X}_\lambda \to \mathcal{Y}_\lambda\}$.

Our batched OPRF functionality is shown in Fig. 1. Note that there are two main different flavours of ideal OPRF functionalities, one where the functionality outputs come from a public PRF family and the other where the outputs come from a perfectly random function, each suited for different applications. Our protocol realizes the former, with the addition that it allows the adversary to attempt to guess the key. This is an artifact of our protocol, where an adversary who manages to guess the PRF key can cheat. We briefly discuss in Section 6.5 whether our protocol can be extended to realize the latter variant, and refer to [20] for a more extensive discussion of different OPRF variants.

Fig. 1 Ideal (batched) OPRF functionality $\mathcal{F}_{\mathsf{OPRF}}$

Parameters: Two parties, $\mathcal{P}_0$ and $\mathcal{P}_1$, also referred to as the client and the server, respectively. A pseudorandom function family $F_\lambda : \mathcal{K}_\lambda \times \mathcal{X}_\lambda \to \mathcal{Y}_\lambda$ with key space $\mathcal{K}_\lambda$, input space $\mathcal{X}_\lambda$ and output space $\mathcal{Y}_\lambda$.

Initialize: Upon receiving $(\mathsf{sid}, \texttt{init}, \ell)$ from $\mathcal{P}_0$ and $\mathcal{P}_1$ with $\ell \in \mathbb{N}$: if $\mathcal{P}_1$ is corrupted, receive $k \in \mathcal{K}_\lambda$ from $\mathcal{A}$; otherwise, sample $k \xleftarrow{\$} \mathcal{K}_\lambda$. Send k to $\mathcal{P}_1$, store (sid, ℓ, k).

Evaluate: Upon receiving $(\mathsf{sid}, \texttt{eval}, \ell)$ from $\mathcal{P}_1$ and $\big(\mathsf{sid}, \texttt{eval}, (x_i)_{i\in[\ell]}\big)$ from $\mathcal{P}_0$ with $x_i \in \mathcal{X}_\lambda$ for all $i \in [\ell]$: output $(F_k(x_i))_{i\in[\ell]}$ to $\mathcal{P}_0$.

Key-Guess: If $\mathcal{P}_0$ is corrupted, receive $(\mathsf{sid}, \texttt{guess}, k')$ from $\mathcal{A}$ with $k' \in \mathcal{K}_\lambda$. If $k' = k$, return `success` to $\mathcal{P}_0$ and ignore all future key-guess queries with the same sid. Otherwise, return `abort` to both parties and abort.

3.2 VOLE and VOLE-based ZK

We use vector oblivious linear evaluation (VOLE) as a form of homomorphic commitment, which is also amenable to proving statements about committed values in zero-knowledge, using techniques from [10, 26, 57]. We formalise this with the following definition, adapted from [1]. The ideal functionality $\mathcal{F}^{q,r}_{\mathsf{sVOLE}}$ for VOLE and VOLE-ZK can be found in Figure 2. For a prime q and extension field parameter r, we define the *degree* of an arithmetic circuit $C : \mathbb{F}_q^\ell \to \mathbb{F}_q$ to be the maximal degree of the univariate polynomial $C(a_1X + b_1, \ldots, a_nX + b_n)$, where $a_i, b_i \in \mathbb{F}_{q^r}$.

Definition 2 (VOLE Commitment). *Let there be two parties $\mathcal{P}_0$ and $\mathcal{P}_1$. Then for $x \in \mathbb{F}_q$, where q is a prime, and $i \in \{0, 1\}$, we denote $[\![x]\!]^{(q)}_{d,i\oplus1} = ((x, \rho_x(t)), (\Delta, \gamma_x))$ to represent a degree-d VOLE commitment to x. That is, $\mathcal{P}_{i\oplus1}$ holds a commitment polynomial $\rho_x(t) := \rho_0+\rho_1\cdot t+\cdots+\rho_d\cdot t^d$, where $\rho_d = x$ and $\rho_0, \ldots, \rho_d \in \mathbb{F}_{q^r}$, and $\mathcal{P}_i$ holds the evaluation $\gamma_x := \rho_x(\Delta)$ for a global key $\Delta \in \mathbb{F}_{q^r}$ that is unknown to $\mathcal{P}_{i\oplus1}$. Moreover, to differentiate between commitments for different initializations sid_j of $\mathcal{F}^{q,r}_{\mathsf{sVOLE}}$, we will sometimes denote $[\![x]\!]^{(q)}_{d,i\oplus1,j}$ to indicate the commitment is under the global key Δ_j corresponding to sid_j. Finally, we will omit the degree and the index $i \in \{0, 1\}$ when they are clear from the context.*

VOLE commitments (under the same global key Δ and with the same receiving party $\mathcal{P}_{i\oplus1}$) support the following local, homomorphic operations (see [1, 10] for details):

- **Add:** $[\![x+y]\!]^{(q)}_d = [\![x]\!]^{(q)}_{d_1} + [\![y]\!]^{(q)}_{d_2}$ defined as $\rho_z(t) := t^{d_2-d_1}\cdot\rho_x(t)+\rho_y(t)$ and $\gamma_z := \Delta^{d_2-d_1}\cdot\gamma_x+\gamma_y$ if (w.l.o.g.) $d_1 \le d_2$.

Fig. 2 Ideal VOLE and VOLE-ZK functionality $\mathcal{F}_{\mathsf{sVOLE}}^{q,r}$

Parameters: Two parties, $\mathcal{P}_0$ and $\mathcal{P}_1$. Base field characteristic q, extension field parameter r (defining $\mathbb{F}_q$ and $\mathbb{F}_{q^r}$).

Initialize: Upon receiving $(\mathsf{sid}, \mathtt{init}, i)$ from $\mathcal{P}_0$ and $\mathcal{P}_1$ with $i \in \{0,1\}$, the functionality initializes an empty dictionary $\mathsf{D} = \{\}$, and samples $\Delta \xleftarrow{\$} \mathbb{F}_{q^r}$ if $\mathcal{P}_i$ is honest. Otherwise, receive $\Delta \in \mathbb{F}_{q^r}$ from $\mathcal{A}$. The functionality returns Δ to $\mathcal{P}_i$ and ignores all future `init` calls for the same (sid, i).

Evaluate: Upon receiving $(\mathsf{sid}, \mathtt{eval}, \ell, \mathsf{id}_1, \ldots, \mathsf{id}_\ell))$ from $\mathcal{P}_0$ and $\mathcal{P}_1$, with $\ell \in \mathbb{N}$ and $\{\mathsf{id}_j\}_j$ distinct identifiers:

- If $\mathcal{P}_{i\oplus 1}$ is honest, sample $\boldsymbol{u} \xleftarrow{\$} \mathbb{F}_q^\ell$. Otherwise, receive $\boldsymbol{u} \in \mathbb{F}_q^\ell$ from $\mathcal{A}$.
- If $\mathcal{P}_i$ is honest, sample $\boldsymbol{v} \xleftarrow{\$} \mathbb{F}_{q^r}^\ell$. Otherwise, receive $\boldsymbol{v} \in \mathbb{F}_{q^r}^\ell$ from $\mathcal{A}$.
- If $\mathcal{P}_{i\oplus 1}$ is honest, compute $\boldsymbol{w} := \boldsymbol{v} - \Delta \cdot \boldsymbol{u}$. Otherwise, receive $\boldsymbol{w} \in \mathbb{F}_{q^r}^\ell$ from $\mathcal{A}$ and replace $\boldsymbol{v} \leftarrow \boldsymbol{w} + \Delta \cdot \boldsymbol{u}$.
- Return $[\![\boldsymbol{u}]\!]_{i\oplus 1}^{(q)}$ to the parties, that is, output $(\boldsymbol{u}, \boldsymbol{w})$ to $\mathcal{P}_{i\oplus 1}$ and $(\Delta, \boldsymbol{v})$ to $\mathcal{P}_i$, and store $\mathsf{D}[\mathsf{id}_j] = u_j$, for $j = 1, \ldots, \ell$

Prove: Upon receiving $(\mathsf{sid}, \mathtt{prove}, \mathsf{id}_1, \ldots, \mathsf{id}_\ell, C_1, \ldots, C_m)$ from both parties, where C_j is a circuit from $\mathbb{F}_{q^r}^\ell \to \mathbb{F}_{q^r}$:

- Retrieve $x_j = \mathsf{D}[\mathsf{id}_j]$, for $j = 1 \in [\ell]$
- Compute $y_j = C_j(x_1, \ldots, x_\ell)$, for $j \in [m]$
- Let d be the maximum degree of any C_j
- If $\mathcal{P}_{i\oplus 1}$ is corrupt, wait for $\mathcal{A}$ to send $(\mathsf{sid}, \mathtt{guess}, S_\Delta)$, where $|S_\Delta| \le d$. Otherwise, set $S_\Delta = \emptyset$
- If $y_j = 0$ for all $j \in [m]$, or if $\Delta \in S_\Delta$, send `success` to both parties. Otherwise, send `abort` and abort.

Global-Key Query: On receiving $(\mathsf{sid}, \mathtt{guess}, \Delta' \in \mathbb{F}_{q^r})$ from $\mathcal{A}$, when $\mathcal{P}_{i\oplus 1}$ is corrupted: if $\Delta' = \Delta$, return `success` to $\mathcal{P}_{i\oplus 1}$. Otherwise, return `abort` to both parties and abort.

- **Affine function:** $[\![cx + y]\!]_d^{(q)} = c \cdot [\![x]\!]_d^{(q)} + y$, where $c, y \in \mathbb{F}_q$ are public, defined as $\rho_z(t) := c \cdot \rho_x(t) + t^d \cdot y$ and $\gamma_z := c \cdot \gamma_x + \Delta^d \cdot y$.
- **Multiply:** $[\![x \cdot y]\!]_{d_1+d_2}^{(q)} = [\![x]\!]_{d_1}^{(q)} \cdot [\![y]\!]_{d_2}^{(q)}$, defined as $\rho_z(t) := \rho_x(t) \cdot \rho_y(t)$ and $\gamma_z := \gamma_x \cdot \gamma_y$.

For succinctness, we will often use the notation $[\![\boldsymbol{x}]\!]_d^{(q)}$ to indicate a commitment to a vector $\boldsymbol{x} \in \mathbb{F}_q^\ell$, where every entry is committed to under the same global key Δ, with the same degree, and belongs to the same receiving party. Accordingly, we will use the notation $[\![\boldsymbol{c} \odot \boldsymbol{x}]\!]_d^{(q)} = \boldsymbol{c} \odot [\![\boldsymbol{x}]\!]_d^{(q)}$ to denote the componentwise product with the public vector $\boldsymbol{c} \in \mathbb{F}_q^\ell$. Finally we will use $[\![\boldsymbol{A} \cdot \boldsymbol{x}]\!]_d^{(q)} = \boldsymbol{A} \cdot [\![\boldsymbol{x}]\!]_d^{(q)}$ for the multiplication by a public matrix $\boldsymbol{A} \in \mathbb{F}_q^{m \times \ell}$, which is obtained by a sequence of affine functions and additions.

The functionality $\mathcal{F}_{\mathsf{sVOLE}}^{q,r}$ currently only allows to commit to random values, but it is not hard to see that this suffices to commit to any input. To specify this, and some other useful functions, we use the following helper commands:

- $[\![\boldsymbol{u}]\!]^{(q)} \leftarrow \mathsf{RandCommit}(\ell, q, r, \Delta)$: run `eval` of $\mathcal{F}_{\mathsf{sVOLE}}$ on fresh identifiers $\mathsf{id}_1, \ldots, \mathsf{id}_\ell$ to commit to $\boldsymbol{u} \xleftarrow{\$} \mathbb{F}_q^\ell$.
- $[\![\boldsymbol{x}]\!]^{(q)} \leftarrow \mathsf{Commit}(\boldsymbol{x}, q, r, \Delta)$ for $\boldsymbol{x} \in \mathbb{F}_q^\ell$: first run $\mathsf{RandCommit}(\ell, q, r, \Delta)$ to get $[\![\boldsymbol{u}]\!]^{(q)}$, then $\mathcal{P}_{i\oplus 1}$ sends $\boldsymbol{\delta} = \boldsymbol{x} - \boldsymbol{u}$ to $\mathcal{P}_i$, and both return $[\![\boldsymbol{x}]\!]^{(q)} := [\![\boldsymbol{u}]\!]^{(q)} + \boldsymbol{\delta}$
- $\mathsf{Prove}([\![x_1]\!]^{(q)}, \ldots, [\![x_\ell]\!]^{(q)}, C_1, \ldots, C_m)$: run `prove` of $\mathcal{F}_{\mathsf{sVOLE}}$ on the identifiers associated with $[\![x_j]\!]^{(q)}$, to prove that $C_k(x_1, \ldots, x_\ell) = 0$ for each $k = 1, \ldots, m$.
- $\mathsf{Open}([\![x]\!]^{(q)})$: $\mathcal{P}_{i\oplus 1}$ sends x to $\mathcal{P}_i$, run `prove` on the circuit $C : y \mapsto y - x$ to check the opening was correct.

When it is clear from context, we sometimes omit the (q, r, Δ) arguments from the commitment commands.

When performing many openings or proofs, it is more efficient to batch them together with a single call to the `prove` command. We implicitly intend that these calls are batched whenever openings and proofs are clearly parallelisable from the protocol description.

For $x \in \mathbb{F}_{q^r}$, we will moreover denote $[\![x]\!]^{(q^r)} \leftarrow \mathsf{Commit}(x, q^r, 1, \Delta)$ for committing to $\boldsymbol{x} := \mathsf{vec}_\alpha(x)$ as $[\![\boldsymbol{x}]\!]^{(q)} \leftarrow \mathsf{Commit}(\boldsymbol{x}, q, r, \Delta)$ and letting $[\![x]\!]^{(q^r)} := \mathsf{elt}_\alpha([\![\boldsymbol{x}]\!]^{(q)})$, which can be obtained through a sequence of affine functions and additions as defined above.

Similarly, the sender can use a specific $\Delta' \in \mathbb{F}_{q^r}$ to authenticate the receiver's commitment as follows. The parties first execute $((x, w), (\Delta, v)) =: [\![x]\!]^{(q)}_{\Delta} \leftarrow \mathsf{Commit}(x, q, r, \Delta)$ for an independent $\Delta \xleftarrow{\$} \mathbb{F}_{q^r}$, the sender sends $\tilde{\Delta} := \Delta' + \Delta$ to the receiver, and the parties define $[\![x]\!]^{(q)}_{\Delta'} := ((x, -w - \tilde{\Delta} \cdot x), (\Delta', -v))$. We will denote this procedure succinctly as $[\![x]\!]^{(q)}_{\Delta'} \leftarrow \mathsf{Commit}(x, q, r, \Delta \to \Delta')$.

For completeness, we present the Prove protocol (which works in the $\mathcal{F}_{\mathsf{sVOLE}}(\mathtt{init}, \mathtt{eval})$-hybrid model) in Fig. 3.

Fig. 3 Protocol $\mathsf{Prove}([\![x_1]\!]^{(q)}_{1,i}, \ldots, [\![x_\ell]\!]^{(q)}_{1,i}, C_1, \ldots, C_m)$

Parameters: Two parties, $\mathcal{P}_0$ and $\mathcal{P}_1$, holding $[\![x_1]\!]^{(q)}_{1,i}, \ldots, [\![x_\ell]\!]^{(q)}_{1,i}$, such that $\mathcal{P}_{i\oplus 1}$ holds the global key $\Delta \in \mathbb{F}_{q^r}$ associated to session sid of $\mathcal{F}^{q,r}_{\mathsf{sVOLE}}$ and $\mathcal{P}_i$ holds $x_1, \ldots, x_\ell \in \mathbb{F}_{q^r}$. Arithmetic circuits $C_1, \ldots, C_m : \mathbb{F}^\ell_{q^r} \to \mathbb{F}_{q^r}$ of degree $d_1, \ldots, d_m \in \mathbb{N}$, respectively. Base field $\mathbb{F}_q$ and extension field $\mathbb{F}_{q^r}$, integer $m \in \mathbb{N}$ and $d := \max\{d_1, \ldots, d_m\}$.

Subprotocol $\mathsf{BatchEval}([\![x_1]\!]^{(q)}_{1,i}, \ldots, [\![x_\ell]\!]^{(q)}_{1,i}, C_1, \ldots, C_m)$:

The parties evaluate the circuits $C_1, \ldots, C_m$ gate-by-gate on the commitments $[\![x_1]\!]^{(q)}_i, \ldots, [\![x_\ell]\!]^{(q)}_i$ to obtain the outputs $[\![z_1]\!]^{(q)}_{d_1,i}, \ldots, [\![z_m]\!]^{(q)}_{d_m,i}$. Here we assume the input commitments have degree 1, but this can readily be extended to higher degrees.

Subprotocol $\mathsf{BatchVer}([\![z_1]\!]^{(q)}_{d_1,i}, \ldots, [\![z_m]\!]^{(q)}_{d_m,i})$:

- The parties send $(\mathsf{sid}, \mathtt{eval}, r \cdot (d-1))$ to $\mathcal{F}^{q,r}_{\mathsf{sVOLE}}$, receive $[\![\boldsymbol{s}^j]\!]^{(q)}_{1,i}$ with $\boldsymbol{s}^j \in \mathbb{F}^r_q$ and $j \in [d-1]$ and compute $[\![s^j]\!]^{(q^r)}_{1,i} := \mathsf{elt}_\alpha([\![\boldsymbol{s}^j]\!]^{(q)}_{1,i})$
- $\mathcal{P}_{i\oplus 1}$ sends a challenge $(\chi_1, \ldots, \chi_m) \xleftarrow{\$} \mathbb{F}^m_{q^r}$.
- The parties compute $[\![z]\!]^{(q^r)}_{d,i} := \sum_{k=1}^m \chi_k \cdot [\![z_k]\!]^{(q)}_{d_k,i}$. $\mathcal{P}_i$ masks the corresponding polynomial as $\pi(t) := \rho_z(t) + \sum_{j=1}^{d-1} t^{j-1} \cdot \rho_{s^j}(t)$ and sends it to $\mathcal{P}_{i\oplus 1}$.
- $\mathcal{P}_{i\oplus 1}$ computes $\varepsilon := \gamma_z + \sum_{j=1}^{d-1} \Delta^{j-1} \cdot \gamma_{s^j}$ and checks if $\pi(\Delta) = \varepsilon$ and if π has degree at most $d-1$, and outputs 1 if all checks succeed, and 0 otherwise.

Lemma 1. *The protocol* Prove *in Figure 3 realizes the* `prove` *command of* $\mathcal{F}^{q,r}_{\mathsf{sVOLE}}$ *in the* $\mathcal{F}^{q,r}_{\mathsf{sVOLE}}(\mathtt{init}, \mathtt{eval})$*-hybrid model, assuming* $q^r \geq 2^\kappa$ *and* $d \in \mathsf{poly}(\kappa)$.

Proof. This follows from a straightforward combination of the proofs of Lemmas 2 and 3 from [1]. □

3.3 BDOZ-style 2PC

Note that two degree-1 VOLE commitments $[\![x_0]\!]^{(q)}_0$ and $[\![x_1]\!]^{(q)}_1$ with different receivers $\mathcal{P}_0$ and $\mathcal{P}_1$ give a secret sharing of $x := x_0 + x_1 \bmod q$ together with information-theoretic MACs $M_{x_i} =$

$\Delta_{i\oplus 1} \cdot x_i + K_{x_i}$, where $M_{x_i} := -\rho_{x_i,0}$ and $K_{x_i} = -\gamma_{x_i}$ on both of the shares $i \in \{0,1\}$. This enables us to marry techniques from VOLE-based ZK with BDOZ-style MPC [13]. For example, by proving consistency of locally generated daBits using VOLE-based ZK proofs for both parties, and afterwards combining these to a secret-shared (and uniformly random) daBits using authenticated multiplication triples.

We write $\langle\!\langle x \rangle\!\rangle^{(q)} = \big((x_i, M_{x_i}), (\Delta_i, K_{x_{i\oplus 1}})\big)_{i\in\{0,1\}}$ for an *authenticated* secret sharing where $\mathcal{P}_i$ holds $x_i, M_{x_i}, \Delta_i, K_{x_{i\oplus 1}}$ for $i \in \{0,1\}$. Such a sharing of $x := x_0 + x_1 \bmod q$ can be easily obtained by combining $[\![x_0]\!]_0^{(q)} = ((x_0, M_{x_0}), (\Delta_1, K_{x_0}))$ and $[\![x_1]\!]_1^{(q)} = ((x_1, M_{x_1}), (\Delta_0, K_{x_1}))$, which we denote as $\langle\!\langle x \rangle\!\rangle^{(q)} \leftarrow ([\![x_0]\!]_0^{(q)}, [\![x_1]\!]_1^{(q)})$. Moreover, for $i \in \{0,1\}$ we will denote $\langle\!\langle x \rangle\!\rangle_i^{(q)} \leftarrow ([\![x_i]\!]_i^{(q)}, x_{i\oplus 1})$ for the *half-authenticated* sharing of $x := x_0 + x_1 \bmod q$ where party $\mathcal{P}_i$ holds x_i together with a MAC M_{x_i}, and party $\mathcal{P}_{i\oplus 1}$ holds $x_{i\oplus 1}$ together with the key $(\Delta_{i\oplus 1}, K_{x_i})$ for the MAC on x_i. Note that if $x_{i\oplus 1} = 0$, such a half-authenticated sharing is equivalent to a (trivial) authenticated sharing of x_i, and so we will also denote $\langle\!\langle x_i \rangle\!\rangle^{(q)} \leftarrow ([\![x_i]\!]_i^{(q)}, 0)$. We will omit the superscript $^{(q)}$ when it is clear from the context.

As in [13], the parties can locally perform *linear* operations on shares that use the same global keys Δ_0, Δ_1. Using similar notation to $[\![\cdot]\!]$ sharings, we can denote this with:

$$\langle\!\langle c \cdot x + y + z \rangle\!\rangle^{(q)} = c \cdot \langle\!\langle x \rangle\!\rangle^{(q)} + \langle\!\langle y \rangle\!\rangle^{(q)} + z$$

for public constants $c, z \in \mathbb{F}_q$.

After evaluating operations of the above form, the parties can *reconstruct* or *open* values similarly to the opening procedure described in Section 3.2. That is, to open $\langle\!\langle x \rangle\!\rangle^{(q)}$, each party $\mathcal{P}_i$, $i \in \{0,1\}$ sends (x_i, M_{x_i}) to $\mathcal{P}_{i\oplus 1}$ who then checks if $M_{x_i} = \Delta_{i\oplus 1} \cdot x_i + K_{x_i}$, and reconstructs the shared value as $x = x_i + x_{i\oplus 1} \bmod q$. We will denote this procedure as $\mathsf{Open}(\langle\!\langle x \rangle\!\rangle^{(q)}, I)$ where $I \subseteq \{0,1\}$ specifies the parties to which the value is being opened.

When $\mathcal{P}_i$ has no information about $\Delta_{i\oplus 1} \in \mathbb{F}_q^{r^{(q)}}$, they can only try to guess a MAC $M_{x_i'}$ on an invalid share $x_i' \neq x_i$ with probability at most $1/|\mathbb{F}_q^{r^{(q)}}|$. That is, when $\Delta_{i\oplus 1} \cdot (x_i - x_i') = M_{x_i} - M_{x_{i'}}$. It therefore suffices to set $r^{(q)} := \kappa / \log_2 q$ to upper bound the cheating probability for an opening by $2^{-\kappa}$.

It is common to batch the opening checks of multiple values together and delay verification to the end of the protocol, similarly to the $\mathsf{BatchVer}$ subprotocol in Figure 3. That is, when values $\langle\!\langle x^j \rangle\!\rangle$, $j \in [m]$ are opened to party $\mathcal{P}_i$ during the protocol, party $\mathcal{P}_{i\oplus 1}$ only sends the shares $x_{i\oplus 1}^j$ to $\mathcal{P}_i$. At the end of the protocol, to verify all openings, $\mathcal{P}_i$ challenges $\mathcal{P}_{i\oplus 1}$ to open a random linear combination of the MACs. We will denote this procedure as $\mathsf{BatchOpen}\big((\langle\!\langle x^1 \rangle\!\rangle^{(q)}, \ldots, \langle\!\langle x^m \rangle\!\rangle^{(q)}), I\big)$.

To multiply two sharings, the standard approach is to use Beaver's trick [12], with a random *multiplication triple* of secret-shared values $(\langle\!\langle a \rangle\!\rangle^{(q)}, \langle\!\langle b \rangle\!\rangle^{(q)}, \langle\!\langle c \rangle\!\rangle^{(q)} = \langle\!\langle a \cdot b \rangle\!\rangle^{(q)})$. In this work, we will refer to the special case where a is known to $\mathcal{P}_0$ and b is known to $\mathcal{P}_1$ as an *authenticated OLE triple*, denoted $([\![a]\!]_0^{(q)}, [\![b]\!]_1^{(q)}, \langle\!\langle c \rangle\!\rangle^{(q)})$. We can use such a triple to multiply a sharing of the form

$$\langle\!\langle x \rangle\!\rangle^{(q)} := (((x, M_x), (\Delta_0, 0)), ((0,0), (\Delta_1, K_x))) \leftarrow ([\![x]\!]_0^{(q)}, 0)$$

with a sharing of the form

$$\langle\!\langle y \rangle\!\rangle^{(q)} := (((0,0), (\Delta_0, K_y)), ((y, M_y), (\Delta_1, 0))) \leftarrow (0, [\![y]\!]_1^{(q)}),$$

that is, where $\mathcal{P}_i$ knows x and $\mathcal{P}_{i\oplus 1}$ knows y. The protocol is essentially the same as Beaver's method. We denote this as $\langle\!\langle x \cdot y \rangle\!\rangle \leftarrow \mathsf{Mult}\,(([\![x]\!]_0, [\![y]\!]_1), ([\![a]\!]_0, [\![b]\!]_1, \langle\!\langle c \rangle\!\rangle))$. Our protocol for generating authenticated OLE triples over $\mathbb{F}_3$ is given in Section 5.2.

4 PRF Construction

The *"crypto dark matter"* or *"alternating moduli"* paradigm, initiated by Boneh et al. [15], explores constructing PRF candidates by mixing linear operations over small moduli. Their original weak PRF construction is given by

$$F_{\mathsf{weak}}^{\mathrm{BIPSW}}(\boldsymbol{A}, \boldsymbol{x}) := \boldsymbol{B} \cdot_3 (\boldsymbol{A} \cdot_2 \boldsymbol{x}), \tag{2}$$

with key $\boldsymbol{A} \in \mathbb{F}_2^{m \times n}$, input $\boldsymbol{x} \in \mathbb{F}_2^n$ and $\boldsymbol{B} \in \mathbb{F}_3^{t \times m}$ a random public matrix. Here we denote $\boldsymbol{A} \cdot_2 \boldsymbol{x} := \boldsymbol{A} \cdot \boldsymbol{x} \bmod 2$ and $\boldsymbol{B} \cdot_3 \boldsymbol{y} := \boldsymbol{B} \cdot \boldsymbol{y} \bmod 3$. The efficiency of the PRF can be improved by choosing the key $\boldsymbol{A}$ to be a structured matrix.

Circulant Matrix. A common choice from prior work [15, 25] is to let $m = n$ and choose $\boldsymbol{A}$ to be a random circulant matrix, where each row is a cyclic shift by one of the previous one. This means only one row of the matrix needs to be stored, and multiplication by $\boldsymbol{A}$ is equivalent to polynomial multiplication over $\mathbb{F}_2$ modulo $X^n - 1$. Past works have typically chosen (optimistic) parameters [3, 15, 25] for this variant as $m = n = 2\lambda$ and $t = \lambda / \log_2 3$. Unfortunately, it turns out these are too aggressive for λ-bit security.

Cheon et al. [21] show that security degrades in the special case where $\boldsymbol{B} = \mathbf{1}^{1 \times n}$ and $\boldsymbol{A} \in \mathbb{F}_2^{n \times n}$ is circulant, although their attack needs an exponential number of samples. It is moreover conjectured that when $\boldsymbol{B} \in \mathbb{F}_3^{t \times n}$ is chosen randomly and $t \ll n$, these attacks are unlikely to apply [25]. However, recent work [38] (concurrent to ours) showed that this does not hold and proposed an attack on the more general construction, with time $2^{0.69\lambda}$ and $2^{0.32\lambda}$ queries when $n = 2\lambda$. They conclude that resisting this attack would require increasing n to at least 2.63λ.

Component-wise Multiplication. Alternatively, Alamati et al. [2] propose to optimize performance within the dark-matter paradigm with the variant

$$F_{\mathsf{weak}}^{\mathrm{APRR}}(\boldsymbol{k}, \boldsymbol{x}) := \boldsymbol{B} \cdot_3 (\boldsymbol{A} \cdot_2 [\boldsymbol{k} \odot_2 \boldsymbol{x}]), \tag{3}$$

where $\odot_2$ denotes componentwise multiplication modulo 2, the key $\boldsymbol{k} \in \mathbb{F}_2^n$, input $\boldsymbol{x} \in \mathbb{F}_2^n$, and both $\boldsymbol{A} \in \mathbb{F}_2^{m \times n}$ and $\boldsymbol{B} \in \mathbb{F}_3^{t \times m}$ are public matrices. They suggest both a one-to-one parameter set $n = 4\lambda$, $m = 7.06\lambda$, $t = 2\lambda / \log_2 3$, as well as a many-to-one parameter set $n = 4\lambda$, $m = 2\lambda$, $t = \lambda / \log_2 3$. Note that the suggested input size for the one-to-one parameters was recently doubled after the cryptanalysis of [7] and that both the one-to-one as well as the many-to-one parameters will most likely have to be increased even more after the recent improved attack from [53]

Our Variant: Finite Field Multiplication. In this work, we consider a novel variant of the dark-matter PRF, where the key $k \in \mathbb{F}_2^n$ and input $x \in \mathbb{F}_2^n$ are interpreted as elements in the finite field $\mathbb{F}_{2^n}$ (under a suitable choice of irreducible polynomial f of degree n over $\mathbb{F}_2$ and a corresponding choice of root $\alpha \in \mathbb{F}_{2^n}$).

Our weak PRF F'_k is parametrized by a random matrix $\boldsymbol{B} \in \mathbb{F}_3^{t \times n}$. For a key $k \in \mathbb{F}_{2^n}$ and input $x \in \mathbb{F}_{2^n}$, we define[3]

$$F'_k(x) = \boldsymbol{B} \cdot_3 \mathsf{vec}_\alpha(k \cdot x). \tag{4}$$

Our main motivation with this variant was to obtain a weak-PRF that is compatible with VOLE protocols over $\mathbb{F}_{2^n}$. However, an additional benefit is that it also resists the recent attack on

[3] Additionally, one can choose another random public matrix $\boldsymbol{A} \in \mathbb{F}_2^{m \times n}$ and let $\boldsymbol{B} \in \mathbb{F}_3^{t \times m}$ to have another parameter $m \in \mathbb{N}$ to tune, similar to [2] and similar to [15, 25] using a non-square key matrix $\boldsymbol{A} \in \mathbb{F}_2^{m \times n}$. However, since it is common to choose $n = m$ [15, 25], and the key $k \in \mathbb{F}_{2^n}$ corresponds to a square matrix $\boldsymbol{K} \in \mathbb{F}_2^{n \times n}$ seen through its action on $\mathbb{F}_2^n$, we chose to use just one public matrix.

the circulant matrix variant [38]. Concretely, the attack relies on the shift-invariant property of circulant matrix multiplication, as well as the fact that a random circulant matrix is not full rank with constant probability. When working over a finite field, reduction by an irreducible polynomial instead of $X^n - 1$ breaks the symmetry of a circulant matrix, and furthermore, the matrix is always invertible if the key is non-zero. Additionally, the attacks from [7, 53] on the component-wise variant of [2] do not apply to our construction. That is, it crucially relies on querying the weak PRF on inputs $\boldsymbol{y}, \boldsymbol{y'} \in \mathbb{F}_2^n$ that differ in just one entry $y_i \neq y'_i$, which result in $F^{\text{APRR}}_{\mathsf{weak}}(\boldsymbol{k}, \boldsymbol{y}) = F^{\text{APRR}}_{\mathsf{weak}}(\boldsymbol{k}, \boldsymbol{y'})$, as defined in equation (3). This is indicative of the corresponding entry k_i of $\boldsymbol{k}$ being equal to zero since it implies that $\boldsymbol{k} \odot \boldsymbol{y} = \boldsymbol{k} \odot \boldsymbol{y'}$ with high probability. However, since our variant uses field multiplication instead of pointwise multiplication, it can *never* occur that $k \cdot y = k \cdot y'$ for $y \neq y'$. We therefore conjecture that choosing $n = 2\lambda$ suffices for λ-bit security of our variant.

4.1 From Weak to Strong PRF

Since a weak PRF F_{weak} is only guaranteed to be pseudorandom on random inputs, it is standard practice to first hash the inputs as

$$F_{\mathsf{strong}}(k, x) := F_{\mathsf{weak}}(k, \mathsf{H}(x)),$$

which is guaranteed to be a strong PRF when H is modelled as a random oracle. However, when the function is evaluated in a distributed fashion, and the receiver holding the input x can be maliciously corrupted, this would add significant overhead.

It was suggested in [15] to instead apply a random linear code $\boldsymbol{G} \in \mathbb{F}_3^{n' \times d}$ in systematic form $\boldsymbol{G} = [\boldsymbol{I}_d \mid \boldsymbol{G'}]^T$ to the input $\boldsymbol{x} \in \mathbb{F}_2^d$, followed by a binary decomposition operator $\mathsf{decomp} : \mathbb{F}_3^{n'} \to \mathbb{F}_2^{d+2(n'-d)}$, together sending

$$\boldsymbol{x} \mapsto \mathsf{decomp}(\boldsymbol{G} \cdot_3 \boldsymbol{x}) := \boldsymbol{x} \| \mathsf{bin}(\boldsymbol{G'}^T \cdot_3 \boldsymbol{x}).$$

The code is in systematic form to easily verify the original input is binary, and so the binary decomposition operation is only applied to the last $n' - d$ coordinates of $\boldsymbol{G} \cdot_3 \boldsymbol{x}$.

Additionally, it was suggested by [3] to append 1 to the result to ensure pseudorandomness on input $\boldsymbol{x} = \boldsymbol{0}$, obtaining the encoded input $\boldsymbol{y} = \mathsf{decomp}(\boldsymbol{G} \cdot_3 \boldsymbol{x}) \| 1 \in \mathbb{F}_2^n$.

Hence for $\boldsymbol{x} \in \mathbb{F}_2^d$ and $d + 2(n' - d) = n - 1$, our strong PRF construction is defined as

$$F_k(\boldsymbol{x}) := \mathbf{B} \cdot_3 \left[\mathsf{vec}_\alpha \left(k \cdot \mathsf{elt}_\alpha \left(\mathsf{decomp}(\boldsymbol{G} \cdot_3 \boldsymbol{x}) \| 1\right)\right)\right], \tag{5}$$

where a common choice [3] is setting $d = \lambda$.

To efficiently verify the input $\boldsymbol{y} = (\boldsymbol{x'} \| 1) \in \mathbb{F}_2^n$ is correctly encoded, one can use the parity check matrix $\boldsymbol{H} \in \mathbb{F}_3^{(n'-d) \times n'}$ of $\boldsymbol{G}$ together with the gadget matrix $\boldsymbol{G}_{\text{gadget}} = (2, 1) \otimes \boldsymbol{I}_{n-1}$ to check that

$$(\boldsymbol{H} \cdot \boldsymbol{G}_{\text{gadget}} \cdot \boldsymbol{x'} = \boldsymbol{0} \mod 3) \wedge (\boldsymbol{x'}_{[1:d]} \in \{0, 1\}^d). \tag{6}$$

We provide protocols evaluating our strong PRF $F_k(x)$ from (5) securely in the fully malicious setting in Section 6.1, as well as in the setting where the receiver is malicious and the sender is semi-honest in Section 6.2. In the semi-honest setting, it is sufficient to let the receiver apply a random oracle $x' \leftarrow \mathsf{H}(x)$ to their input before supplying it to a semi-honest protocol evaluating $F'_k(x')$ from (4). This approach is sketched in Section 6.3.

5 Building Blocks for Correlated Randomness

In this section, we present our protocols for the main building blocks used in our OPRF. These produce various forms of correlated randomness, which we model in our ideal preprocessing functionality $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$ in Figure 4. The functionality extends $\mathcal{F}_{\mathsf{sVOLE}}$, containing commands for two copies of $\mathcal{F}_{\mathsf{sVOLE}}$ over $\mathbb{F}_2$ and $\mathbb{F}_p$ for an odd prime p. This is similar to the mixed-field VOLE functionality from [1]. Whereas [1] gives efficient protocols realizing mixed-field VOLE in the setting where $p \geq \ell \cdot 2^\kappa$ or $p \geq \ell$, we focus on the setting where $p \ll \ell$, for which we give an efficient protocol in Section 5.1, which implements the local daBits command ① of Figure 4.

We furthermore extend the ideal mixed-field VOLE functionality to two-sided authenticated mixed-field correlations, or secret-shared daBits ②, which can be used for conversions in secure two-party computation. Additionally, Figure 4 contains commands for generating authenticated OLE ③ (used by our daBits protocol) and authenticated VOLE ④, used in our OPRF. We define the following shorthand notation to use in our protocols to commit to a chosen input $\boldsymbol{x} \in \mathbb{F}_{q^r}^\ell$:

– $\langle\!\langle \gamma \cdot \boldsymbol{x} \rangle\!\rangle^{(q^r)} \leftarrow \mathsf{aCommit}(\boldsymbol{x}, q^r, \gamma, \Delta_0, \Delta_1)$: The parties send $(\mathsf{sid}_{\mathsf{VOLE}}, \mathsf{sid}_{\mathsf{MAC}}, \mathtt{aVOLE}, q^r, \ell)$ to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$(④) to obtain $\langle\!\langle \gamma \cdot \boldsymbol{y} \rangle\!\rangle^{(q^r)} \leftarrow (\llbracket \boldsymbol{w} \rrbracket_{0,\mathsf{MAC}}^{(q^r)}, \llbracket \boldsymbol{v} \rrbracket_{1,\mathsf{MAC}}^{(q^r)})$ and $\llbracket \gamma \rrbracket_{1,\mathsf{MAC}}^{(q^r)}$. $\mathcal{P}_0$ sends $\boldsymbol{\delta} := \boldsymbol{x} - \boldsymbol{y}$ to $\mathcal{P}_1$. Return $\langle\!\langle \gamma \cdot \boldsymbol{x} \rangle\!\rangle^{(q^r)} \leftarrow (\llbracket \boldsymbol{w} \rrbracket_{0,\mathsf{MAC}}^{(q^r)}, \llbracket \boldsymbol{v} \rrbracket_{1,\mathsf{MAC}}^{(q^r)} + \boldsymbol{\delta} \odot \llbracket \gamma \rrbracket_{1,\mathsf{MAC}}^{(q^r)})$

We realize the ideal functionality incrementally and leverage the previously defined functionalities, where we for example denote $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}(①,②)$ for the ideal functionality consisting of the functionalities indicated by ① and ② in Figure 4. We prove simulation-based security of all our protocols in (a simplified version of) the UC framework. Some of our protocols also make use of a standard coin-tossing functionality, denoted $\mathcal{F}_{\mathsf{Random}}$, and a weak equality-test functionality from [55], denoted $\mathcal{F}_{\mathsf{EQ}}$. Details of these functionalities are in Appendix **??**.

5.1 Local Doubly-Authenticated Bits

We propose a new protocol that generates local doubly authenticated bits for a general prime p, where p is independent of the number of correlations ℓ. The high-level overview of the protocol is that the receiving party $\mathcal{P}_i$ commits to $M = B\ell + C$ random bits, both modulo 2 and modulo p, and proves to the other party that these are consistent. The consistency check follows a standard cut-and-choose approach where C random candidate daBits are opened and the remaining ones are mapped into ℓ buckets of size B, after which $\mathcal{P}_i$ proves consistency of the daBits in each bucket.

Looking ahead to Section 5.3, after proving correctness of the local daBits, two local daBits from different parties can be combined to obtain a *secret-shared* daBit. Note that this approach is fundamentally different from the original daBits protocol from [50], where consistency is only checked *after* combining two local daBits into a secret-shared daBit. This immediately saves a factor $B' \cdot (B' - 1)$ multiplication triples, where a typical bucket size for [50] is $B' = 3$ when generating a batch of $\geq 10^6$ daBits.

Furthermore, we propose a novel approach to check consistency of the daBits in each bucket using linear codes. Our idea is to use the parity-check matrix P of a $[B, k, d]$ binary linear code to check combinations of the daBits in each bucket. The intuition is that the modulo 2 committed bits $\llbracket b_1 \rrbracket_i^{(2)}, \ldots, \llbracket b_B \rrbracket_i^{(2)}$ and modulo p committed bits $\llbracket b'_1 \rrbracket_i^{(p)}, \ldots, \llbracket b'_B \rrbracket_i^{(p)}$ will only map to the same value if their difference is a valid codeword. Since the minimum distance of the code is d, this implies that they are either all equal or at least d of them are different. Under a suitable choice of parameters we can guarantee that the probability that at least d maliciously-chosen daBits end up in the same

⓪ **Subfield VOLE:** The ideal functionality adopts the `init`, `eval` and `prove` commands from the ideal functionalities $\mathcal{F}_{\mathsf{sVOLE}}^{2,r^{(2)}}$ and $\mathcal{F}_{\mathsf{sVOLE}}^{p,r^{(p)}}$ (Fig. 2). These are now suffixed with the modulus $q \in \{2, p\}$, e.g. $(\mathsf{sid}, \texttt{init}, i, q)$.

① **Local Doubly-Authenticated Bits:** Upon receiving $(\mathsf{sid}, \texttt{laBits}, \ell, i)$ from $\mathcal{P}_0$ and $\mathcal{P}_1$, $i \in \{0, 1\}$:
- Sample $\boldsymbol{b} \xleftarrow{\$} \{0,1\}^\ell$ and send the parties fresh VOLE commitments $([\![\boldsymbol{b}]\!]_i^{(2)}, [\![\boldsymbol{b}]\!]_i^{(p)})$ (where P_i gets $\boldsymbol{b}$)

② **Doubly-Authenticated Bits:** Upon receiving $(\mathsf{sid}, \texttt{daBits}, \ell)$ from $\mathcal{P}_0$ and $\mathcal{P}_1$:
- Sample $\boldsymbol{b} \xleftarrow{\$} \{0,1\}^\ell$.
- For each $q \in \{2, p\}$, interpret $\boldsymbol{b}$ as an element of $\mathbb{F}_q^\ell$, and sample a fresh secret-sharing $\langle\langle \boldsymbol{b} \rangle\rangle^{(q)}$.
- Output $(\langle\langle \boldsymbol{b} \rangle\rangle^{(2)}, \langle\langle \boldsymbol{b} \rangle\rangle^{(p)})$.

③ **Authenticated OLE:** Upon receiving $(\mathsf{sid}, \texttt{aOLE}, q, \ell)$ from $\mathcal{P}_0$ and $\mathcal{P}_1$:
- Sample $\boldsymbol{a}^0, \boldsymbol{a}^1 \xleftarrow{\$} \mathbb{F}_q^\ell$.
- Sample fresh authentications $[\![\boldsymbol{a}^0]\!]_0^{(q)}$ and $[\![\boldsymbol{a}^1]\!]_1^{(q)}$.
- Sample a secret-sharing $\langle\langle \boldsymbol{c} \rangle\rangle^{(q)}$, where $\boldsymbol{c} := \boldsymbol{a}^0 \circ \boldsymbol{a}^1$.
- Output $([\![\boldsymbol{a}^0]\!]_0^{(q)}, [\![\boldsymbol{a}^1]\!]_1^{(q)}, \langle\langle \boldsymbol{c} \rangle\rangle^{(q)})$

④ **Authenticated VOLE:** Upon receiving $(\mathsf{sid}_{\mathsf{VOLE}}, \mathsf{sid}_{\mathsf{MAC}}, \texttt{aVOLE}, q^r, \ell)$ from $\mathcal{P}_0$ and $\mathcal{P}_1$:
- Sample $\boldsymbol{y} \xleftarrow{\$} \mathbb{F}_{q^r}^\ell$.
- Sample an authentication $[\![\boldsymbol{y}]\!]_{0,\mathsf{VOLE}}^{(q^r)}$. That is, $\mathcal{P}_0$ obtains $\boldsymbol{w} \in \mathbb{F}_{q^r}^\ell$ and $\mathcal{P}_1$ obtains $\boldsymbol{v} \in \mathbb{F}_{q^r}^\ell$, satisfying $\boldsymbol{v} + \boldsymbol{w} = \gamma \cdot \boldsymbol{y}$.
- Similarly sample $[\![\gamma]\!]_{1,\mathsf{MAC}}^{(q^r)}$, $[\![\boldsymbol{v}]\!]_{1,\mathsf{MAC}}^{(q^r)}$ and $[\![\boldsymbol{w}]\!]_{0,\mathsf{MAC}}^{(q^r)}$.
- Output $\langle\langle \gamma \cdot \boldsymbol{y} \rangle\rangle^{(q^r)} \leftarrow ([\![\boldsymbol{w}]\!]_{0,\mathsf{MAC}}^{(q^r)}, [\![\boldsymbol{v}]\!]_{1,\mathsf{MAC}}^{(q^r)})$ and $[\![\gamma]\!]_{1,\mathsf{MAC}}^{(q^r)}$.

Special Abort: If $\mathcal{P}_1$ is corrupted:
- Receive $(\mathsf{sid}_{\mathsf{VOLE}}, \mathsf{sid}_{\mathsf{MAC}}, \texttt{abort}^*, q)$ from $\mathcal{A}$. Return Δ_0 to $\mathcal{P}_1$ and `abort` to both parties, and abort.

Special Guess: If $\mathcal{P}_0$ is corrupted:
- Receive $(\mathsf{sid}_{\mathsf{VOLE}}, \mathsf{sid}_{\mathsf{MAC}}, \texttt{guess}^*, (a, b, c), q)$ from $\mathcal{A}$ with $a, b, c \in \mathbb{F}_{q^r}$. If $c = a \cdot \Delta_1 + b \cdot (\Delta_1 \cdot \gamma)$, return `success` to $\mathcal{P}_0$. Otherwise, return `fail`.

Global-Key Query: If $\mathcal{P}_{i \oplus 1}$ is corrupted, receive $(\mathsf{sid}, \texttt{guess}, \Delta', q)$ from $\mathcal{A}$ with $q \in \{2, p\}$. If $\Delta' = \Delta_i^{(q)}$ and $\mathcal{P}_i$ is honest, return $(\texttt{success}, S_i)$ to $\mathcal{P}_{i \oplus 1}$, where S_i contains all the outputs of $\mathcal{P}_i$, otherwise return `fail`.

Corrupted Parties: In the above commands, a corrupted party may choose its own outputs; the functionality receives these and re-samples the honest parties' outputs according to the correct distribution. Details are omitted for brevity.

Fig. 4: Ideal preprocessing functionality $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$

bucket is small. By picking the parity-check matrix P in systematic form $P = [P'|I_{B-k}]$, it is clear to see that, for each row P_j, $j \in [B-k]$, the product $\bigoplus_{\nu=1}^{B} P_{j,\nu} \cdot b_\nu$ is randomized by the $(k+j)$-th daBit.

Hence, after verifying all the parity-check results, the first k daBits in each bucket are guaranteed to be consistent and remain uniformly random from the verifier's point of view, and the remaining $B-k$ bits are discarded (e.g., a fraction $5/12$ for $B = 12$, $k = 7$). Compared to [50], where $B' - 1$ daBits in each bucket are sacrificed (e.g., a fraction $2/3$ for $B' = 3$), our approach sacrifices about half as many local daBits, under a suitable choice of linear code.

The protocol $\Pi_{\mathsf{laBits}}^{p,r^{(2)},r^{(p)}}$ is shown in Figure 6. Suitable choices of linear codes can be found in Table 1. We instantiate the protocol with the $[12, 7, 4]$ linear code, meaning that buckets are of size 12 and we only need to sacrifice $12 - 7 = 5$ daBits per bucket, outputing the remaining 7. Finally, we use the protocol in Figure 5 to halve the degree of the opening check to verify the parity-checks of the modulo p daBits.

$[B, k, d]$	ZK degree	Min. batch size	Comm. per daBit
$[5, 2, 3]$	2	2^{20}	$2.5 \log_2 3 + \frac{3}{2} \approx 5.46$
$[6, 3, 3]$	2	2^{20}	$2 \log_2 3 + 1 \approx 4.17$
$[7, 4, 3]$	2	2^{20}	$\frac{7}{4} \log_2 3 + \frac{3}{4} \approx 3.52$
$[7, 3, 4]$	2	2^{13}	$\frac{7}{3} \log_2 3 + \frac{4}{3} \approx 5.03$
$[8, 4, 4]$	2	2^{13}	$2 \log_2 3 + 1 \approx 4.17$
$[12, 7, 4]$	3	2^{14}	$\frac{12}{7} \log_2 3 + \frac{5}{7} \approx 3.43$

Table 1: Minimum batch size and communication cost (in bits) for the new cut-and-choose protocol, with failure probability $\leq 2^{-40}$. Each size-B bucket is verified using $B-k$ parity checks, producing k correct daBits per bucket. The $[B, k, d]$ linear codes were found at https://codetables.de.

Fig. 5 $\Pi_{\mathsf{XORCheck}}^{p}(\llbracket x_1 \rrbracket^{(2)}, \llbracket y_1 \rrbracket^{(p)}, \ldots, \llbracket x_m \rrbracket^{(2)}, \llbracket y_m \rrbracket^{(p)})$

1. Compute $\llbracket c \rrbracket^{(2)} = \sum_{j=1}^{m} \llbracket x_j \rrbracket^{(2)}$
2. Run $\mathsf{Open}(\llbracket c \rrbracket_i^{(2)})$, so $\mathcal{P}_{i\oplus 1}$ learns c
3. Let $C : \mathbb{F}_p^m \to \mathbb{F}_p$ be a circuit that, on input $\boldsymbol{y}$, computes $z_L = (-1)^c \prod_{j=1}^{\lfloor m/2 \rfloor} (2y_j - 1)$ and $z_R = \prod_{j=\lfloor m/2 \rfloor + 1}^{m} (2y_j - 1)$, and outputs $z = z_L - z_R$.
4. Run $\mathsf{Prove}(\llbracket \boldsymbol{y} \rrbracket^{(p)}, C)$, to prove that $C(\boldsymbol{y}) = 0$

Proposition 1. *In $\Pi_{\mathsf{laBits}}^{p,r^{(2)},r^{(p)}}$ step 6, if all checks pass in a bucket containing at least one faulty daBit, then the bucket must have at least d faulty daBits*

Proof. Let $\boldsymbol{e} \neq 0 \in \{0,1\}^B$ be the vector indicating error positions for the daBits, in a bucket that contains at least one faulty daBit. If all checks are satisfied, we have $H\boldsymbol{e} = 0$. Since H is a parity-check matrix for a distance d code, it holds that $H\boldsymbol{e} \neq 0$ for all $\boldsymbol{e}$ with weight less than d. Therefore, there are at least d faulty daBits in the bucket. □

Lemma 2. *The adversary's overall success probability is small, for any number of corrupt laBits, $\hat{C}$.*

Fig. 6 Local daBits Protocol, $\Pi_{\mathsf{laBits}}^{p,r^{(2)},r^{(p)}}$

Parameters: Two parties $\mathcal{P}_0$ and $\mathcal{P}_1$, number of local daBits ℓ, bucketing parameters B, C such that $M = B\ell + C$. Public parity check matrix $P \in \mathbb{F}_2^{(B-k)\times B}$ for a $[B, k, d]$ linear code in systematic form $P = [P'|I_{B-k}]$ such that P' is full row rank. Let $P_j \subset \{1, \ldots, B\}$ be the index set of non-zero entries in j'th row of P. Let $w = \max\{|P_j|\}$ be the maximum row-weight of P. Base field characteristic p, and extension field parameters $r^{(2)}, r^{(p)}$ for $\mathbb{F}_2, \mathbb{F}_p$ respectively. $i \in \{0, 1\}$ indicating the receiving party.

Initialize: The parties send $(\mathsf{sid}_2, \mathtt{init}, i \oplus 1)$ to $\mathcal{F}_{\mathsf{sVOLE}}^{2,r^{(2)}}$ and $(\mathsf{sid}_p, \mathtt{init}, i \oplus 1)$ to $\mathcal{F}_{\mathsf{sVOLE}}^{p,r^{(p)}}$. $\mathcal{P}_{i\oplus 1}$ receives $\Delta^{(2)} \in \mathbb{F}_{2^{r^{(2)}}}$ and $\Delta^{(p)} \in \mathbb{F}_{p^{r^{(p)}}}$.

Generate Candidates:

1. The parties run $[\![\boldsymbol{b}]\!]_i^{(2)} \leftarrow \mathsf{RandCommit}(M, 2)$, so that $\mathcal{P}_i$ learns $\boldsymbol{b} \xleftarrow{\$} \{0,1\}^M$, and then run $[\![\boldsymbol{b}]\!]_i^{(p)} \leftarrow \mathsf{Commit}(\boldsymbol{b}, p)$.

Check Consistency

1. The parties run $\mathsf{Prove}([\![\boldsymbol{b}]\!]_i^{(p)}, C)$, with circuit $C(x) = x(1-x)$, to show that $\boldsymbol{b} \in \{0,1\}$.
2. $\mathcal{P}_{i\oplus 1}$ samples a subset $S \subset [M]$ of size C, and sends S to $\mathcal{P}_i$.
3. For $j \in S$ call $\tilde{b}_j \leftarrow \mathsf{Open}([\![b_j]\!]^{(2)})$.
4. Compute $[\![z_j]\!]_i^{(p)} := [\![b_j]\!]_i^{(p)} - \tilde{b}_j$ for $j \in S$ and check $1 \leftarrow \mathsf{BatchVer}(([\![z_j]\!]_i^{(p)})_{j\in S})$
5. $\mathcal{P}_{i\oplus 1}$ samples a random permutation π on $B\ell$ elements and sends π to $\mathcal{P}_i$. The parties use π to assign the remaining $B\ell$ elements $([\![b_j]\!]_i^{(2)}, [\![b_j]\!]_i^{(p)})_{j\in[M]\setminus S}$ into ℓ buckets of size B.
6. Let L be an empty list. For each bucket:
   - Relabel the elements of the bucket to $([\![b_1]\!]_i^{(2)}, [\![b_1]\!]_i^{(p)}), \ldots, ([\![b_B]\!]_i^{(2)}, [\![b_B]\!]_i^{(p)})$.
   - For $j = 1, \ldots, B-k$: the parties run $\Pi_{\mathsf{XORCheck}}^{p}(([\![b_\nu]\!]^{(2)}, [\![b_\nu]\!]^{(p)})_{\nu\in P_j})$
   - Append $([\![b_1]\!]_i^{(2)}, [\![b_1]\!]_i^{(p)}), \ldots, ([\![b_k]\!]_i^{(2)}, [\![b_k]\!]_i^{(p)})$ to L.
7. Output L.

Proof. From Proposition 1, we know that for $\mathcal{A}$ to cheat the XOR check, at least d corrupt laBits must be in all buckets with corruptions. Furthermore, these corruptions need to form a codeword, otherwise the check will fail. From this, we upper-bound the probability of $\mathcal{A}$ succeeding, by letting $\mathcal{A}$ win if every corrupt bucket contains at least d corruptions. We model this with the following balls-and-bins game:

1. There are $\ell B + C$ distinct balls and ℓ distinct buckets, each with B distinct slots that can hold one ball. $\hat{C}$ of the balls are labelled *bad*.
2. Pick C balls at random. If any of these is bad, output 0.
3. Otherwise, shuffle the remaining ℓB balls into the slots of the buckets.
4. If any bucket contains $\geq d$ bad balls, output 1. Otherwise, output 0.

Let m denote the number of buckets containing bad balls. For fixed m, we want to count the number of ways of assigning the $\hat{C}$ bad balls into these buckets following the constraints. First, assume the corrupted balls are indistinguishable. This is equivalent to counting the number of ways of throwing n indistinguishable balls into m buckets, such that each bucket has between 0 and $\delta - 1$ balls, where we have made the substitutions $n = \hat{C} - md$, $\delta = B - d + 1$. By the inclusion-exclusion principle[4] this is equal to:

$$c_{n,m,\delta} = \sum_{j\geq 0} (-1)^j \binom{m}{j} \binom{n - j\delta + m - 1}{m - 1}$$

[4] Similarly to e.g. https://math.stackexchange.com/questions/665178/, with a slight tweak as we allow for buckets with zero balls.

To take into account that these balls are not indistinguishable, we can assume the worst case, namely, they can be ordered in $\binom{B}{k}$ ways where $k = \max(d, \lfloor B/2 \rfloor)$. The m buckets can be selected in $\binom{\ell}{m}$ different ways, and the number of buckets must lie in the range $m \in [\lceil \hat{C}/B \rceil \dots \lfloor \hat{C}/d \rfloor]$. To account for the total number of ways to place $\hat{C}$ balls into ℓB slots, we normalise by a factor N of

$$N = \frac{(\ell B - \hat{C})!\hat{C}!}{(\ell B)!} = \frac{1}{\binom{\ell B}{\hat{C}}}$$

Lastly, given $\hat{C}$ bad balls, these must not be detected in the initial opening of C balls. This happens with probability

$$p_{C,\hat{C}} = \binom{B\ell + C - \hat{C}}{C} / \binom{B\ell + C}{C}$$

By independence of the two events, we have $\mathcal{A}$ wins with probability at most

$$p_{C,\hat{C}} \cdot N \cdot \sum_{m=\lceil \hat{C}/B \rceil}^{\lfloor \hat{C}/d \rfloor} \binom{\ell}{m} \binom{B}{\max(d, \lfloor B/2 \rfloor)}^{m} c_{\hat{C}-md,m,B-d+1}.$$

□

Remark 1 (Choosing Parameters). For given parameters (ℓ, B, d), bounding the above probability requires fixing a cut-and-choose parameter C such that for all adversarial choices of $\hat{C}$, the overall probability is sufficiently small. This is computationally expensive when the batch size ℓ is large. When choosing our parameters, shown in Table 1, we optimize the naive search using a few empirical observations:

- The adversary's optimal strategy is either to (1) choose $\hat{C} = d$ and hope that all d balls land in the same bucket, or (2) choose $\hat{C}$ very close to $\ell \cdot B$, hoping to flood the buckets such that every bucket has at least d bad balls.
- The second attack is prevented by choosing $\hat{C}$ large enough that the initial cut-and-choose check compensates the bucket check. In our experiments, $\hat{C} \approx 10$–20 was enough for our parameter choices, but we chose $\hat{C} = 128$ as a conservative option that covers all parameters.
- With $\ell = 2^{13}, 2^{14}$, we conducted an exhaustive search to verify that with $C = 128$, $\hat{C} = d$ is optimal for the adversary. For $\ell = 2^{20}$, the search space is too large and we relied on our heuristic analysis.

Theorem 1. *The protocol $\Pi_{\mathsf{laBits}}^{p,r^{(2)},r^{(p)}}$ in Figure 6 realizes the ideal functionality $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}(①)$ against malicious adversaries in the $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}(⓪)$-hybrid model with statistical small failure probability.*

Proof (sketch). When both parties are honest, correctness follows directly. When $\mathcal{P}_i$ is malicious, they can only cheat by committing to $b \oplus 1$ modulo p instead of b for some subset $\hat{C} \subset [M]$. The corruptions are either revealed during the C initial openings, or during the exclusive-or check. We show in Lemma 2 that the cheating can only go undetected with statistically small probability.

A malicious $\mathcal{P}_{i\oplus 1}$ has no way of influencing the output of $\mathcal{P}_i$, so we prove that he learns nothing about the output. Since P and P' are full rank matrices, any values opened by $\mathcal{P}_i$ during the exclusive-or check will be uniformly random and independent of all other opened values. Thus $\mathcal{P}_{i\oplus 1}$ cannot extract any information about the output. For the full proof see Appendix C.1. □

When amortizing the protocol over large ℓ, the communication is B/k $\mathbb{F}_p$ and $(B-k)/k$ $\mathbb{F}_2$ elements and uses B/k VOLE entries over $\mathbb{F}_2$ and $\mathbb{F}_p$ for each local daBit outputted.

5.2 Authenticated OLE

The authenticated OLE functionality $\mathcal{F}_{\mathsf{mRand}}^{3,r^{(2)},r^{(3)}}$ (③) for small fields can be realized similarly to the TinyOT-style approach for generating authenticated $\mathbb{F}_2$-OLE triples, as in [34]. We present our detailed protocol $\Pi_{\mathsf{aOLE}}^{3,r^{(3)}}$ in Appendix B (Fig. 21), which realizes $\mathcal{F}_{\mathsf{mRand}}^{3,r^{(2)},r^{(3)}}$ (③) in the $\mathcal{F}_{\mathsf{sVOLE}^-}^{3,r^{(3)}}$, $\mathcal{F}_{\mathsf{Random}}$-hybrid model.

The ideal functionality and proof are adapted to account for recent work [23] that highlights mistakes in previous proofs of TinyOT-style approaches. $\Pi_{\mathsf{aOLE}}^{3,r^{(3)}}$ is specifically designed for $p = 3$, as our OPRF utilizes this base field characteristic. If functionality $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$ is required for general p this protocol must be changed to work for general p. The protocol works using a cut-and-choose approach where an initial $M = B^2 \cdot \ell + C$ candidate authenticated OLE are generated, where most of them are sacrificed to ensure the consistency of the remaining ℓ. Amortized over large ℓ, the protocol has a total communication cost of $7B^2 - 3B - 1$ $\mathbb{F}_p$ elements and uses $4B^2$ VOLE entries over $\mathbb{F}_p$. For $\ell \geq 2^{16}$ the bucketing parameters are set to $(B, C) = (3, 3)$ to ensure statistical security of 2^{-40}. For smaller batch sizes, we simply increase B until we reach 2^{-40} probability of failure. For a single evaluation we end up at $B = 6$.

5.3 Doubly-Authenticated Bits

The doubly-authenticated bits (daBits) functionality $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$ (②) can be realized in a fairly straightforward way using the local doubly-authenticated bits and authenticated OLE functionalities $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$ (①, ③). The protocol computes the exclusive OR between the local daBits of each party, to form an authenticated secret sharing of the daBits. The local daBits over $\mathbb{F}_2$ can be trivially combined to form an authenticated secret sharing of the bit over $\mathbb{F}_2$. To produce authenticated shares over $\mathbb{F}_p$, the exclusive OR can be computed as $b_0 \oplus b_1 := b_0 + b_1 - 2b_0b_1 \bmod p$. We use an authenticated OLE to compute shares of b_0b_1, enabling the exclusive OR computation.

The crucial difference with the usual approach taken in previous work [6, 50] is that we prove correctness of the local daBits *before* combining them to a secret-shared daBit, as opposed to proving correctness *after* combining them, effectively requiring a factor $\approx 1/B$ authenticated OLE triples in comparison. We detail our protocol $\Pi_{\mathsf{daBits}}^{p,r^{(2)},r^{(p)}}$ in Figure 7, which realizes the doubly-authenticated bits functionality $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$ (②) in the $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$ (⓪, ①, ③)-hybrid model.

The protocol communicates 2 $\mathbb{F}_p$ elements, and uses 2 local daBits and 1 authenticated OLE per generated daBit.

Theorem 2. *The protocol $\Pi_{\mathsf{daBits}}^{p,r^{(2)},r^{(p)}}$ in Figure 7 realizes the doubly-authenticated bits functionality $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$ (②) in the $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$ (⓪, ①, ③)-hybrid model against malicious adversaries, assuming $r^{(p)} \geq \kappa / \log_2 p$.*

Proof (sketch). The parties receive consistent local daBits from the ideal preprocessing functionality, which directly give a secret sharing of a random bit modulo 2. Additionally, they receive random authenticated OLE triples from the ideal preprocessing functionality, which they can use to compute the secret sharing of this same bit modulo p by computing the XOR of the local daBits modulo p. The batch openings do not reveal any information about the local daBits $\boldsymbol{b}^0$, $\boldsymbol{b}^1$ since they are completely masked by the $\boldsymbol{a}^0$, $\boldsymbol{a}^1$ components of the OLE triple, respectively. See Appendix C.2 for the full proof. □

Fig. 7 Doubly-Authenticated Bits Protocol $\Pi_{\mathsf{daBits}}^{p,r^{(2)},r^{(p)}}$

Parameters: Two parties $\mathcal{P}_0$ and $\mathcal{P}_1$, number of Doubly-Authenticated Bits ℓ, base field characteristic p, and extension field parameters $r^{(2)}, r^{(p)}$ for $\mathbb{F}_2, \mathbb{F}_p$ respectively.

Initialization:

- The parties send $(\mathsf{sid}, \texttt{init}, 0, q)$ and $(\mathsf{sid}, \texttt{init}, 1, q)$ to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$ for each $q \in \{2, p\}$. $\mathcal{P}_0$ receives $\Delta_0^{(2)} \in \mathbb{F}_{2^{r^{(2)}}}$ and $\Delta_0^{(p)} \in \mathbb{F}_{p^{r^{(p)}}}$ and $\mathcal{P}_1$ receives $\Delta_1^{(2)} \in \mathbb{F}_{2^{r^{(2)}}}$ and $\Delta_1^{(p)} \in \mathbb{F}_{p^{r^{(p)}}}$
- The parties send $(\mathsf{sid}, \texttt{laBits}, \ell, 0)$ and $(\mathsf{sid}, \texttt{laBits}, \ell, 1)$ to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(3)}}$ and receive $[\![\boldsymbol{b}^0]\!]_0^{(2)}, [\![\boldsymbol{b}^0]\!]_0^{(p)}$ for $\boldsymbol{b}^0 \in \{0,1\}^\ell$ and $[\![\boldsymbol{b}^1]\!]_1^{(2)}, [\![\boldsymbol{b}^1]\!]_1^{(p)}$ for $\boldsymbol{b}^1 \in \{0,1\}^\ell$.
- The parties send $(\mathsf{sid}, \mathsf{aOLE}, p, \ell)$ to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(3)}}$ and receive $([\![\boldsymbol{a}^0]\!]_0^{(p)}, [\![\boldsymbol{a}^1]\!]_1^{(p)}, \langle\!\langle \boldsymbol{c} \rangle\!\rangle^{(p)})$, where $\boldsymbol{a}^0 \odot \boldsymbol{a}^1 = \boldsymbol{c} \bmod p$.

Combine `laBits`:

- The parties compute $[\![\boldsymbol{d}]\!]_0^{(p)} := [\![\boldsymbol{a}^0]\!]_0^{(p)} - [\![\boldsymbol{b}^0]\!]_0^{(p)}$ and $[\![\boldsymbol{e}]\!]_1^{(p)} := [\![\boldsymbol{a}^1]\!]_1^{(p)} - [\![\boldsymbol{b}^1]\!]_1^{(p)}$, and $\mathcal{P}_0$ opens $\boldsymbol{d} \in \mathbb{F}_p^\ell$ to $\mathcal{P}_1$ using $\mathsf{BatchOpen}\left([\![d_1]\!]_0^{(p)}, \ldots, [\![d_\ell]\!]_0^{(p)}\right)$ and $\mathcal{P}_1$ opens $\boldsymbol{e} \in \mathbb{F}_p^\ell$ to $\mathcal{P}_0$ using $\mathsf{BatchOpen}\left([\![e_1]\!]_1^{(p)}, \ldots, [\![e_\ell]\!]_1^{(p)}\right)$.
- The parties define the sharings $\langle\!\langle \boldsymbol{b} \rangle\!\rangle^{(2)} \leftarrow ([\![\boldsymbol{b}_0]\!]_0^{(2)}, [\![\boldsymbol{b}_1^{(2)}]\!]_1)$ and, for $i \in \{0,1\}$, $\langle\!\langle \boldsymbol{b}^i \rangle\!\rangle^{(p)} \leftarrow ([\![\boldsymbol{b}^i]\!]_i^{(p)}, 0)$, $\langle\!\langle \boldsymbol{a}^i \rangle\!\rangle^{(p)} \leftarrow ([\![\boldsymbol{a}^i]\!]_i^{(p)}, 0)$.
- The parties compute $\langle\!\langle \boldsymbol{b}^0 \odot \boldsymbol{b}^1 \rangle\!\rangle^{(p)} = \boldsymbol{e} \odot \boldsymbol{d} - \boldsymbol{e} \odot \langle\!\langle \boldsymbol{a}^0 \rangle\!\rangle^{(p)} - \boldsymbol{d} \odot \langle\!\langle \boldsymbol{a}^1 \rangle\!\rangle^{(p)} + \langle\!\langle \boldsymbol{c} \rangle\!\rangle^{(p)}$ and $\langle\!\langle \boldsymbol{b} \rangle\!\rangle^{(p)} = \langle\!\langle \boldsymbol{b}^0 \rangle\!\rangle^{(p)} + \langle\!\langle \boldsymbol{b}^1 \rangle\!\rangle^{(p)} - 2 \cdot \langle\!\langle \boldsymbol{b}^0 \odot \boldsymbol{b}^1 \rangle\!\rangle^{(p)}$.
- The parties output $(\langle\!\langle \boldsymbol{b} \rangle\!\rangle^{(2)}, \langle\!\langle \boldsymbol{b} \rangle\!\rangle^{(p)})$.

Remark 2. The protocol $\Pi_{\mathsf{daBits}}^{p,r^{(2)},r^{(p)}}$ can be optimized for our OPRF construction. As the application of the $\boldsymbol{B}$ matrix is linear, the sender will never verify the values computed by the receiver after the moduli conversion. This allows the daBits to only be half-authenticated over $\mathbb{F}_p$. This optimization propagates to the authenticated OLE protocol, where the receiver does not need to commit to its share of the products. Overall, this reduces communication of the authenticated OLE protocol by $B^2 + B \cdot (B-1)$ $\mathbb{F}_p$ elements and reduces the number of VOLE entries by B^2 per authenticated OLE.

5.4 Authenticated VOLE

One of the main building blocks we need for our OPRF is the ability to multiply a key $k \in \mathbb{F}_{2^n}$ held by the sender $\mathcal{P}_1$ with a vector of inputs $\boldsymbol{y} := (y^1, \ldots, y^\ell) \in \mathbb{F}_{2^n}^\ell$ held by the receiver $\mathcal{P}_0$ such that the parties obtain authenticated secret-shares of the product $k \cdot \boldsymbol{y}$ under their respective MAC keys $\Delta_0^{(2)}, \Delta_1^{(2)}$. We formalize this primitive by an ideal *authenticated VOLE* functionality $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}(④)$ in Figure 4 and provide a protocol realizing this functionality in Figure 8. We present the protocol $\Pi_{\mathsf{aVOLE}}^{q,r}$ for general $q \in \{2, p\}$, which is more general than what we need for our OPRF protocol in Section 6, but might be of independent interest for constructing maliciously-secure MPC protocols based on VOLE.

Theorem 3. *The protocol $\Pi_{\mathsf{aVOLE}}^{q,r}$ in Figure 8 realizes the ideal functionality $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}(④)$ against malicious adversaries in the $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}(⓪)$-, $\mathcal{F}_{\mathsf{EQ}}$-hybrid model, assuming $q^r \geq 2^\kappa$.*

Proof (sketch). When both parties behave honestly correctness is simple. The commitment to $(\boldsymbol{y} \| s)$ gives the parties secret shares of $\boldsymbol{y} \cdot \gamma$. All the other commitments are used to ensure correct

Fig. 8 Authenticated VOLE protocol $\Pi^{q,r}_{\mathsf{aVOLE}}$

Parameters: Two parties, the receiver $\mathcal{P}_0$ and the sender $\mathcal{P}_1$, length of authenticated VOLE correlation ℓ, base field characteristic $q \in \{2, p\}$ and extension field parameter $r = r^{(q)}$. $\mathsf{H} : \{0,1\}^* \to \{0,1\}^\lambda$ is a random oracle.

Initialization:

- Send $(\mathsf{sid}_{\mathsf{MAC}}, \mathtt{init}, 1, q)$, $(\mathsf{sid}_{\mathsf{VOLE}}, \mathtt{init}, 1, q)$ and $(\mathsf{sid}_{\mathsf{Mask}}, \mathtt{init}, 1, q)$ to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$. $\mathcal{P}_1$ receives Δ_1, γ, $\Delta_{\mathsf{Mask}} \in \mathbb{F}_{q^r}$.
- Send $(\mathsf{sid}_{\mathsf{MAC}}, \mathtt{init}, 0, q)$ to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$. $\mathcal{P}_0$ receives $\Delta_0 \in \mathbb{F}_{q^r}$.
- Let $[\![\Delta_0]\!]^{(q^r)}_{0,\mathsf{VOLE}} \leftarrow \mathsf{Commit}(\Delta_0, q^r, 1, \gamma)$, using $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$ with $\mathcal{P}_0$ as receiver. That is, $\mathcal{P}_0$ obtains $a \in \mathbb{F}_{q^r}$ and $\mathcal{P}_1$ obtains $b \in \mathbb{F}_{q^r}$ such that $a + b = \Delta_0 \cdot \gamma$.
- Let $[\![\gamma]\!]^{(q^r)}_{1,\mathsf{MAC}} \leftarrow \mathsf{Commit}(\gamma, q^r, 1, \Delta_0)$, using $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$ with $\mathcal{P}_1$ as receiver. That is $\mathcal{P}_0$ obtains $c \in \mathbb{F}_{q^r}$ and $\mathcal{P}_1$ obtains $d \in \mathbb{F}_{q^r}$ such that $c + d = \gamma \cdot \Delta_0$.
- $\mathcal{P}_0$ sends $(\mathsf{sid}, \mathtt{check}, a - c, 0)$ and $\mathcal{P}_1$ sends $(\mathsf{sid}, \mathtt{check}, d - b, 0)$ to $\mathcal{F}_{\mathsf{EQ}}$. If any of the parties receive `abort` from $\mathcal{F}_{\mathsf{EQ}}$, they abort. Otherwise, they continue.

Generating Candidate Correlation:

- Query $[\![(\boldsymbol{y}\|s)]\!]^{(q^r)}_{0,\mathsf{VOLE}} \leftarrow \mathsf{RandCommit}(\ell+1, q^r, 1, \gamma)$ to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$ with $\mathcal{P}_0$ as receiver. That is, $\mathcal{P}_0$ obtains $(\boldsymbol{y}\|s), (\boldsymbol{w}\|e) \in \mathbb{F}_{q^r}^{\ell+1}$ and $\mathcal{P}_1$ obtains $(\boldsymbol{v}\|f) \in \mathbb{F}_{q^r}^{\ell+1}$, satisfying $(\boldsymbol{v}\|f) + (\boldsymbol{w}\|e) = \gamma \cdot (\boldsymbol{y}\|s)$.
- Query $\mathsf{Commit}((\boldsymbol{y}\|s), q^r, 1, \Delta_{\mathsf{Mask}} \to \Delta_1 \cdot \gamma)$ to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$ with $\mathcal{P}_0$ as receiver. That is, $\mathcal{P}_0$ obtains $(\bar{\boldsymbol{w}}\|\bar{e}) \in \mathbb{F}_{q^r}^{\ell+1}$ and $\mathcal{P}_1$ obtains $(\bar{\boldsymbol{v}}\|\bar{f}) \in \mathbb{F}_{q^r}^{\ell+1}$, satisfying $(\bar{\boldsymbol{w}}\|\bar{e}) + (\bar{\boldsymbol{v}}\|\bar{f}) = (\Delta_1 \cdot \gamma) \cdot (\boldsymbol{y}\|s)$.
- Query $[\![\Delta_0 \cdot \boldsymbol{y}]\!]^{(q^r)}_{0,\mathsf{VOLE}} \leftarrow \mathsf{Commit}(\Delta_0 \cdot \boldsymbol{y}, q^r, 1, \gamma)$ to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$ with $\mathcal{P}_0$ as receiver. That is, $\mathcal{P}_0$ obtains $\tilde{\boldsymbol{w}} \in \mathbb{F}_{q^r}^{\ell}$ and $\mathcal{P}_1$ obtains $\tilde{\boldsymbol{v}} \in \mathbb{F}_{q^r}^{\ell}$ satisfying $\tilde{\boldsymbol{w}} + \tilde{\boldsymbol{v}} = \gamma \cdot (\Delta_0 \cdot \boldsymbol{y})$. Note that this implies that $(\tilde{\boldsymbol{w}} - \Delta_0 \cdot \boldsymbol{w}) + \tilde{\boldsymbol{v}} = \Delta_0 \cdot \boldsymbol{v}$.

Consistency Checks:

- Run $\mathsf{Prove}([\![\Delta_0]\!]^{(q^r)}_{1,0,\mathsf{VOLE}}, [\![\boldsymbol{y}]\!]^{(q^r)}_{1,0,\mathsf{VOLE}}, [\![\Delta_0 \cdot \boldsymbol{y}]\!]^{(q^r)}_{1,0,\mathsf{VOLE}}, C_1, \ldots, C_\ell)$ using $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$, where the circuits $C_j : \mathbb{F}_{q^r}^{2\ell+1} \to \mathbb{F}_{q^r}$ are given by $C_j(X_1, \ldots, X_{2\ell+1}) := X_1 \cdot X_{j+1} - X_{\ell+j+1}$. That is, $\mathcal{P}_0$ proves to $\mathcal{P}_1$ that $[\![\Delta_0]\!]^{(q^r)}_{1,0,\mathsf{VOLE}} \cdot [\![\boldsymbol{y}]\!]^{(q^r)}_{1,0,\mathsf{VOLE}} - [\![\Delta_0 \cdot \boldsymbol{y}]\!]^{(q^r)}_{1,0,\mathsf{VOLE}}$ opens to $\mathbf{0}$.
- Query $[\![(\boldsymbol{w}\|e)]\!]^{(q^r)}_{0,\mathsf{MAC}} \leftarrow \mathsf{Commit}((\boldsymbol{w}\|e), q^r, 1, \Delta_1)$ to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$ with $\mathcal{P}_0$ as receiver. That is, $\mathcal{P}_0$ obtains $(\boldsymbol{t}\|\tau) \in \mathbb{F}_{q^r}^{\ell+1}$ and $\mathcal{P}_1$ obtains $(\boldsymbol{u}\|\sigma) \in \mathbb{F}_{q^r}^{\ell+1}$ satisfying $(\boldsymbol{t}\|\tau) + (\boldsymbol{u}\|\sigma) = \Delta_1 \cdot (\boldsymbol{w}\|e)$.
- $\mathcal{P}_0$ computes $\tilde{t} := \mathsf{H}\left((\bar{\boldsymbol{w}}\|\bar{e}) - (\boldsymbol{t}\|\tau)\right)$ and sends $(\mathsf{sid}, \mathtt{check}, \tilde{t}, 0)$, $\mathcal{P}_1$ computes $\tilde{u} := \mathsf{H}\left((\boldsymbol{u}\|\sigma) - (\bar{\boldsymbol{v}}\|\bar{f}) + \Delta_1 \cdot (\boldsymbol{v}\|f)\right)$ and sends $(\mathsf{sid}, \mathtt{check}, \tilde{u}, 0)$ to $\mathcal{F}_{\mathsf{EQ}}$. If any of the parties receive `abort` from $\mathcal{F}_{\mathsf{EQ}}$, they abort. Otherwise, they continue.
- Output $[\![\boldsymbol{w}]\!]^{(q^r)}_{0,\mathsf{MAC}} := ((\boldsymbol{w}, \bar{\boldsymbol{w}}), (\Delta_1, \Delta_1 \cdot \boldsymbol{v} - \bar{\boldsymbol{v}}))$ and $[\![\boldsymbol{v}]\!]^{(q^r)}_{1,\mathsf{MAC}} := ((\Delta_0, \Delta_0 \cdot \boldsymbol{w} - \tilde{\boldsymbol{w}}), (\boldsymbol{v}, \tilde{\boldsymbol{v}}))$.

authentication of these secret shares. When either $\mathcal{P}_0$ or $\mathcal{P}_1$ are malicious the different checks verify different parts of the protocol.

The first call to $\mathcal{F}_{\mathsf{EQ}}$ during initialization ensures that $\mathcal{P}_0$ commits to the same Δ_0 as it receives from $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$. The `prove` call checks that $\mathcal{P}_0$'s commitment to $\Delta_0 \cdot \boldsymbol{y}$ is consistent with the $\boldsymbol{y}$ and Δ_0 they previously committed to. Together with the first equality check, this guarantees that $\tilde{\boldsymbol{v}}$ will be a valid MAC on $\boldsymbol{v}$ under the receiver's key Δ_0.

The second call to $\mathcal{F}_{\mathsf{EQ}}$ verifies multiple values at the same time. First it verifies that $\mathcal{P}_0$ commits to the same $\boldsymbol{w}$ they received from the initial commitment to $\boldsymbol{y}$. Secondly, appending s to $\boldsymbol{y}$ and e to $\boldsymbol{w}$ ensures that $\mathcal{P}_1$ cannot derandomize Δ_{Mask} to anything other than $\Delta_1 \cdot \gamma$ since otherwise the check fails. These checks together ensure that $\bar{\boldsymbol{w}}$ will be a valid MAC on $\boldsymbol{w}$ under the sender's key Δ_1.

The `abort`* and `guess`* calls are used during the simulation of the two calls to $\mathcal{F}_{\mathsf{EQ}}$. The special abort simulates the leakage of Δ_0 if $\mathcal{P}_1$ commits to $\gamma' \neq \gamma$ during initialization. The special guess simulates the possibility for $\mathcal{P}_0$ to guess a random linear combination of Δ_1 and $\Delta_1 \cdot \gamma$ to make the second $\mathcal{F}_{\mathsf{EQ}}$ call succeed on invalid inputs.

The fully detailed proof can be found in Appendix C.3.

□

Authenticating a VOLE of large enough length ℓ requires communicating an amortized 4 $\mathbb{F}_{q^r}$ elements and 4 VOLE entries over $\mathbb{F}_{q^r}$ for each authenticated VOLE entry.

Remark 3. Note that the OPRF protocol in Figure 9 can actually make use of a simplified variant of the consistency check in Figure 8. That is, the receiver only commits to e instead of $(\boldsymbol{w}\|e)$ and the parties only check that $\bar{e} - \tau \stackrel{?}{=} \sigma - \bar{f} + \Delta_1 \cdot f$ to guarantee that the sender actually uses $\Delta_1 \cdot \gamma$ for the commitment to $(\boldsymbol{y}\|s)$. With this check the authenticated VOLE protocol by itself does not guarantee that $\bar{\boldsymbol{w}}$ is a valid MAC on $\boldsymbol{w}$, but in order to obtain a valid MAC on a different value $\boldsymbol{w}'$, the receiver would need to guess $\Delta_1 \cdot \gamma$, which happens with negligible probability. This faulty MAC will be noticed by the sender in the OPRF protocol in Figure 9 during the opening of the c^j. So this optimization does not affect security and reduces communication cost.

6 OPRF Protocols

With the building blocks from Section 5 in hand, we are now ready to construct our maliciously-secure protocol for obliviously evaluating the PRF construction introduced in Section 4. Our discussion focusses on the fully-malicious setting, where both the sender as well as the receiver can be maliciously corrupted, which we elaborate in Section 6.1. We prove simulation-based security of our protocol in the UC framework. We additionally sketch protocols for:

(1) The setting where the sender is semi-honest and the receiver is malicious in Section 6.2;
(2) The semi-honest setting in Section 6.3;
(3) The setting where the inputs as well as the key are secret-shared in an authenticated way, and the parties obtain authenticated shares of the PRF output, in Section 6.4;
(4) The verifiable setting, where receivers can verify whether the sender uses the same key across different executions, in Section 6.5.

6.1 Malicious Protocol

Since the malicious receiver does not need to follow the protocol, the OPRF protocol requires the receiver to provide proof that its input follows the input encoding. Recall from Section 4 that the

encoded input is produced by a binary decomposition of a random linear code applied to the input. As the random linear code, $\boldsymbol{G} \in \mathbb{F}_3^{n' \times d}$ is in systematic form and $\boldsymbol{x} \in \mathbb{F}_2^d$, the first d entries of $\boldsymbol{G} \cdot_3 \boldsymbol{x}$ will be in $\{0, 1\}$. Therefore, the binary decomposition only needs to be applied to the last $n' - d$ entries. The sender can check that the input provided is a valid encoding by first applying $\boldsymbol{G}_{\text{gadget}}$ to inverse the binary decomposition and then applying the parity check matrix $\boldsymbol{H}$. By this construction, we have

$$\boldsymbol{H} \cdot \boldsymbol{G}_{\text{gadget}} \cdot \boldsymbol{x'} = \boldsymbol{0} \iff \exists \boldsymbol{x} : \mathsf{decomp}(\boldsymbol{G} \cdot_3 \boldsymbol{x}) = \boldsymbol{x'}.$$

To ensure that the 0 input, i.e, $\boldsymbol{x} = \boldsymbol{0}$, produces a random output a 1 is appended to $\boldsymbol{x'}$. Thus the input to the weak PRF will be $\boldsymbol{y} := (\boldsymbol{x'}||1)$.

Proving in zero-knowledge that the input is a valid encoding, requires a little more tooling, since $\boldsymbol{y} \in \mathbb{F}_2^n$ and the parity check must be done over $\mathbb{F}_3$. Using a commitment to $\boldsymbol{y}$ and n local daBits from $\mathcal{F}_{\mathsf{mRand}}^{3,r^{(2)},r^{(p)}}$ (①), they can convert the input from $\mathbb{F}_2^n$ to $\mathbb{F}_3^n$. From here the encoding can be shown to be valid efficiently by using the linearity of $\boldsymbol{G}_{\text{gadget}}$ and $\boldsymbol{H}$. Lastly, the last bit of the input is checked to be equal to 1, proving a valid input encoding.

As the sender now has proof that the input is encoded correctly, the weak PRF can be evaluated. This requires three main steps, 1) field multiplication, 2) modular conversion, and 3) matrix multiplication. To ensure that any malicious behaviour is detected, all three steps must be authenticated. Computing the field multiplication $k \cdot \mathsf{elt}_\alpha(\boldsymbol{y})$ in an oblivious way can be done using a VOLE over $\mathbb{F}_{2^n}$ resulting in the output of the multiplication being secret shared between the parties. Using $\mathcal{F}_{\mathsf{mRand}}^{3,r^{(2)},r^{(p)}}$ (④) further ensures that these shares are authenticated.

Using n daBits from $\mathcal{F}_{\mathsf{mRand}}^{3,r^{(2)},r^{(p)}}$ (②) allows the two parties to convert the secret shares of $k \cdot \mathsf{elt}_\alpha(\boldsymbol{y})$ modulo 2 to shares modulo 3. Since the daBits are authenticated, the shares remain authenticated after conversion. Both the sender and receiver can now multiply the public matrix $\boldsymbol{B}$ with their shares locally. Finally, the sender will open its shares to the receiver, which enables the receiver to check if the sender behaved honestly, and learn the output $F_k(\boldsymbol{x})$.

Theorem 4. *The protocol $\Pi_{\mathsf{OPRF}}^{\mathsf{M}}$ in Figure 9 realizes the ideal $\mathcal{F}_{\mathsf{OPRF}}$ functionality for the strong PRF family F_k from (5) against a malicious receiver and a malicious sender in the $\mathcal{F}_{\mathsf{mRand}}^{3,r^{(2)},r^{(3)}}$ (⓪,①,②,④)-hybrid model, assuming $n = d + 2(n' - d) + 1$, $r^{(2)} = n \geq \kappa$ and $r^{(3)} \geq \kappa / \log_2 3$.*

Proof (sketch). Correctness when both parties behave honestly follows directly from the construction. The field multiplication is computed using a VOLE, the conversion of moduli is achieved using daBits and the final output is computed by applying $\boldsymbol{B}$ and revealed to the receiver by opening of the secret sharing.

The simulator for malicious $\mathcal{P}_0$ is relatively straightforward. Only three parts need special care to prove the theorem. First, the input of $\mathcal{P}_0$ can be extracted from the input encoding since the matrix $\boldsymbol{G}$ is in systematic form. This allows the simulator to compute consistent OPRF outputs by forwarding the extracted inputs to $\mathcal{F}_{\mathsf{OPRF}}$. Secondly the batch opening requires the simulator to simulate sending the sender's share of c. As this is masked by the sender's daBit share, c is uniformly random from the point of view of the adversary $\mathcal{A}$ and the environment $\mathcal{Z}$, hence it can just be sampled randomly by the simulator. Lastly the output shares of the simulator must be consistent with the output. The simulator can compute the share of $\mathcal{P}_0$, and knows the OPRF output from $\mathcal{F}_{\mathsf{OPRF}}$, thus can trivially compute the expected share of the honest party. The simulator for a malicious $\mathcal{P}_1$ is very similar, except the input and output does not need to be handled since $\mathcal{A}$ never learns these.

Fig. 9 Maliciously-Secure OPRF protocol $\Pi^{\mathsf{M}}_{\mathsf{OPRF}}$

Parameters: Two parties $\mathcal{P}_0$ and $\mathcal{P}_1$, also referred to as the client and the server, respectively. The client holds a batch of inputs $(\boldsymbol{x}^j)_{j\in[\ell]} \in (\mathbb{F}_2^d)^\ell$. Public matrix $\boldsymbol{B} \in \mathbb{F}_3^{t\times n}$, linear code $\boldsymbol{G} \in \mathbb{F}_3^{n'\times d}$ with corresponding parity-check matrix $\boldsymbol{H} \in \mathbb{F}_3^{(n'-d)\times n'}$, gadget matrix $\boldsymbol{G}_{\mathrm{gadget}} = (2,1)\otimes \boldsymbol{I}_{n-1}$, integers $n, t, d, n', r^{(2)}, r^{(3)} \in \mathbb{N}$ depending on the security parameter $\lambda \in \mathbb{N}$ with $r^{(2)} = n$.

Preprocessing:

- Send $(\mathsf{sid}_{\mathsf{MAC}}, \mathtt{init}, 1, q)$ and $(\mathsf{sid}_{\mathsf{Key}}, \mathtt{init}, 1, q)$ to $\mathcal{F}_{\mathsf{mRand}}^{3,r^{(2)},r^{(3)}}$ for each $q \in \{2,3\}$. $\mathcal{P}_1$ receives $\Delta_1^{(2)} \in \mathbb{F}_{2^{r^{(2)}}}$, $\Delta_1^{(3)} \in \mathbb{F}_{3^{r^{(3)}}}$ and $k \in \mathbb{F}_{2^n}$, $\Delta_k^{(3)} \in \mathbb{F}_{3^{r^{(3)}}}$, respectively.
- Send $(\mathsf{sid}_{\mathsf{MAC}}, \mathtt{init}, 0, q)$ to $\mathcal{F}_{\mathsf{mRand}}^{3,r^{(2)},r^{(3)}}$ for each $q \in \{2,p\}$. $\mathcal{P}_0$ receives $\Delta_0^{(2)} \in \mathbb{F}_{2^{r^{(2)}}}$ and $\Delta_0^{(3)} \in \mathbb{F}_{3^{r^{(3)}}}$.
- Send $(\mathsf{sid}_{\mathsf{Key}}, \mathtt{laBits}, n\cdot\ell, 0)$ and $(\mathsf{sid}_{\mathsf{MAC}}, \mathtt{daBits}, n\cdot\ell)$ to $\mathcal{F}_{\mathsf{mRand}}^{3,n,r^{(3)}}$ and receive $[\![\boldsymbol{b'}^j]\!]^{(2)}_{0,\mathsf{Key}}$, $[\![\boldsymbol{b'}^j]\!]^{(3)}_{0,\mathsf{Key}}$ and $\langle\!\langle \boldsymbol{b}^j\rangle\!\rangle^{(2)}$, $\langle\!\langle \boldsymbol{b}^j\rangle\!\rangle^{(3)}$, respectively, for $\boldsymbol{b'}^j, \boldsymbol{b}^j \in \{0,1\}^n$, $j \in [\ell]$.

Key-Multiplication and Input-Encoding:

- $\mathcal{P}_0$ computes $\boldsymbol{x'}^j := \mathsf{decomp}(\boldsymbol{G}\cdot_3 \boldsymbol{x}^j) \in \mathbb{F}_2^{d+2(n'-d)}$ and $y^j := \mathsf{elt}_\alpha(\boldsymbol{x'}^j \| 1) \in \mathbb{F}_{2^n}$ for each $j \in [\ell]$.
- Query $\mathsf{aCommit}(y^j, 2^n, 1, k, \Delta_0^{(2)}, \Delta_1^{(2)})$ to $\mathcal{F}_{\mathsf{mRand}}^{3,r^{(2)},r^{(3)}}$ and receive $([\![w^j]\!]^{(2^n)}_{0,\mathsf{MAC}}, [\![v^j]\!]^{(2^n)}_{1,\mathsf{MAC}})$, which defines $\langle\!\langle k\cdot y^j\rangle\!\rangle^{(2^n)}$, for each $j \in [\ell]$. Note that this in particular defines a commitment $[\![y^j]\!]^{(2^n)}_{0,\mathsf{Key}} := \big((y^j, w^j), (k, -v^j)\big)$.
- Compute $[\![b'^j]\!]^{(2^n)}_{0,\mathsf{Key}} := \mathsf{elt}_\alpha([\![\boldsymbol{b'}^j]\!]^{(2)}_{0,\mathsf{Key}})$ and $[\![c'^j]\!]^{(2^n)}_{0,\mathsf{Key}} := [\![y^j]\!]^{(2^n)}_{0,\mathsf{Key}} + [\![b'^j]\!]^{(2^n)}_{0,\mathsf{Key}}$, for each $j \in [\ell]$, and execute $(c'^1, \ldots, c'^\ell) \leftarrow \mathsf{BatchOpen}\left([\![c'^1]\!]^{(2^n)}_{0,\mathsf{Key}}, \ldots, [\![c'^\ell]\!]^{(2^n)}_{0,\mathsf{Key}}\right)$.
- Compute $[\![\boldsymbol{y}^j]\!]^{(3)}_{0,\mathsf{Key}} := \mathsf{vec}_\alpha(c'^j) + [\![\boldsymbol{b'}^j]\!]^{(3)}_{0,\mathsf{Key}} + \mathsf{vec}_\alpha(c'^j)\odot [\![\boldsymbol{b'}^j]\!]^{(3)}_{0,\mathsf{Key}}$ for each $j \in [\ell]$.
- Put $[\![\boldsymbol{x'}^j]\!]^{(3)}_{0,\mathsf{Key}} := [\![\boldsymbol{y}^j_{[1,n-1]}]\!]^{(3)}_{0,\mathsf{Key}}$, compute $[\![\boldsymbol{z}^j]\!]^{(3)}_{0,\mathsf{Key}} := \boldsymbol{H}\cdot\boldsymbol{G}_{\mathrm{gadget}}\cdot[\![\boldsymbol{x'}^j]\!]^{(3)}_{0,\mathsf{Key}}$, and put $[\![z^{\ell+j}]\!]^{(3)}_{0,\mathsf{Key}} := [\![\boldsymbol{y}^j_n]\!]^{(3)}_{0,\mathsf{Key}} - 1$ for each $j \in [\ell]$.
- Check $1 \leftarrow \mathsf{BatchVer}([\![\boldsymbol{z}^1]\!]^{(3)}_{0,\mathsf{Key}}, \ldots, [\![\boldsymbol{z}^\ell]\!]^{(3)}_{0,\mathsf{Key}}, [\![z^{\ell+1}]\!]^{(3)}_{0,\mathsf{Key}}, \ldots, [\![z^{2\ell}]\!]^{(3)}_{0,\mathsf{Key}})$, and abort otherwise.

Modulus Conversion and Final Matrix Multiplication:

- Compute $\langle\!\langle b^j\rangle\!\rangle^{(2^n)} := \mathsf{elt}_\alpha(\langle\!\langle \boldsymbol{b}^j\rangle\!\rangle^{(2)})$, $\langle\!\langle c^j\rangle\!\rangle^{(2^n)} \leftarrow \langle\!\langle k\cdot y^j\rangle\!\rangle^{(2^n)} + \langle\!\langle b^j\rangle\!\rangle^{(2^n)}$, for each $j \in [\ell]$, and execute $(c^1, \ldots, c^\ell) \leftarrow \mathsf{BatchOpen}\left((\langle\!\langle c^1\rangle\!\rangle^{(2^n)}, \ldots, \langle\!\langle c^\ell\rangle\!\rangle^{(2^n)}), \{0,1\}\right)$. Then convert to sharings of $\mathsf{vec}_\alpha(k\cdot y^j)$ modulo 3 as $\langle\!\langle \mathsf{vec}_\alpha(k\cdot y^j)\rangle\!\rangle^{(3)} = \mathsf{vec}_\alpha(c^j) + \langle\!\langle \boldsymbol{b}^j\rangle\!\rangle^{(3)} + \mathsf{vec}_\alpha(c^j)\odot\langle\!\langle \boldsymbol{b}^j\rangle\!\rangle^{(3)}$.
- Compute $\langle\!\langle F_k(\boldsymbol{x}^j)\rangle\!\rangle^{(3)} = \boldsymbol{B}\cdot_3 \langle\!\langle \mathsf{vec}_\alpha(k\cdot y^j)\rangle\!\rangle^{(3)}$ and open to $\mathcal{P}_0$ as $(F_k(\boldsymbol{x}^1), \ldots, F_k(\boldsymbol{x}^\ell)) \leftarrow \mathsf{BatchOpen}\left((\langle\!\langle F_k(\boldsymbol{x}^1)\rangle\!\rangle^{(3)}, \ldots, \langle\!\langle F_k(\boldsymbol{x}^\ell)\rangle\!\rangle^{(3)}), 0\right)$.

The fully detailed proof can be found in Appendix C.4.

□

The OPRF protocol Π^M_{OPRF} amortized over large batch size ℓ requires communicating 3 $\mathbb{F}_{2^n}$ and t $\mathbb{F}_3$ elements along with the consumption of n daBits, n laBits and 1 entry in an authenticated VOLE per PRF evaluation.

6.2 Half-Malicious Protocol

We observe that the majority of the communication of our maliciously-secure OPRF protocol in Section 6.1 comes from the authenticated OLE protocol in Section 5.2, which incurs a factor B^2 overhead for each required triple. In the setting where only the receiver can be maliciously corrupted, and the sender is semi-honest, we note that the authenticated OLE triples are no longer needed by the following reasoning. Since the sender is semi-honest, the sender's modulo 2 and modulo 3 shares of the daBit no longer need to be authenticated. Additionally, since the modulo 3 sharing of the final output is only opened to the receiver, the receiver's modulo 3 share of the daBit does not need to be authenticated.

We formalize this correlation as half-half-authenticated bits (haBits) and provide a protocol realizing it using a single $\mathbb{F}_2$-VOLE + $\log_2 3$ bits of communication per haBit in Figure 11. Moreover, only the receiver's share of the key-multiplication needs to be authenticated, which we formalize as half-authenticated VOLE , and present a protocol in Figure 12, saving more than half of the consistency checks needed for the authenticated VOLE protocol. Ultimately, this leads to a half-malicious OPRF protocol, which we present in Figure 13. First we present the ideal functionality for the building blocks described above in Figure 10. We define the following shorthand notation to use in our protocol to commit to a chosen input $\boldsymbol{x} \in \mathbb{F}_{q^r}^{\ell}$:

- $\langle\!\langle \gamma \cdot \boldsymbol{x} \rangle\!\rangle_0^{(q^r)} \leftarrow \mathsf{haCommit}(\boldsymbol{x}, q^r, \gamma, \Delta_1)$: The parties send $(\mathsf{sid}_{\mathsf{VOLE}}, \mathsf{sid}_{\mathsf{MAC}}, \mathtt{haVOLE}, q^r, \ell)$ to the functionality $\mathcal{F}^{p,r^{(2)},r^{(p)}}_{\mathsf{mRand}}$ (⑤) to obtain $\langle\!\langle \gamma \cdot \boldsymbol{y} \rangle\!\rangle_0^{(q^r)} \leftarrow ([\![\boldsymbol{w}]\!]^{(q^r)}_{0,\mathsf{MAC}}, \boldsymbol{v})$. $\mathcal{P}_0$ sends $\boldsymbol{\delta} := \boldsymbol{x} - \boldsymbol{y}$ to $\mathcal{P}_1$. Return $\langle\!\langle \gamma \cdot \boldsymbol{x} \rangle\!\rangle_0^{(q^r)} \leftarrow ([\![\boldsymbol{w}]\!]^{(q^r)}_{0,\mathsf{MAC}}, \boldsymbol{v} + \gamma \cdot \boldsymbol{\delta})$.

The amortized communication is equal to our fully malicious OPRF sending 3 $\mathbb{F}_{2^n}$ and 1 $\mathbb{F}_{3^{r(3)}}$ elements. However it consumes n laBits, n haBits and 1 entry in a half-authenticated VOLE per PRF evaluation, which performs significantly better than our fully-malicious protocol, see Table 2.

Half-Half-Authenticated Bits.

Theorem 5. *The protocol $\Pi^{p,r^{(2)}}_{\mathsf{haBits}}$ in Figure 11 securely realizes the functionality $\mathcal{F}^{p,r^{(2)},r^{(p)}}_{\mathsf{mRand}}$ (⑤) against malicious prover $\mathcal{P}_0$ and semi-honest verifier $\mathcal{P}_1$ in the $(\mathcal{F}^{2,r^{(2)}}_{\mathsf{sVOLE}})$-hybrid model, modeling H as a random oracle.*

Proof (sketch). First of all, the parties locally sample random bits $\boldsymbol{b}^0$, $\boldsymbol{b}^1$, which immediately gives them secret shares of random bits $\boldsymbol{b}^0 \oplus \boldsymbol{b}^1$ modulo 2. Subsequently, the prover commits to their modulo 2 shares $\boldsymbol{b}^0$ to authenticate it and the parties hash parts of this VOLE commitment using a random oracle to obtain random-message OTs modulo p with choice bits $\boldsymbol{b}^0$. They can then derandomize this OT with respect to the sender's message to obtain secret shares of $\boldsymbol{b}^0 \odot \boldsymbol{b}^1$ modulo p, which in turn gives them secret shares of the XOR modulo p. The derandomization value δ_j leaks no information about b^1_j since $\mathcal{P}_0$ only knows one of $m_{0,j}$, $m_{1,j}$ and the other value looks uniformly random. See appendix C.5 for the full proof. □

⑤ **Half-Half-Authenticated Bits:** Upon receiving $(\mathsf{sid}, \mathtt{haBits}, \ell)$ from $\mathcal{P}_0$ and $\mathcal{P}_1$:
- Sample $\boldsymbol{b} \xleftarrow{\$} \{0,1\}^\ell$.
- Sample a fresh half-authenticated secret-sharing $\langle\langle \boldsymbol{b} \rangle\rangle_0^{(2)} \leftarrow (\llbracket \boldsymbol{b}^0_{(2)} \rrbracket_0^{(2)}, b^1_{(2)})$
- Sample a fresh unauthenticated secret-sharing $\langle \boldsymbol{b} \rangle^{(p)}$.
- Output $(\langle\langle \boldsymbol{b} \rangle\rangle_0^{(2)}, \langle \boldsymbol{b} \rangle^{(p)})$.

⑥ **Half-Authenticated VOLE:** Upon receiving $(\mathsf{sid}_{\mathsf{VOLE}}, \mathsf{sid}_{\mathsf{MAC}}, \mathtt{haVOLE}, q^r, \ell)$ from $\mathcal{P}_0$ and $\mathcal{P}_1$:
- Sample $\boldsymbol{y} \xleftarrow{\$} \mathbb{F}_{q^r}^\ell$.
- Sample an authentication $\llbracket \boldsymbol{y} \rrbracket_{0,\mathsf{VOLE}}^{(q^r)}$. That is, $\mathcal{P}_0$ obtains $\boldsymbol{w} \in \mathbb{F}_{q^r}^\ell$ and $\mathcal{P}_1$ obtains $\boldsymbol{v} \in \mathbb{F}_{q^r}^\ell$, satisfying $\boldsymbol{v} + \boldsymbol{w} = \gamma \cdot \boldsymbol{y}$.
- Similarly sample $\llbracket \boldsymbol{w} \rrbracket_{0,\mathsf{MAC}}^{(q^r)}$.
- Return $\langle\langle \gamma \cdot \boldsymbol{y} \rangle\rangle_0^{(q^r)} \leftarrow (\llbracket \boldsymbol{w} \rrbracket_{0,\mathsf{MAC}}^{(q^r)}, \boldsymbol{v})$.

Fig. 10: Ideal preprocessing functionality $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$ (Continued for Half-Malicious Setting.)

Fig. 11 Half-Half-Authenticated Bits Protocol $\Pi_{\mathsf{haBits}}^{p,r^{(2)}}$

Parameters: Two parties $\mathcal{P}_0$ and $\mathcal{P}_1$ also referred to as the prover and verifier, respectively, number of haBits ℓ, finite field $\mathbb{F}_p$, extension field parameter $r^{(2)}$ for $\mathbb{F}_2$. $\mathsf{H} : \{0,1\}^* \to \mathbb{F}_p$ is a random oracle.

Initialize: The parties send $(\mathsf{sid}, \mathtt{init}, 1)$ to $\mathcal{F}_{\mathsf{sVOLE}}^{2,r^{(2)}}$. $\mathcal{P}_1$ receives $\Delta^{(2)} \in \mathbb{F}_{2^{r^{(2)}}}$.

Protocol:
- For $i \in \{0,1\}$, $\mathcal{P}_i$ samples $\boldsymbol{b}^i \xleftarrow{\$} \mathbb{F}_2^\ell$.
- The parties send $\mathsf{Commit}(\boldsymbol{b}^0, 2, r^{(2)}, \Delta^{(2)})$ to $\mathcal{F}_{\mathsf{sVOLE}}^{2,r^{(2)}}$ and obtain $\llbracket \boldsymbol{b}^0 \rrbracket_0^{(2)} = \Big((\boldsymbol{b}^0, \boldsymbol{M}), (\Delta^{(2)}, \boldsymbol{K})\Big)$ satisfying $\boldsymbol{M} = \Delta^{(2)} \cdot \boldsymbol{b}^0 + \boldsymbol{K}$ in $\mathbb{F}_{2^{r^{(2)}}}$.
- $\mathcal{P}_0$ computes $m_{b_j^0,j} := \mathsf{H}(j, M_j)$.
- $\mathcal{P}_1$ computes $m_{0,j} := \mathsf{H}(j, K_j)$ and $m_{1,j} := \mathsf{H}(j, \Delta^{(2)} + K_j)$, for each $j \in [\ell]$.
- $\mathcal{P}_1$ sets $\delta_j := m_{0,j} - m_{1,j} - 2 \cdot b_j^1 \bmod p$, for each $j \in [\ell]$, and sends $\boldsymbol{\delta}$ to $\mathcal{P}_0$.
- $\mathcal{P}_0$ puts $\tilde{\boldsymbol{b}}^0 := \boldsymbol{m}_{\boldsymbol{b}^0} + \boldsymbol{b}^0 \odot \boldsymbol{\delta} + \boldsymbol{b}^0 \bmod p$ and $\mathcal{P}_1$ puts $\tilde{\boldsymbol{b}}^1 := \boldsymbol{b}^1 - \boldsymbol{m_0} \bmod p$.
- The parties output the sharings $\langle\langle \boldsymbol{b} \rangle\rangle_0^{(2)} \leftarrow (\llbracket \boldsymbol{b}^0 \rrbracket_0^{(2)}, \boldsymbol{b}^1)$ and $\langle \boldsymbol{b} \rangle^{(3)} := (\tilde{\boldsymbol{b}}^0, \tilde{\boldsymbol{b}}^1)$

Half-Authenticated VOLE.

Theorem 6. *The protocol $\Pi_{\mathsf{haVOLE}}^{q,r}$ in Figure 12 realizes the ideal functionality $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$ (⑥) against a malicious receiver and semi-honest sender in the $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$ (⓪)-, $\mathcal{F}_{\mathsf{EQ}}$-hybrid model, assuming $q^r \geq 2^\kappa$.*

Proof. The proof follows from a straightforward adaptation of the proof of Theorem 3. □

Remark 4. Note that the consistency check in Figure 12 is actually not needed for our half-malicious OPRF protocol in Figure 13. Taking away the consistency check, the receiver's MAC on $\boldsymbol{w}$ is not guaranteed to be correct, but the receiver can not adapt their MAC in a way that it will be valid for a different choice of share $\boldsymbol{w}'$ without successfully guessing $\Delta_1 \cdot \gamma$. The faulty MAC will be recognized by the sender during the opening of the c^j, so this optimization does not affect security and reduces communication cost.

Fig. 12 Half-Authenticated VOLE protocol $\Pi^{q,r}_{\mathsf{haVOLE}}$

Parameters: Two parties $\mathcal{P}_0$ and $\mathcal{P}_1$, also referred to as the receiver, having input $\boldsymbol{y} \in \mathbb{F}_{q^r}^{\ell}$ ($\boldsymbol{u}$ can also be a vector over the base field $\mathbb{F}_q$ but to fit with our OPRF protocol we use VOLE over the extension field) and the sender, having input $\gamma \in \mathbb{F}_{q^r}$, respectively, length of authenticated VOLE correlation ℓ, base field characteristic $q \in \{2, p\}$ and extension field parameter $r = r^{(q)}$. $\mathsf{H} : \{0,1\}^* \to \{0,1\}^{\lambda}$ is a random oracle.

Initialization:

- Send $(\mathsf{sid}_{\mathsf{MAC}}, \mathtt{init}, 1, q)$, $(\mathsf{sid}_{\mathsf{VOLE}}, \mathtt{init}, 1, q)$ and $(\mathsf{sid}_{\mathsf{Mask}}, \mathtt{init}, 1, q)$ to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$. $\mathcal{P}_1$ receives Δ_1, γ and $\Delta_{\mathsf{Mask}} \in \mathbb{F}_{q^r}$.

Generating Candidate Correlation:

- Query $[\![(\boldsymbol{y}\|s)]\!]^{(q^r)}_{0,\mathsf{VOLE}} \leftarrow \mathsf{RandCommit}((\boldsymbol{y}\|s), q^r, 1, \gamma)$ to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$ with $\mathcal{P}_0$ as receiver. That is, $\mathcal{P}_0$ obtains $(\boldsymbol{w}\|e) \in \mathbb{F}_{q^r}^{\ell+1}$ and $\mathcal{P}_1$ obtains $(\boldsymbol{v}\|f) \in \mathbb{F}_{q^r}^{\ell+1}$, satisfying $(\boldsymbol{v}\|f) + (\boldsymbol{w}\|e) = \gamma \cdot (\boldsymbol{y}\|s)$.
- Query $\mathsf{Commit}((\boldsymbol{y}\|s), q^r, 1, \Delta_{\mathsf{Mask}} \to \Delta_1 \cdot \gamma)$ to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$ with $\mathcal{P}_0$ as receiver. That is, $\mathcal{P}_0$ obtains $(\bar{\boldsymbol{w}}\|\bar{e}) \in \mathbb{F}_{q^r}^{\ell+1}$ and $\mathcal{P}_1$ obtains $(\bar{\boldsymbol{v}}\|\bar{f}) \in \mathbb{F}_{q^r}^{\ell+1}$, satisfying $(\bar{\boldsymbol{w}}\|\bar{e}) + (\bar{\boldsymbol{v}}\|\bar{f}) = (\Delta_1 \cdot \gamma) \cdot (\boldsymbol{y}\|s)$.

Consistency Check:

- Query $[\![(\boldsymbol{w}\|e)]\!]^{(q^r)}_{0,\mathsf{MAC}} \leftarrow \mathsf{Commit}((\boldsymbol{w}\|e), q^r, 1, \Delta_1)$ to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$ with $\mathcal{P}_0$ as receiver. That is, $\mathcal{P}_0$ obtains $(\boldsymbol{t}\|\tau) \in \mathbb{F}_{q^r}^{\ell+1}$ and $\mathcal{P}_1$ obtains $(\boldsymbol{u}\|\sigma) \in \mathbb{F}_{q^r}^{\ell+1}$ satisfying $(\boldsymbol{t}\|\tau) + (\boldsymbol{u}\|\sigma) = \Delta_1 \cdot (\boldsymbol{w}\|e)$.
- $\mathcal{P}_0$ computes $\tilde{t} := \mathsf{H}\left((\bar{\boldsymbol{w}}\|\bar{e}) - (\boldsymbol{t}\|\tau)\right)$ and sends $(\mathsf{sid}, \mathtt{check}, \tilde{t}, 0)$, $\mathcal{P}_1$ computes $\tilde{u} := \mathsf{H}\left((\boldsymbol{u}\|\sigma) - (\bar{\boldsymbol{v}}\|\bar{f}) + \Delta_1 \cdot (\boldsymbol{v}\|f)\right)$ and sends $(\mathsf{sid}, \mathtt{check}, \tilde{u}, 0)$ to $\mathcal{F}_{\mathsf{EQ}}$. If any of the parties receive $\mathtt{abort}$ from $\mathcal{F}_{\mathsf{EQ}}$, they abort. Otherwise, they continue.
- Output $[\![\boldsymbol{w}]\!]^{(q^r)}_{0,\mathsf{MAC}} := ((\boldsymbol{w}, \bar{\boldsymbol{w}}), (\Delta_1, \Delta_1 \cdot \boldsymbol{v} - \bar{\boldsymbol{v}}))$ to the parties and $\boldsymbol{v}$ to $\mathcal{P}_1$.

Half-Malicious OPRF Protocol. We now present the Half-Malicious OPRF protocol in Figure 13. We highlight the parts where the half-malicious protocol significantly differs from the fully-malicious protocol in Section 6.1 by *[Half Malicious]*.

Theorem 7. *The protocol $\Pi^{\mathsf{HM}}_{\mathsf{OPRF}}$ in Figure 13 realizes the ideal $\mathcal{F}_{\mathsf{OPRF}}$ functionality for the strong PRF family F_k from (5) against a malicious receiver and semi-honest sender in the $\mathcal{F}_{\mathsf{mRand}}^{3,r^{(2)},r^{(p)}}$ (⓪, ①, ⑤, ⑥)-hybrid model, assuming $n = d + 2(n' - d) + 1$, $r^{(2)} = n \geq \kappa$ and $r^{(3)} \geq \kappa / \log_2 3$.*

Proof. Correctness when both parties are honest follows very similarly to the proof of Theorem 4, with the main difference being that only the receiver's shares are authenticated throughout the computation under the sender's MAC keys $\Delta_1^{(2)} \in \mathbb{F}_{2^{r^{(2)}}}$ and $\Delta_1^{(3)} \in \mathbb{F}_{3^{r^{(3)}}}$. This is realized by using half-half-authenticated bits (haBits) instead of doubly-authenticated bits (daBits) to perform the modulus conversion at the end of the protocol and by using the half-authenticated VOLE functionality $\mathcal{F}_{\mathsf{mRand}}^{3,r^{(2)},r^{(3)}}$ (⑤) to get one-sided instead of two-sided authenticated shares of the key-input multiplication $k \cdot y^j$.

The simulators for a maliciously corrupted receiver $\mathcal{P}_0$ and a semi-honestly corrupted sender $\mathcal{P}_1$ can be seen as a special case of the proof of Theorem 4, except that there are no openings to $\mathcal{P}_0$ that need to be simulated. □

Note that in the converse setting, where the receiver is semi-honest but the sender can be maliciously corrupted, we still need both the sender's modulo 2 as well as the modulo 3 share of the daBit to be authenticated, which does not seem to lead to a significantly cheaper protocol.

Fig. 13 Half-Maliciously-Secure OPRF protocol $\Pi^{\mathsf{HM}}_{\mathsf{OPRF}}$

Parameters: Two parties $\mathcal{P}_0$ and $\mathcal{P}_1$, also referred to as the client and the server, respectively. The client holds a batch of inputs $(\boldsymbol{x}^j)_{j\in[m]} \in (\mathbb{F}_2^d)^m$. Public matrix $\boldsymbol{B} \in \mathbb{F}_3^{t\times n}$, linear code $\boldsymbol{G} \in \mathbb{F}_3^{n'\times d}$ with corresponding parity-check matrix $\boldsymbol{H} \in \mathbb{F}_3^{(n'-d)\times n'}$, gadget matrix $\boldsymbol{G}_{\mathrm{gadget}} = (2,1)\otimes \boldsymbol{I}_{n-1}$, integers $n, t, d, n', r^{(2)}, r^{(3)} \in \mathbb{N}$ depending on the security parameter $\lambda \in \mathbb{N}$ with $r^{(2)} = n$.

Preprocessing:

- Send $(\mathsf{sid}_{\mathsf{MAC}}, \mathtt{init}, 1, q)$ and $(\mathsf{sid}_{\mathsf{Key}}, \mathtt{init}, 1, q)$ to $\mathcal{F}_{\mathsf{mRand}}^{3,r^{(2)},r^{(3)}}$ for each $q \in \{2,3\}$. $\mathcal{P}_1$ receives $\Delta_1^{(2)} \in \mathbb{F}_{2^{r^{(2)}}}$, $\Delta_1^{(3)} \in \mathbb{F}_{3^{r^{(3)}}}$ and $k \in \mathbb{F}_{2^n}$, $\Delta_k^{(3)} \in \mathbb{F}_{3^{r^{(3)}}}$, respectively.
- Send $(\mathsf{sid}_{\mathsf{Key}}, \mathtt{laBits}, n\cdot\ell, 0)$ to $\mathcal{F}_{\mathsf{mRand}}^{3,n,r^{(3)}}$ and receive $[\![\boldsymbol{b'}^j]\!]^{(2)}_{0,\mathsf{Key}}$, $[\![\boldsymbol{b'}^j]\!]^{(3)}_{0,\mathsf{Key}}$ for $\boldsymbol{b'}^j \in \{0,1\}^n$ and $j \in [\ell]$.
- *[Half Malicious]*: Send $(\mathsf{sid}_{\mathsf{MAC}}, \mathtt{haBits}, n\cdot\ell)$ to $\mathcal{F}_{\mathsf{mRand}}^{3,r^{(2)},r^{(3)}}$ and receive $\langle\!\langle \boldsymbol{b}^j \rangle\!\rangle_0^{(2)}$, $\langle \boldsymbol{b}^j \rangle^{(3)}$ for $\boldsymbol{b}^j \in \{0,1\}^n$ and $j \in [\ell]$.

Key-Multiplication and Input-Encoding:

- $\mathcal{P}_0$ computes $\boldsymbol{x'}^j := \mathsf{decomp}(\boldsymbol{G} \cdot_3 \boldsymbol{x}^j) \in \mathbb{F}_2^{d+2(n'-d)}$ and $y^j := \mathsf{elt}_\alpha(\boldsymbol{x'}^j \| 1) \in \mathbb{F}_{2^n}$ for each $j \in [\ell]$.
- *[Half Malicious]:* Query $\mathsf{haCommit}(y^j, 2^n, 1, k, \Delta_1^{(2)})$ to $\mathcal{F}_{\mathsf{mRand}}^{3,r^{(2)},r^{(3)}}$ and receive $([\![w^j]\!]^{(2^n)}_{0,\mathsf{MAC}}, v^j)$, which defines $\langle\!\langle k\cdot y^j \rangle\!\rangle_0^{(2^n)}$, for each $j \in [\ell]$. Note that this in particular defines a commitment $[\![y^j]\!]^{(2^n)}_{0,\mathsf{Key}} := \big((y^j, w^j), (k, -v^j)\big)$.
- Compute $[\![b'^j]\!]^{(2^n}_{0,\mathsf{Key}} := \mathsf{elt}_\alpha([\![\boldsymbol{b'}^j]\!]^{(2)}_{0,\mathsf{Key}})$ and $[\![c'^j]\!]^{(2^n)}_{0,\mathsf{Key}} := [\![y^j]\!]^{(2^n)}_{0,\mathsf{Key}} + [\![b'^j]\!]^{(2^n)}_{0,\mathsf{Key}}$, for each $j \in [\ell]$, and execute $(c'^1, \ldots, c'^\ell) \leftarrow \mathsf{BatchOpen}\left([\![c'^1]\!]^{(2^n)}_{0,\mathsf{Key}}, \ldots, [\![c'^\ell]\!]^{(2^n)}_{0,\mathsf{Key}}\right)$.
- Compute $[\![\boldsymbol{y}^j]\!]^{(3)}_{0,\mathsf{Key}} := \mathsf{vec}_\alpha(c'^j) + [\![\boldsymbol{b'}^j]\!]^{(3)}_{0,\mathsf{Key}} + \mathsf{vec}_\alpha(c'^j) \odot [\![\boldsymbol{b'}^j]\!]^{(3)}_{0,\mathsf{Key}}$ for each $j \in [\ell]$.
- Put $[\![\boldsymbol{x'}^j]\!]^{(3)}_{0,\mathsf{Key}} := [\![\boldsymbol{y}^j_{[1,n-1]}]\!]^{(3)}_{0,\mathsf{Key}}$, compute $[\![\boldsymbol{z}^j]\!]^{(3)}_{0,\mathsf{Key}} := \boldsymbol{H} \cdot \boldsymbol{G}_{\mathrm{gadget}} \cdot [\![\boldsymbol{x'}^j]\!]^{(3)}_{0,\mathsf{Key}}$, and put $[\![z^{\ell+j}]\!]^{(3)}_{0,\mathsf{Key}} := [\![y^j_n]\!]^{(3)}_{0,\mathsf{Key}} - 1$ for each $j \in [\ell]$.
- Check $1 \leftarrow \mathsf{BatchVer}([\![\boldsymbol{z}^1]\!]^{(3)}_{0,\mathsf{Key}}, \ldots, [\![\boldsymbol{z}^\ell]\!]^{(3)}_{0,\mathsf{Key}}, [\![z^{\ell+1}]\!]^{(3)}_{0,\mathsf{Key}}, \ldots, [\![z^{2\ell}]\!]^{(3)}_{0,\mathsf{Key}})$, and abort otherwise.

Modulus Conversion and Final Matrix Multiplication:

- *[Half Malicious]*: Convert to secret sharings over $\mathbb{F}_{2^n}$ as $\langle\!\langle k\cdot y^j \rangle\!\rangle_0^{(2^n)} := \mathsf{elt}_\alpha(\langle\!\langle k\cdot \boldsymbol{y}^j \rangle\!\rangle_0^{(2)})$ and $\langle\!\langle b^j \rangle\!\rangle_0^{(2^n)} := \mathsf{elt}_\alpha(\langle\!\langle \boldsymbol{b}^j \rangle\!\rangle_0^{(2)})$, where $y^j = \mathsf{elt}_\alpha(\boldsymbol{y}^j)$ and $b^j = \mathsf{elt}_\alpha(\boldsymbol{b}^j)$, compute $\langle\!\langle c^j \rangle\!\rangle_0^{(2^n)} \leftarrow \langle\!\langle k\cdot y^j \rangle\!\rangle_0^{(2^n)} + \langle\!\langle b^j \rangle\!\rangle_0^{(2^n)}$ and execute $(c^1, \ldots, c^\ell) \leftarrow \mathsf{BatchOpen}\left((\langle\!\langle c^1 \rangle\!\rangle_0^{(2^n)}, \ldots, \langle\!\langle c^\ell \rangle\!\rangle_0^{(2^n)}), 1\right)$, and $\mathcal{P}_1$ simply sends their shares $(c_1^1, \ldots, c_1^\ell)$ for the opening to $\mathcal{P}_0$. Then convert to sharings of $\mathsf{vec}_\alpha(k\cdot y^j)$ modulo 3 by computing $\langle \mathsf{vec}_\alpha(k\cdot y^j)\rangle^{(3)} = \mathsf{vec}_\alpha(c^j) + \langle \boldsymbol{b}^j \rangle^{(3)} + \mathsf{vec}_\alpha(c^j) \odot \langle \boldsymbol{b}^j \rangle^{(3)}$.
- *[Half Malicious]*: Compute $\langle \boldsymbol{B} \cdot \mathsf{vec}_\alpha(k\cdot y^j)\rangle^{(3)} = \boldsymbol{B} \cdot_3 \langle \mathsf{vec}_\alpha(k\cdot y^j)\rangle^{(3)}$ and reconstruct this value to the receiver as $F_k(x^j) \in \mathbb{F}_3^t$.

6.3 Semi-Honest Protocol

In the semi-honest setting, the receiver does not need to prove that they encoded their input correctly, and might as well use a hash function $\mathsf{H} : \{0,1\}^* \to \{0,1\}^{n-1}$, modelled as a random oracle, to encode their inputs as $y^j := \mathsf{elt}_\alpha(\mathsf{H}(\boldsymbol{x}^j)\|1)$. Furthermore, the parties can simply use the standard VOLE functionality $\mathcal{F}_{\mathsf{sVOLE}}^{2^n,1}$ to compute shares of the key-multiplication $w^j + v^j = k \cdot y^j$. The modulus conversion can then be done using efficient OT-based techniques from previous work [2] or [39]. We chose not to implement this protocol since we expect the resulting OPRF to have very similar performance to [2].

6.4 Shared-Input Shared-Output Protocol

For some applications, it is useful for the parties to only obtain secret shares of the receiver's OPRF outputs, in order to enable some follow-up computation using the secret-shared PRF outputs. For example, this can be used to achieve stronger security for Circuit-PSI protocols as in [54, 58]. Our protocol from Figure 9 naturally enables this by omitting the final $\mathsf{BatchOpen}$ call. Formally, the ideal OPRF functionality $\mathcal{F}_{\mathsf{OPRF}}$ in Figure 1 would need to be adapted to output authenticated secret shares $\langle\!\langle F_k(\boldsymbol{x}^j)\rangle\!\rangle^{(3)}$ with respect to MAC keys $\Delta_0^{(3)}, \Delta_1^{(3)} \in \mathbb{F}_{3^{r^{(3)}}}$ consistent with the ideal functionality $\mathcal{F}_{\mathsf{mRand}}^{3,r^{(2)},r^{(3)}}$. As an artifact of this, the ideal functionality $\mathcal{F}_{\mathsf{OPRF}}$ must also inherit the "**Global Key-Query**", "**Special Abort**" and "**Special Guess**" queries from $\mathcal{F}_{\mathsf{mRand}}^{3,r^{(2)},r^{(3)}}$ to make the security proof go through. Other than some small tweaks the proof will be analogous to Theorem 4.

Certain applications [43, 46, 49] additionally require the receiver's inputs $\boldsymbol{x}^j$ and the sender's key k to be secret-shared between both parties. Enabling this would require larger changes to the protocol structure, which we will sketch below. First, the input-encoding step takes as input authenticated secret sharings $\langle\!\langle \boldsymbol{x}^j\rangle\!\rangle^{(2)}$ and can be executed in a distributed way by: first the parties can convert these to authenticated sharings $\langle\!\langle \boldsymbol{x}^j\rangle\!\rangle^{(3)}$ modulo 3 using daBits and can locally compute $\langle\!\langle \boldsymbol{x'}^j\rangle\!\rangle^{(3)} := \langle\!\langle \boldsymbol{G} \cdot_3 \boldsymbol{x}^j\rangle\!\rangle^{(3)}$; then compute shares of the binary decomposition $\langle\!\langle \mathsf{decomp}(\boldsymbol{G} \cdot_3 \boldsymbol{x}^j)\rangle\!\rangle^{(3)}$ (using authenticated $\mathbb{F}_3$-multiplication triples, which can be obtained from two authenticated OLE triples as in Figure 15) as

$$\langle\!\langle (\boldsymbol{x'}^j_{[1,d]}, 2 \cdot \boldsymbol{x'}^j_{[d+1,n']} + 2 \cdot (\boldsymbol{x'}^j_{[d+1,n']})^2, \boldsymbol{x'}^j_{[d+1,n']} + 2 \cdot (\boldsymbol{x'}^j_{[d+1,n']})^2) \rangle\!\rangle^{(3)}.$$

Convert each entry to a modulo 2 sharing using $d+2(n'-d)$ daBits and $\mathbb{F}_3$-multiplication triples, and then locally to an $\mathbb{F}_{2^n}$ sharing to obtain $\langle\!\langle y^j\rangle\!\rangle^{(2^n)}$. We now need a VOLE functionality which takes authenticated secret sharings $\langle\!\langle k\rangle\!\rangle^{(2^n)}$, $\langle\!\langle y^j\rangle\!\rangle^{(2^n)}$ and outputs $\langle\!\langle k \cdot \boldsymbol{y}\rangle\!\rangle^{(2^n)}$, which can be obtained using the authenticated VOLE functionality and some additional consistency checks (as in Figure 16). The modulus conversion and final matrix multiplication steps remain identical to Figure 9, such that the parties finally obtain $\langle\!\langle F_k(\boldsymbol{x}^j)\rangle\!\rangle^{(3)}$ for $j \in [\ell]$. The resulting protocol is presented in Figure 17, but we first present the ideal functionalities of the building blocks introduced above in Figure 14.

The estimated overhead of having secret-shared inputs is approximately 4.4× the cost of the secret-shared output OPRF. The concrete cost is 21.6KB per evaluation. A similar computation overhead is expected, as the increase in cost comes from the increased usages of building blocks.

Authenticated Multiplication Triples.

Theorem 8. *The protocol $\Pi_{\mathsf{aTriple}}^{q,r}$ in Figure 15 realizes the ideal functionality $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$ (⑦) against malicious adversaries in the $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$ (⓪,③)-hybrid model, assuming $q^r \geq 2^\kappa$.*

⑦ **Authenticated Multiplication Triples:** Upon receiving $(\mathsf{sid}, \mathtt{triples}, q, \ell)$ from $\mathcal{P}_0$ and $\mathcal{P}_1$:
- Sample $\boldsymbol{a}, \boldsymbol{b} \xleftarrow{\$} \mathbb{F}_q^\ell$ and let $\boldsymbol{c} := \boldsymbol{a} \odot \boldsymbol{b}$.
- Sample secret-sharings $\langle\!\langle \boldsymbol{a} \rangle\!\rangle^{(q)}$, $\langle\!\langle \boldsymbol{b} \rangle\!\rangle^{(q)}$ and $\langle\!\langle \boldsymbol{c} \rangle\!\rangle^{(q)}$.
- Output $(\langle\!\langle \boldsymbol{a} \rangle\!\rangle^{(q)}, \langle\!\langle \boldsymbol{b} \rangle\!\rangle^{(q)}, \langle\!\langle \boldsymbol{c} \rangle\!\rangle^{(q)})$

⑧ **Authenticated Shared-Input VOLE:** The parties hold inputs $\boldsymbol{y}^0, \boldsymbol{y}^1 \in \mathbb{F}_{q^r}^\ell$ and keys $\gamma^0, \gamma^1 \in \mathbb{F}_{q^r}$ (corresponding to $\mathsf{sid}_{\mathsf{VOLE}}$), which are committed to as $[\![\boldsymbol{y}^0]\!]_{0,\mathsf{MAC}}^{(q^r)}, [\![\boldsymbol{y}^1]\!]_{1,\mathsf{MAC}}^{(q^r)}$ and $[\![\gamma^0]\!]_{0,\mathsf{MAC}}^{(q^r)}, [\![\gamma^1]\!]_{1,\mathsf{MAC}}^{(q^r)}$, respectively. Let $\langle\!\langle \boldsymbol{y} \rangle\!\rangle^{(q^r)} \leftarrow ([\![\boldsymbol{y}^0]\!]_{0,\mathsf{MAC}}^{(q^r)}, [\![\boldsymbol{y}^1]\!]_{1,\mathsf{MAC}}^{(q^r)})$ and $\langle\!\langle \gamma \rangle\!\rangle^{(q^r)} \leftarrow ([\![\gamma^0]\!]_{0,\mathsf{MAC}}^{(q^r)}, [\![\gamma^1]\!]_{1,\mathsf{MAC}}^{(q^r)})$. Upon receiving $(\mathsf{sid}_{\mathsf{VOLE}}, \mathsf{sid}_{\mathsf{MAC}}, \mathtt{asiVOLE}, \langle\!\langle \boldsymbol{y} \rangle\!\rangle^{(q^r)}, \langle\!\langle \gamma \rangle\!\rangle^{(q^r)})$ from $\mathcal{P}_0$ and $\mathcal{P}_1$:
- Execute $\mathsf{aCommit}(\boldsymbol{y}, q, r, \gamma, \Delta_0^{(q)}, \Delta_1^{(q)})$, return $\langle\!\langle \gamma \cdot \boldsymbol{y} \rangle\!\rangle^{(q^r)} \leftarrow ([\![\boldsymbol{a}]\!]_{0,\mathsf{MAC}}^{(q^r)}, [\![\boldsymbol{b}]\!]_{1,\mathsf{MAC}}^{(q^r)})$.

Special Abort:
- If $\mathcal{P}_i$ is corrupted, for $i \in \{0,1\}$, receive $(\mathsf{sid}_{\mathsf{VOLE}}, \mathsf{sid}_{\mathsf{MAC}}, \mathtt{abort}^*, i, q)$ from $\mathcal{A}$. Return $\Delta_{i\oplus 1}$ to $\mathcal{P}_i$ and $\mathtt{abort}$ to both parties, and abort.

Special Guess:
- If $\mathcal{P}_i$ is corrupted, for $i \in \{0,1\}$, receive $(\mathsf{sid}_{\mathsf{VOLE}}, \mathsf{sid}_{\mathsf{MAC}}, \mathtt{guess}^*, (a,b,c), i, q)$ from $\mathcal{A}$ with $a, b, c \in \mathbb{F}_{q^r}$. If $c = a \cdot \gamma^{i\oplus 1} + b \cdot (\Delta_{i\oplus 1} \cdot \gamma^{i\oplus 1})$, return $\mathtt{success}$ to $\mathcal{P}_i$. Otherwise, return $\mathtt{fail}$.

⑨ **Shared-Input Shared-Output PRF:** $F_k : \mathbb{F}_2^d \to \mathbb{F}_3^t$ is a public PRF taking keys $k \in \mathbb{F}_{2^n}$. The keys $k^0, k^1 \in \mathbb{F}_{2^n}$ (corresponding to $\mathsf{sid}_{\mathsf{Key}}$) are committed to as $[\![k^0]\!]_{0,\mathsf{MAC}}^{(2^n)}, [\![k^1]\!]_{1,\mathsf{MAC}}^{(2^n)}$, respectively. The parties hold inputs $\boldsymbol{x}^{j,0}, \boldsymbol{x}^{j,1} \in \mathbb{F}_2^d$, which are committed to as $[\![\boldsymbol{x}^{j,0}]\!]_{0,\mathsf{MAC}}^{(2)}, [\![\boldsymbol{x}^{j,1}]\!]_{1,\mathsf{MAC}}^{(2)}$, for $j \in [\ell]$. Let $\langle\!\langle \boldsymbol{x}^j \rangle\!\rangle^{(2)} \leftarrow ([\![\boldsymbol{x}^{j,0}]\!]_{0,\mathsf{MAC}}^{(2)}, [\![\boldsymbol{x}^{j,1}]\!]_{1,\mathsf{MAC}}^{(2)})$ and $\langle\!\langle k \rangle\!\rangle^{(2^n)} \leftarrow ([\![k^0]\!]_{0,\mathsf{MAC}}^{(q^r)}, [\![k^1]\!]_{1,\mathsf{MAC}}^{(q^r)})$. Upon receiving $(\mathsf{sid}_{\mathsf{Key}}, \mathsf{sid}_{\mathsf{MAC}}, \mathtt{sisoPRF}, (\langle\!\langle \boldsymbol{x}^j \rangle\!\rangle^{(2)})_{j\in[\ell]}, \langle\!\langle k \rangle\!\rangle^{(2^n)}, q, \ell)$ from $\mathcal{P}_0$ and $\mathcal{P}_1$:
- Output $(\langle\!\langle F_k(\boldsymbol{x}^j) \rangle\!\rangle^{(3)})_{j\in[\ell]}$ authenticated under $\Delta_0^{(3)}, \Delta_1^{(3)} \in \mathbb{F}_{3^{r^{(3)}}}$.

Allow the "**Special Abort**" and "**Special Guess**" commands from $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$ (⑧).

Fig. 14: Ideal preprocessing functionality $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$ (Continued for Shared-Input Setting.)

Fig. 15 Authenticated Multiplication Triple Protocol $\Pi_{\mathsf{aTriple}}^{q,r}$

Parameters: Two parties $\mathcal{P}_0$ and $\mathcal{P}_1$, number of authenticated multiplication triples ℓ, base field characteristic $q \in \{2, p\}$ and extension field parameter $r = r^{(q)}$ defining $\mathbb{F}_{q^r}$.

Protocol:
- Send $(\mathsf{sid}_{\mathsf{MAC}}, \mathtt{init}, 0, q)$ and $(\mathsf{sid}_{\mathsf{MAC}}, \mathtt{init}, 1, q)$ to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$, such that $\mathcal{P}_0$ receives $\Delta_0 \in \mathbb{F}_{q^r}$ and $\mathcal{P}_1$ receives $\Delta_1 \in \mathbb{F}_{q^r}$.
- Send $(\mathsf{sid}_{\mathsf{MAC}}, \mathsf{aOLE}, q, 2\ell)$ to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$ and obtain $([\![\boldsymbol{a}^0]\!]_0^{(q)}, [\![\boldsymbol{b}^1]\!]_1^{(q)}, \langle\!\langle \boldsymbol{c_{ab}} \rangle\!\rangle^{(q)})$ and $([\![\boldsymbol{b}^0]\!]_0^{(q)}, [\![\boldsymbol{a}^1]\!]_1^{(q)}, \langle\!\langle \boldsymbol{c_{ba}} \rangle\!\rangle^{(q)})$, where $\boldsymbol{a}^0, \boldsymbol{a}^1, \boldsymbol{b}^0, \boldsymbol{b}^1, \boldsymbol{c_{ab}}, \boldsymbol{c_{ba}} \in \mathbb{F}_q^\ell$ satisfy $\boldsymbol{a}^0 \odot \boldsymbol{b}^1 = \boldsymbol{c_{ab}}$ and $\boldsymbol{b}^0 \odot \boldsymbol{a}^1 = \boldsymbol{c_{ba}}$, and $\boldsymbol{c}_{\boldsymbol{ab}}^0, \boldsymbol{c}_{\boldsymbol{ab}}^1, \boldsymbol{c}_{\boldsymbol{ba}}^0, \boldsymbol{c}_{\boldsymbol{ba}}^1 \in \mathbb{F}_q^\ell$ satisfy $\boldsymbol{c}_{\boldsymbol{ab}}^0 + \boldsymbol{c}_{\boldsymbol{ab}}^1 = \boldsymbol{c_{ab}}$ and $\boldsymbol{c}_{\boldsymbol{ba}}^0 + \boldsymbol{c}_{\boldsymbol{ba}}^1 = \boldsymbol{c_{ba}}$.
- Query $[\![\boldsymbol{a}^i \odot \boldsymbol{b}^i]\!]_i^{(q)} \leftarrow \mathsf{Commit}(\boldsymbol{a}^i \odot \boldsymbol{b}^i, q, r, \Delta_{i\oplus 1})$ for each $i \in \{0,1\}$.
- Send $(\mathsf{sid}_{\mathsf{MAC}}, \mathtt{prove}, \mathsf{id}_1^i, \ldots, \mathsf{id}_{3\ell}^i, C_1, \ldots, C_\ell, q)$ to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$ for each $i \in \{0,1\}$, where $\mathsf{id}_1^i, \ldots, \mathsf{id}_\ell^i$ are associated with $[\![\boldsymbol{a}^i]\!]_i^{(q)}$, $\mathsf{id}_{\ell+1}^i, \ldots, \mathsf{id}_{2\ell}^i$ are associated with $[\![\boldsymbol{b}^i]\!]_i^{(q)}$, $\mathsf{id}_{2\ell+1}^i, \ldots, \mathsf{id}_{3\ell}^i$ are associated with $[\![\boldsymbol{a}^i \odot \boldsymbol{b}^i]\!]_i^{(q)}$, and the circuits $C_j : \mathbb{F}_q^{3\ell} \to \mathbb{F}_q$ are given by $C_j(X_1, \ldots, X_{3\ell}) := X_j \cdot X_{\ell+j} - X_{2\ell+j}$ for $j \in [\ell]$.
- Put $[\![\boldsymbol{c}^i]\!]_i^{(q)} := [\![\boldsymbol{a}^i \odot \boldsymbol{b}^i]\!]_i^{(q)} + [\![\boldsymbol{c}_{\boldsymbol{ab}}^i]\!]_i^{(q)} + [\![\boldsymbol{c}_{\boldsymbol{ba}}^i]\!]_i^{(q)}$ for $i \in \{0,1\}$, and $\langle\!\langle \boldsymbol{a} \rangle\!\rangle^{(q)} \leftarrow ([\![\boldsymbol{a}^0]\!]_0^{(q)}, [\![\boldsymbol{a}^1]\!]_1^{(q)})$, $\langle\!\langle \boldsymbol{b} \rangle\!\rangle^{(q)} \leftarrow ([\![\boldsymbol{b}^0]\!]_0^{(q)}, [\![\boldsymbol{b}^1]\!]_1^{(q)})$, $\langle\!\langle \boldsymbol{c} \rangle\!\rangle^{(q)} \leftarrow ([\![\boldsymbol{c}^0]\!]_0^{(q)}, [\![\boldsymbol{c}^1]\!]_1^{(q)})$.
- Output the triples $(\langle\!\langle \boldsymbol{a} \rangle\!\rangle^{(q)}, \langle\!\langle \boldsymbol{b} \rangle\!\rangle^{(q)}, \langle\!\langle \boldsymbol{c} \rangle\!\rangle^{(q)})$.

Proof (sketch). The idea is to combine two authenticated OLE triples $(\llbracket \boldsymbol{a}^0 \rrbracket_0^{(q)}, \llbracket \boldsymbol{b}^1 \rrbracket_1^{(q)}, \langle\langle \boldsymbol{c_{ab}} \rangle\rangle^{(q)})$ and $(\llbracket \boldsymbol{b}^0 \rrbracket_0^{(q)}, \llbracket \boldsymbol{a}^1 \rrbracket_1^{(q)}, \langle\langle \boldsymbol{c_{ba}} \rangle\rangle^{(q)})$ into an authenticated multiplication triple

$$(\boldsymbol{a}^0 + \boldsymbol{a}^1) \odot (\boldsymbol{b}^0 + \boldsymbol{b}^1) = \boldsymbol{a}^0 \odot \boldsymbol{b}^0 + \boldsymbol{a}^1 \odot \boldsymbol{b}^1 + \boldsymbol{a}^0 \odot \boldsymbol{b}^1 + \boldsymbol{b}^0 \odot \boldsymbol{a}^1.$$

That is, the parties have authenticated shares of the cross-terms $\boldsymbol{a}^0 \odot \boldsymbol{b}^1 = \boldsymbol{c_{ab}}$ and $\boldsymbol{b}^0 \odot \boldsymbol{a}^1 = \boldsymbol{c_{ba}}$, can commit to $\boldsymbol{a}^0 \odot \boldsymbol{b}^0$ and $\boldsymbol{a}^1 \odot \boldsymbol{b}^1$ and use a VOLE-ZK proof to show that these were computed correctly. Since no messages are sent directly between the parties, only through the underlying ideal functionalities, a simulator can be constructed in a straightforward way. □

Claim. The protocol $\Pi^{q,r}_{\mathsf{aTriple}}$ requires amortized 2 $\mathbb{F}_q$ elements of communication plus the cost of 2 calls to $\mathcal{F}^{p,r^{(2)},r^{(p)}}_{\mathsf{mRand}}$ (③). Instantiating $\mathcal{F}^{p,r^{(2)},r^{(p)}}_{\mathsf{mRand}}$ (③) with $\Pi^{3,r^{(3)}}_{\mathsf{aOLE}}$ gives a concrete efficiency of 246 bits of communication per multiplication triple over $\mathbb{F}_3$ with authentication over $\mathbb{F}_{3^{r^{(3)}}}$

Authenticated Shared-Input VOLE.

Theorem 9. *The protocol $\Pi^{q,r}_{\mathsf{asiVOLE}}$ in Figure 16 realizes the ideal functionality $\mathcal{F}^{p,r^{(2)},r^{(p)}}_{\mathsf{mRand}}$ (⑧) against malicious adversaries in the $\mathcal{F}^{p,r^{(2)},r^{(p)}}_{\mathsf{mRand}}$ (⓪, ④)-, $\mathcal{F}_{\mathsf{EQ}}$-hybrid model, assuming $q^r \geq 2^\kappa$.*

Proof (sketch). To compute authenticated secret shares of the scalar-vector product

$$(\gamma^0 + \gamma^1) \cdot (\boldsymbol{y}^0 + \boldsymbol{y}^1) = \gamma^0 \cdot \boldsymbol{y}^0 + \gamma^1 \cdot \boldsymbol{y}^1 + \gamma^0 \cdot \boldsymbol{y}^1 + \gamma^1 \cdot \boldsymbol{y}^1,$$

the idea is to let the parties commit to the local terms $\gamma^0 \cdot \boldsymbol{y}^0$, $\gamma^1 \cdot \boldsymbol{y}^1$ and use the authenticated VOLE functionality $\mathcal{F}^{p,r^{(2)},r^{(p)}}_{\mathsf{mRand}}$ (④) to compute authenticated shares of the cross terms $\gamma^0 \cdot \boldsymbol{y}^1$, $\gamma^1 \cdot \boldsymbol{y}^1$.

To prove that the commitments to $\gamma^i \cdot \boldsymbol{y}^i$ are consistent with the commitments to γ^i and $\boldsymbol{y}^i$ the parties can simply use a VOLE-ZK proof. Proving that the authenticated VOLE outputs are consistent with the committed inputs consists of two parts: (1) Check that the keys corresponding to the session-id $\mathsf{sid}_{\mathsf{VOLE}}$ used as input to the authenticated VOLE functionality are consistent with the committed keys γ^i, which can be done using a similar consistency check as the last three steps of initialization in Figure 8; (2) Check that the input vectors to the authenticated VOLE calls are consistent with the committed inputs $\boldsymbol{y}^i$, which can be done by using a consistency check on the outputs of the authenticated VOLE calls, which we will explain below.

Let $(\llbracket \boldsymbol{v} \rrbracket^{(q^r)}_{0,\mathsf{MAC}}, \llbracket \boldsymbol{w} \rrbracket^{(q^r)}_{1,\mathsf{MAC}}) \leftarrow \mathsf{aCommit}(\boldsymbol{y}^0, q^r, 1, \gamma^1, \Delta_0, \Delta_1)$, then we want to check that $\llbracket \boldsymbol{v} \rrbracket^{(q^r)}_{0,\mathsf{MAC}}$ is consistent with $\llbracket \boldsymbol{y}^0 \rrbracket^{(q^r)}_{0,\mathsf{MAC}}$. To do so, $\mathcal{P}_0$ commits to the MAC $\boldsymbol{M}_{\boldsymbol{y}^0} = \Delta_1 \cdot \boldsymbol{y}^0 + \boldsymbol{K}_{\boldsymbol{y}^0}$ under γ^1 to obtain $(\boldsymbol{e}, \boldsymbol{M}_{\boldsymbol{y}^0}), (\gamma^1, \boldsymbol{f})$ such that $\boldsymbol{e} + \boldsymbol{f} = \gamma^1 \cdot \boldsymbol{M}_{\boldsymbol{y}^0}$. Then the parties check if $\boldsymbol{e} - \boldsymbol{M}_{\boldsymbol{v}}$ is equal to $\Delta_1 \cdot \boldsymbol{w} - \boldsymbol{K}_{\boldsymbol{v}} + \gamma^1 \cdot \boldsymbol{K}_{\boldsymbol{y}^0} + \boldsymbol{f}$ using $\mathcal{F}_{\mathsf{EQ}}$. To see why this works, imagine that a corrupt $\mathcal{P}_0$ uses $\tilde{\boldsymbol{y}}^0 \neq \boldsymbol{y}^0$ as input to $\mathsf{aCommit}$ or uses $\tilde{\boldsymbol{M}} \neq \boldsymbol{M}_{\boldsymbol{y}^0}$ as input to Commit. This implies that

$$\begin{aligned}\Delta_1 \cdot \boldsymbol{w} - \boldsymbol{K}_{\boldsymbol{v}} + \gamma^1 \cdot \boldsymbol{K}_{\boldsymbol{y}^0} + \boldsymbol{f} &= \Delta_1 \cdot (\gamma^1 \cdot \tilde{\boldsymbol{y}}^0 - \boldsymbol{v}) - \boldsymbol{K}_{\boldsymbol{v}} + \gamma^1 \cdot \boldsymbol{K}_{\boldsymbol{y}^0} + \boldsymbol{e} - \gamma^1 \cdot \tilde{\boldsymbol{M}} \\ &= \boldsymbol{e} - \boldsymbol{M}_{\boldsymbol{v}} + \Delta_1 \cdot \gamma^1 \cdot (\tilde{\boldsymbol{y}}^0 - \boldsymbol{y}^0) + \gamma^1 \cdot (\boldsymbol{M}_{\boldsymbol{y}^0} - \tilde{\boldsymbol{M}}).\end{aligned}$$

So the only way for the corrupt party to pass the equality check is if they can guess a linear combination of $\Delta_1 \cdot \gamma^1$ and γ^1, which they are only be able to do with negligible probability. Note that the simulator can check correctness of such a guess using the "Special Guess" command in Figure 14. The consistency check for the other party's input $\boldsymbol{y}^1$ works analogously. The simulators can be constructed in a similar way to the proof of Theorem 3. □

Claim. The protocol $\Pi^{q,r}_{\mathsf{asiVOLE}}$ requires amortized communication of 4 $\mathbb{F}_{q^r}$ elements plus the cost of 2 calls to $\mathcal{F}^{p,r^{(2)},r^{(p)}}_{\mathsf{mRand}}$ (④), per entry in the VOLE. Instantiating the ideal functionality with $\Pi^{q,r}_{\mathsf{aVOLE}}$ gives a concrete efficiency of 4096 bit per entry in the authenticated VOLE over $\mathbb{F}_{2^{256}}$.

Fig. 16 Authenticated Shared-Input VOLE Protocol $\Pi^{q,r}_{\mathsf{asiVOLE}}$

Parameters: Two parties $\mathcal{P}_0$ and $\mathcal{P}_1$, inputs $\langle\!\langle \boldsymbol{y} \rangle\!\rangle^{(q^r)} \leftarrow (\llbracket \boldsymbol{y}^0 \rrbracket^{(q^r)}_{0,\mathsf{MAC}}, \llbracket \boldsymbol{y}^1 \rrbracket^{(q^r)}_{1,\mathsf{MAC}})$ and $\langle\!\langle \gamma \rangle\!\rangle^{(q^r)} \leftarrow (\llbracket \gamma^0 \rrbracket^{(q^r)}_{0,\mathsf{MAC}}, \llbracket \gamma^1 \rrbracket^{(q^r)}_{1,\mathsf{MAC}})$, where $\boldsymbol{y} \in \mathbb{F}_{q^r}^{\ell}$ ($\boldsymbol{y}$ can also be a vector over the base field $\mathbb{F}_q$, but to fit with our SOPRF protocol we use VOLE over the extension field) and $\gamma \in \mathbb{F}_{q^r}$. Length of authenticated VOLE correlation ℓ, base field characteristic $q \in \{2, p\}$ and extension field parameter $r = r^{(q)}$. $\mathsf{H} : \{0,1\}^* \to \{0,1\}^{\lambda}$ is a random oracle.

Generating Candidate Correlations:

- Send $(\mathsf{sid}_{\mathsf{MAC}}, \mathtt{init}, i, q)$ and $(\mathsf{sid}_{\mathsf{VOLE}}, \mathtt{init}, i, q)$ to $\mathcal{F}^{p,r^{(2)},r^{(p)}}_{\mathsf{mRand}}$, such that $\mathcal{P}_i$ obtains $\Delta_i, \gamma^i \in \mathbb{F}_{q^r}$, for $i \in \{0,1\}$.
- Execute $\langle\!\langle \gamma^1 \cdot \boldsymbol{y}^0 \rangle\!\rangle^{(q^r)} \leftarrow (\llbracket \boldsymbol{v} \rrbracket^{(q^r)}_{0,\mathsf{MAC}}, \llbracket \boldsymbol{w} \rrbracket^{(q^r)}_{1,\mathsf{MAC}}) \leftarrow \mathsf{aCommit}(\boldsymbol{y}^0, q^r, 1, \gamma^1, \Delta_0, \Delta_1)$ and $\langle\!\langle \gamma^0 \cdot \boldsymbol{y}^1 \rangle\!\rangle^{(q^r)} \leftarrow (\llbracket \boldsymbol{t} \rrbracket^{(q^r)}_{0,\mathsf{MAC}}, \llbracket \boldsymbol{u} \rrbracket^{(q^r)}_{1,\mathsf{MAC}}) \leftarrow \mathsf{aCommit}(\boldsymbol{y}^1, q^r, 1, \gamma^0, \Delta_1, \Delta_0)$.
- Query $\llbracket \gamma^i \cdot \boldsymbol{y}^i \rrbracket^{(q^r)}_i \leftarrow \mathsf{Commit}(\gamma^i \cdot \boldsymbol{y}^i, q^r, 1, \Delta_{i\oplus 1})$ for each $i \in \{0,1\}$.

Consistency Checks:

- To check that the keys from $\mathsf{sid}_{\mathsf{VOLE}}$ are consistent with $\llbracket \gamma^i \rrbracket^{(q^r)}_{i,\mathsf{MAC}}$, for each $i \in \{0,1\}$:
  - Execute $\llbracket \Delta_i \rrbracket^{(q^r)}_{i,\mathsf{VOLE}} \leftarrow \mathsf{Commit}(\Delta_i, q^r, 1, \gamma^{i\oplus 1})$ such that $\mathcal{P}_i$ obtains $\alpha^i \in \mathbb{F}_{q^r}$ and $\mathcal{P}_{i\oplus 1}$ obtains $\beta^i \in \mathbb{F}_{q^r}$ satisfying $\alpha^i + \beta^i = \gamma^{i\oplus 1} \cdot \Delta_i$.
  - Rewrite $\llbracket \gamma^{i\oplus 1} \rrbracket^{q^r}_{i\oplus 1,\mathsf{MAC}}$ such that $\mathcal{P}_{i\oplus 1}$ obtains $\delta^i \in \mathbb{F}_{q^r}$ and $\mathcal{P}_i$ obtains $\varepsilon^i \in \mathbb{F}_{q^r}$ satisfying $\delta^i + \varepsilon^i = \Delta_i \cdot \gamma^{i\oplus 1}$.
  - $\mathcal{P}_i$ sends $(\mathsf{sid}, \mathtt{check}, \alpha^i - \varepsilon^i, i)$ and $\mathcal{P}_{i\oplus 1}$ sends $(\mathsf{sid}, \mathtt{check}, \delta^i - \beta^i, i)$ to $\mathcal{F}_{\mathsf{EQ}}$.
- To prove that $\llbracket \gamma^i \cdot \boldsymbol{y}^i \rrbracket^{(q^r)}_i$ is consistent with $\llbracket \gamma^i \rrbracket^{(q^r)}_{i,\mathsf{MAC}} \cdot \llbracket \boldsymbol{y}^i \rrbracket^{(q^r)}_{i,\mathsf{MAC}}$, for each $i \in \{0,1\}$:
  - Send $(\mathsf{sid}_{\mathsf{MAC}}, \mathtt{prove}, \mathsf{id}^i_1, \ldots, \mathsf{id}^i_{2\ell+1}, C_1, \ldots, C_\ell, q)$ to $\mathcal{F}^{p,r^{(2)},r^{(p)}}_{\mathsf{mRand}}$, where id^i_1 corresponds to $\llbracket \gamma^i \rrbracket^{(q^r)}_{i,\mathsf{MAC}}$, $\mathsf{id}^i_2, \ldots, \mathsf{id}^i_{\ell+1}$ correspond to $\llbracket \boldsymbol{y}^i \rrbracket^{(q^r)}_{i,\mathsf{MAC}}$, $\mathsf{id}^i_{\ell+2}, \ldots, \mathsf{id}^i_{2\ell+1}$ correspond to $\llbracket \gamma^i \cdot \boldsymbol{y}^i \rrbracket^{(q^r)}_i$, and the circuits $C_j : \mathbb{F}^{2\ell+1}_{q^r} \to \mathbb{F}_{q^r}$ are given by $C_j(X_1, \ldots, X_{2\ell+1}) := X_1 \cdot X_{j+1} - X_{\ell+j+1}$.
- To check that $\llbracket \boldsymbol{v} \rrbracket^{(q^r)}_{0,\mathsf{MAC}}$ is consistent with $\llbracket \boldsymbol{y}^0 \rrbracket^{(q^r)}_{0,\mathsf{MAC}}$:
  - Execute $((\boldsymbol{e}, \boldsymbol{M}_{\boldsymbol{y}^0}), (\gamma^1, \boldsymbol{f})) := \llbracket \boldsymbol{M}_{\boldsymbol{y}^0} \rrbracket^{(q^r)}_{0,\mathsf{VOLE}} \leftarrow \mathsf{Commit}(\boldsymbol{M}_{\boldsymbol{y}^0}, q^r, 1, \gamma^1)$, i.e., such that $\boldsymbol{e} + \boldsymbol{f} = \gamma^1 \cdot \boldsymbol{M}_{\boldsymbol{y}^0}$
  - $\mathcal{P}_0$ computes $\tilde{e} := \mathsf{H}(\boldsymbol{e} - \boldsymbol{M}_{\boldsymbol{v}})$ and sends $(\mathsf{sid}, \mathtt{check}, \tilde{e}, 0)$, $\mathcal{P}_1$ computes $\tilde{f} := \mathsf{H}\left(\Delta_1 \cdot \boldsymbol{w} - \boldsymbol{K}_{\boldsymbol{v}} + \gamma^1 \cdot \boldsymbol{K}_{\boldsymbol{y}^0} + \boldsymbol{f}\right)$ and sends $(\mathsf{sid}, \mathtt{check}, \tilde{f}, 0)$ to $\mathcal{F}_{\mathsf{EQ}}$.
- To check that $\llbracket \boldsymbol{u} \rrbracket^{(q^r)}_{1,\mathsf{MAC}}$ is consistent with $\llbracket \boldsymbol{y}^1 \rrbracket^{(q^r)}_{1,\mathsf{MAC}}$:
  - Execute $((\gamma^0, \boldsymbol{d}), (\boldsymbol{c}, \boldsymbol{M}_{\boldsymbol{y}^1})) := \llbracket \boldsymbol{M}_{\boldsymbol{y}^1} \rrbracket^{(q^r)}_{1,\mathsf{VOLE}} \leftarrow \mathsf{Commit}(\boldsymbol{M}_{\boldsymbol{y}^1}, q^r, 1, \gamma^0)$, i.e., such that $\boldsymbol{c} + \boldsymbol{d} = \gamma^0 \cdot \boldsymbol{M}_{\boldsymbol{y}^1}$
  - $\mathcal{P}_1$ computes $\tilde{c} := \mathsf{H}(\boldsymbol{c} - \boldsymbol{M}_{\boldsymbol{u}})$ and sends $(\mathsf{sid}, \mathtt{check}, \tilde{c}, 1)$, $\mathcal{P}_0$ computes $\tilde{d} := \mathsf{H}\left(\Delta_0 \cdot \boldsymbol{t} - \boldsymbol{K}_{\boldsymbol{u}} + \gamma^0 \cdot \boldsymbol{K}_{\boldsymbol{y}^1} + \boldsymbol{d}\right)$ and sends $(\mathsf{sid}, \mathtt{check}, \tilde{d}, 1)$ to $\mathcal{F}_{\mathsf{EQ}}$.
- Put $\llbracket \boldsymbol{a} \rrbracket^{(q^r)}_{0,\mathsf{MAC}} := \llbracket \boldsymbol{v} \rrbracket^{(q^r)}_{0,\mathsf{MAC}} + \llbracket \boldsymbol{t} \rrbracket^{(q^r)}_{0,\mathsf{MAC}} + \llbracket \gamma^0 \cdot \boldsymbol{y}^0 \rrbracket^{(q^r)}_0$ and $\llbracket \boldsymbol{b} \rrbracket^{(q^r)}_{1,\mathsf{MAC}} := \llbracket \boldsymbol{w} \rrbracket^{(q^r)}_{1,\mathsf{MAC}} + \llbracket \boldsymbol{u} \rrbracket^{(q^r)}_{1,\mathsf{MAC}} + \llbracket \gamma^1 \cdot \boldsymbol{y}^1 \rrbracket^{(q^r)}_{1,\mathsf{MAC}}$, and output $\langle\!\langle \gamma \cdot \boldsymbol{y} \rangle\!\rangle^{(q^r)} \leftarrow (\llbracket \boldsymbol{a} \rrbracket^{(q^r)}_{0,\mathsf{MAC}}, \llbracket \boldsymbol{b} \rrbracket^{(q^r)}_{1,\mathsf{MAC}})$.

Shared-Input Shared-Output Protocol.

Theorem 10. *The protocol* $\Pi^{\mathsf{M}}_{\mathsf{SISO\text{-}PRF}}$ *in Figure 17 realizes the ideal functionality* $\mathcal{F}^{p,r^{(2)},r^{(p)}}_{\mathsf{mRand}}$ (⑨) *against malicious adversaries in the* $\mathcal{F}^{p,r^{(2)},r^{(p)}}_{\mathsf{mRand}}$ (⓪, ③, ⑦, ⑧)*-hybrid model, assuming* $n = d+2(n'-d)+1$, $r^{(2)} = n \geq \kappa$ *and* $r^{(3)} \geq \kappa/\log_2 3$.

Proof. The main differences with the OPRF protocol in Figure 9 are: (1) The input encoding is computed jointly on the secret-shared inputs $[\![\boldsymbol{x}^j]\!]^{(2)}$ instead of being computed locally by $\mathcal{P}_0$ and checked using VOLE-ZK; (2) The key-multiplication is computed using authenticated shared-input VOLE instead of authenticated VOLE since the inputs are secret-shared and authenticated. It is easy to see that the second change does not influence the correctness of the protocol, but the first change warrants some explanation.

The conversion from the modulo 2 sharing of the input to a modulo 3 sharing of the input is done similar to the final modulus conversion using daBits. The multiplication by the code $\boldsymbol{G}$ can simply be done locally on the modulo 3 shares. To evaluate the decomp operator on the secret shares, we note that for $x = x_0 + x_1 \cdot 2 \in \mathbb{Z}_3$, with $x_0, x_1 \in \{0,1\}$ and $x_0 \wedge x_1 = 0$, it holds that

$$x^2 \equiv x_0 + x_1 \bmod 3 \implies 2x + 2x^2 \equiv x_1 \bmod 3 \quad \text{and} \quad x + 2x^2 \equiv x_0 \bmod 3.$$

So we can compute the binary composition using a single $\mathbb{F}_3$-multiplication triple for each entry. Finally the conversion from the secret-shared binary values modulo 3 to a secret-sharing modulo 2 can be done similarly to the conversion in the other direction using daBits, with the only difference that now we need to compute the XOR modulo 3 on secret-shared values before opening the masked value, which again requires $\mathbb{F}_3$-multiplication triples. After obtaining the secret-sharing of the encoded inputs modulo 2, correctness follows similarly to Figure 9. The simulators can be constructed similarly to the proof of Theorem 4. □

Claim. The protocol $\Pi^{\mathsf{M}}_{\mathsf{SISO\text{-}PRF}}$ requires amortized communication of $2 \cdot d + 2 \cdot n$ $\mathbb{F}_2$ elements and $2(n'-d) + 2(d + 2(n'-d)) + d + 2(n'-d)$ $\mathbb{F}_3$ elements per evaluation. Additional $n + n - 1 + d$ daBits, $n - 1 + (n'-d)$ authenticated triples and 1 authenticated secret shared VOLE entry are required as building blocks per evaluation.

To compute the overhead with respect to $\Pi^{\mathsf{M}}_{\mathsf{OPRF}}$, the implementation parameters was selected. Based on experiments we assume 1 bit of communication is needed for each entry in a VOLE correlations. This gives a cost of 4.95 KB per evaluation for $\Pi^{\mathsf{M}}_{\mathsf{OPRF}}$, where $\Pi^{\mathsf{M}}_{\mathsf{SISO\text{-}PRF}}$ requires 21.6 KB per evaluation. This gives a communication overhead of 4.36×. Almost all of both VOLE correlations usage and communication comes from the additional daBit used and the authenticated triple. Thus a similar computation overhead would be expected.

6.5 (Towards) Verifiable OPRF

For some applications, notably Privacy Pass [24], we need clients to be able to verify that the server is using the same key for different evaluations. We can further distinguish between *private* verifiabilty and *public* verifiability. The former meaning that a *single* client can verify whether the server is using the same key for multiple evaluations or multiple sessions, and the latter meaning that *multiple* independent clients can verify this.

Private Verifiability. We note that private verifiability comes almost for free for our construction. Within a single session of the protocol in Figure 9, with the session-id $\mathsf{sid} = (\mathsf{sid}_{\mathsf{MAC}}, \mathsf{sid}_{\mathsf{Key}})$, the

Fig. 17 Shared-Input Shared-Output PRF protocol $\Pi^{\mathsf{M}}_{\mathsf{SISO\text{-}PRF}}$

Parameters: Two parties $\mathcal{P}_0$ and $\mathcal{P}_1$ holding secret sharings $(\langle\!\langle \boldsymbol{x}^j \rangle\!\rangle^{(2)})_{j\in[\ell]}$, $\langle\!\langle k \rangle\!\rangle^{(2^n)}$ with $\boldsymbol{x}^j \in \mathbb{F}_2^d$ and $k \in \mathbb{F}_{2^n}$, authenticated under the parties' MAC keys $\Delta_0^{(2)}, \Delta_1^{(2)} \in \mathbb{F}_{2^n}$, which correspond to the session-id $\mathsf{sid}_{\mathsf{MAC}}$ of $\mathcal{F}_{\mathsf{mRand}}^{3,r^{(2)},r^{(3)}}$. Moreover, the parties' key-shares $k^0, k^1 \in \mathbb{F}_{2^n}$ correspond to the session-id $\mathsf{sid}_{\mathsf{Key}}$ of $\mathcal{F}_{\mathsf{mRand}}^{3,r^{(2)},r^{(3)}}$. Public matrix $\boldsymbol{B} \in \mathbb{F}_3^{t\times n}$, linear code $\boldsymbol{G} \in \mathbb{F}_3^{n'\times d}$, integers $n, t, d, n', r^{(2)}, r^{(3)} \in \mathbb{N}$ depending on the security parameter $\lambda \in \mathbb{N}$ with $r^{(2)} = n$ and $d + 2(n' - d) + 1 = n$.

Preprocessing:

- For each $i \in \{0,1\}$, send $(\mathsf{sid}_{\mathsf{MAC}}, \mathtt{init}, i, q)$ and $(\mathsf{sid}_{\mathsf{Key}}, \mathtt{init}, i, q)$ to $\mathcal{F}_{\mathsf{mRand}}^{3,r^{(2)},r^{(3)}}$ for each $q \in \{2,3\}$. $\mathcal{P}_i$ receives $\Delta_i^{(2)} \in \mathbb{F}_{2^{r^{(2)}}}$, $\Delta_i^{(3)} \in \mathbb{F}_{3^{r^{(3)}}}$ and $k^i \in \mathbb{F}_{2^n}$, $\Delta_{k^i}^{(3)} \in \mathbb{F}_{3^{r^{(3)}}}$, respectively.
- Send $(\mathsf{sid}_{\mathsf{MAC}}, \mathtt{daBits}, (n + 2d + 2(n' - d)) \cdot \ell)$ to $\mathcal{F}_{\mathsf{mRand}}^{3,r^{(2)},r^{(3)}}$ and receive $\langle\!\langle \boldsymbol{b}^j \rangle\!\rangle^{(2)}$, $\langle\!\langle \boldsymbol{b}^j \rangle\!\rangle^{(3)}$, $\langle\!\langle \boldsymbol{b'}^j \rangle\!\rangle^{(2)}$, $\langle\!\langle \boldsymbol{b'}^j \rangle\!\rangle^{(3)}$ and $\langle\!\langle \tilde{\boldsymbol{b}}^j \rangle\!\rangle^{(2)}$, $\langle\!\langle \tilde{\boldsymbol{b}}^j \rangle\!\rangle^{(3)}$ for $\boldsymbol{b}^j \in \{0,1\}^n$, $\boldsymbol{b'}^j \in \{0,1\}^d$ and $\tilde{\boldsymbol{b}}^j \in \{0,1\}^{d+2(n'-d)}$, all for each $j \in [\ell]$.
- Send $(\mathsf{sid}_{\mathsf{MAC}}, \mathtt{triples}, 3, (d + 3(n' - d)) \cdot \ell)$ to $\mathcal{F}_{\mathsf{mRand}}^{3,r^{(2)},r^{(3)}}$ and receive $(\langle\!\langle \boldsymbol{d}^j \rangle\!\rangle^{(3)}, \langle\!\langle \boldsymbol{e}^j \rangle\!\rangle^{(3)}, \langle\!\langle \boldsymbol{f}^j \rangle\!\rangle^{(3)})$ with $\boldsymbol{d}^j, \boldsymbol{e}^j, \boldsymbol{f}^j \in \mathbb{F}_3^{d+3(n'-d)}$ for each $j \in [\ell]$.

Key-Multiplication and Input-Encoding:

- Compute $\langle\!\langle \boldsymbol{c'}^j \rangle\!\rangle^{(2)} := \langle\!\langle \boldsymbol{x}^j \rangle\!\rangle^{(2)} + \langle\!\langle \boldsymbol{b'}^j \rangle\!\rangle^{(2)}$ and $(\boldsymbol{c'}^1, \ldots, \boldsymbol{c'}^\ell) \leftarrow \mathsf{BatchOpen}\Big((\langle\!\langle \boldsymbol{c'}^1 \rangle\!\rangle^{(2)}, \ldots, \langle\!\langle \boldsymbol{c'}^\ell \rangle\!\rangle^{(2)}), \{0,1\}\Big)$. Then convert to sharings mod 3 as $\langle\!\langle \boldsymbol{x}^j \rangle\!\rangle^{(3)} := \boldsymbol{c'}^j + \langle\!\langle \boldsymbol{b'}^j \rangle\!\rangle^{(3)} + \boldsymbol{c'}^j \odot \langle\!\langle \boldsymbol{b'}^j \rangle\!\rangle^{(3)}$.
- The parties compute $\langle\!\langle \boldsymbol{x'}^j \rangle\!\rangle^{(3)} := \boldsymbol{G} \cdot \langle\!\langle \boldsymbol{x}^j \rangle\!\rangle^{(3)}$ and compute $\langle\!\langle \mathsf{decomp}(\boldsymbol{G} \cdot_3 \boldsymbol{x}^j) \rangle\!\rangle^{(3)}$ as

$$\langle\!\langle \boldsymbol{y}^j \rangle\!\rangle^{(3)} := \langle\!\langle \big(\boldsymbol{x'}^j_{[1,d]}, 2 \cdot \boldsymbol{x'}^j_{[d+1,n']} + 2 \cdot (\boldsymbol{x'}^j_{[d+1,n']})^2, \boldsymbol{x'}^j_{[d+1,n']} + 2 \cdot (\boldsymbol{x'}^j_{[d+1,n']})^2\big) \rangle\!\rangle^{(3)},$$

  using $(n' - d)$ multiplication triples to compute $\langle\!\langle (\boldsymbol{x'}^j_{[d+1,n']})^2 \rangle\!\rangle^{(3)}$ for each $j \in [\ell]$.
- Compute $\langle\!\langle \tilde{\boldsymbol{c}}^j \rangle\!\rangle^{(3)} := \langle\!\langle \boldsymbol{y}^j \rangle\!\rangle^{(3)} + \langle\!\langle \tilde{\boldsymbol{b}}^j \rangle\!\rangle^{(3)} + \langle\!\langle \boldsymbol{y}^j \rangle\!\rangle^{(3)} \odot \langle\!\langle \tilde{\boldsymbol{b}}^j \rangle\!\rangle^{(3)}$ using $d+2(n'-d)$ multiplication triples to compute the multiplication for each $j \in [\ell]$. Execute $(\tilde{\boldsymbol{c}}^1, \ldots, \tilde{\boldsymbol{c}}^\ell) \leftarrow \mathsf{BatchOpen}((\langle\!\langle \tilde{\boldsymbol{c}}^1 \rangle\!\rangle^{(3)}, \ldots, \langle\!\langle \tilde{\boldsymbol{c}}^j \rangle\!\rangle^{(3)}), \{0,1\})$ and compute $\langle\!\langle \boldsymbol{y}^j \rangle\!\rangle^{(2)} := \tilde{\boldsymbol{c}}^j + \langle\!\langle \tilde{\boldsymbol{b}}^j \rangle\!\rangle^{(2)}$, followed by $\langle\!\langle y^j \rangle\!\rangle^{2^n} := \mathsf{elt}_\alpha(\langle\!\langle \boldsymbol{y}^j \| 1 \rangle\!\rangle^{(2)})$ (where appending a public entry can be done similarly to adding a public constant).
- Send $\big(\mathsf{sid}_{\mathsf{Key}}, \mathsf{sid}_{\mathsf{MAC}}, \mathtt{asiVOLE}, (\langle\!\langle y^1 \rangle\!\rangle^{(2^n)}, \ldots, \langle\!\langle y^\ell \rangle\!\rangle^{(2^n)}), \langle\!\langle k \rangle\!\rangle^{(2^n)}\big)$ to $\mathcal{F}_{\mathsf{mRand}}^{3,r^{(2)},r^{(3)}}$ and receive $\langle\!\langle k \cdot y^j \rangle\!\rangle^{(2^n)}$, $j \in [\ell]$.

Modulus Conversion and Final Matrix Multiplication:

- Compute $\langle\!\langle b^j \rangle\!\rangle^{(2^n)} := \mathsf{elt}_\alpha(\langle\!\langle \boldsymbol{b}^j \rangle\!\rangle^{(2)})$, $\langle\!\langle c^j \rangle\!\rangle^{(2^n)} \leftarrow \langle\!\langle k \cdot y^j \rangle\!\rangle^{(2^n)} + \langle\!\langle b^j \rangle\!\rangle^{(2^n)}$, for $j \in [\ell]$, and execute $(c_1, \ldots, c^\ell) \leftarrow \mathsf{BatchOpen}\Big((\langle\!\langle c^1 \rangle\!\rangle^{(2^n)}, \ldots, \langle\!\langle c^j \rangle\!\rangle^{(2^n)}), \{0,1\}\Big)$. Then convert to sharings of $\mathsf{vec}_\alpha(k \cdot y^j)$ modulo 3 as $\langle\!\langle \mathsf{vec}_\alpha(k \cdot y^j) \rangle\!\rangle^{(3)} = \mathsf{vec}_\alpha(c^j) + \langle\!\langle \boldsymbol{b}^j \rangle\!\rangle^{(3)} + \mathsf{vec}_\alpha(c^j) \odot \langle\!\langle \boldsymbol{b}^j \rangle\!\rangle^{(3)}$.
- Compute and output $\langle\!\langle F_k(\boldsymbol{x}^j) \rangle\!\rangle^{(3)} = \boldsymbol{B} \cdot_3 \langle\!\langle \mathsf{vec}_\alpha(k \cdot y^j) \rangle\!\rangle^{(3)}$.

client $\mathcal{P}_0$ is guaranteed that the server $\mathcal{P}_1$ is using the same PRF key k for *all* evaluations by definition of the authenticated VOLE functionality $\mathcal{F}_{\mathsf{mRand}}^{3,n,r^{(3)}}$ (④).

If the client wants to verify that the server is using the same PRF key during two different sessions $\mathsf{sid}, \mathsf{sid}'$, this can be done similar to [59]. That is, let the client sample $u \xleftarrow{\$} \mathbb{F}_{2^n}$ and query an additional $\mathsf{Commit}(u, 2^n, 1, k)$ to $\mathcal{F}_{\mathsf{mRand}}^{3,n,r^{(3)}}$ with session-id $\mathsf{sid}_{\mathsf{Key}}$ (resp. $\mathsf{Commit}(u, 2^n, 1, k')$ with session-id $\mathsf{sid}_{\mathsf{Key}'}$) such that $\mathcal{P}_0$ obtains $w \in \mathbb{F}_{2^n}$ (resp. $w' \in \mathbb{F}_{2^n}$) and $\mathcal{P}_1$ obtains $v \in \mathbb{F}_{2^n}$ (resp. $v' \in \mathbb{F}_{2^n}$), satisfying $w + v = k \cdot u$ (resp. $w' + v' = k' \cdot u$). To check whether $k = k'$, the client now sends $(\mathsf{sid}, \texttt{check}, w - w', 0)$ and the server sends $(\mathsf{sid}, \texttt{check}, v' - v, 0)$ to $\mathcal{F}_{\mathsf{EQ}}$. If any of the parties receive `abort` from $\mathcal{F}_{\mathsf{EQ}}$, they abort. It is easy to see that the check passes if $k = k'$, and that the server can only cheat if they manage to guess u correctly, which happens with negligible probability. Note that a malicious client can cheat on the equality check if they manage to guess k correctly, which is covered by the **Key-Guess** query in our $\mathcal{F}_{\mathsf{OPRF}}$ functionality.

Public Verifiability. First note that if we are in a setting where different clients can communicate to each other through a private channel, and we assume that no client can simultaneously be corrupted with the server, it is easy to achieve verifiability between clients by simply letting them exchange the checking information $u, w \in \mathbb{F}_{2^n}$. However, this is not always a realistic scenario and ideally we want any client to be able to verify that the server is using a specific key based on some public verification information VK which the server publishes and authenticates.

To achieve this, we follow the approach sketched in [3]. That is, first of all, the server publishes the evaluation $\boldsymbol{z}^{*,j} := F_k(\boldsymbol{x}^{*,j})$ of the PRF on $j \in [\lambda]$ public input points $\boldsymbol{x}^{*,j} \in \mathbb{F}_2^d$ together with public-verifier NIZK proof π that they performed the evaluation correctly, i.e., $\mathsf{VK} := \{(\boldsymbol{x}^{*,j}, \boldsymbol{z}^{*,j})_{j\in[\lambda]}, \pi\}$. To obtain the PRF evaluations on ℓ input points, the client submits $\nu = C + \ell$ inputs to the OPRF protocol, consisting of their original ℓ input points together with $C \in \mathbb{N}$ points randomly selected from $\{\boldsymbol{x}^{*,j}\}_{j\in[\lambda]}$. Upon receiving the outputs, the client checks if the evaluations on the public input points match the public output points. Note that in [3], the receiver additionally inserts multiple copies of each of their input points and checks if evaluations on the same input are equal, but this is not needed for out construction since the server is guaranteed to use a single key to evaluate the entire batch.

Recent cryptanalysis [22] shows that the above approach is not as secure as expected for the FHE-based protocol [3], which significantly increased the parameters. However, this attack crucially makes use of the fact that the server can evaluate arbitrary circuits on the FHE ciphertexts submitted by the client. Due to the structure of our protocol (Fig. 9), the only way the server can attempt to cheat is by using a different key $k' \in \mathbb{F}_{2^n}$ than the key $k \in \mathbb{F}_{2^n}$ they used to evaluate the checkpoints $\boldsymbol{z}^{*,j} := F_k(\boldsymbol{x}^{*,j})$. The only thing one has to show is that it is hard to find $k, k' \in \mathbb{F}_{2^n}$ such that $F_k(\boldsymbol{x}^{*,j_i}) = F_{k'}(\boldsymbol{x}^{*,j_i})$ for all $i \in [C]$, where $j_1, \ldots, j_C \xleftarrow{\$} [\lambda]$ is negligible in λ. We expect this to be hard, but leave it to future work to determine a suitable choice of $C \geq 2$ and efficiently instantiate the NIZK proof.

Strong UC Security. We recall that the ideal OPRF functionality needed to instantiate the OPAQUE PAKE protocol [40] is strictly stronger than the OPRF functionality $\mathcal{F}_{\mathsf{OPRF}}$ (Fig. 1) realized by our protocol. In the sense that, the ideal functionality [40] acts as a perfectly random independent function for each choice of server key k, which is impossible to realize without a random oracle. Beullens et al. [14] develop a framework that realizes this functionality from a random oracle and a functionality that securely evaluates a function satisfying so-called "*one-more unpredictability*" and "*weak key-collision resistance*" properties. This "*weak key-collision resistance*"

property basically translates to the problem discussed in the previous paragraph for $C = 1$, which is unfortunately not satisfied by our PRF, making our construction not directly compatible with the 2Hash framework [14].

7 Implementation and Benchmarks

We implemented our client-malicious and fully malicious OPRF protocols in Rust (see Section 6.2 and 6.1), as an extension of the Swanky library [32]. In this paper, we use the notation $1\text{KB} = 10^3\text{B}$. All numbers from related work are modified to follow the same convention. The implementation can be found in our repository.[5]

7.1 Setup

Parameter set: The parameter set chosen for the PRF is the most common (optimistic) parameters [3, 15, 25], $(n, t) = (2\lambda, \lambda/\log_2(3))$. For a computational security of 128-bits, we set parameters $(n, t) = (256, 82)$ and pick $(n', d) = (191, 128)$ for the input-encoding, as in [3].

Primitive instantiation: To generate the large number of VOLE correlations needed, we use the VOLE expansion of [57] to expand a small number of base VOLEs into a large number of VOLE correlations. This expansion cost amortizes to cost 1 bit per element. The VOLE expansion relies on OTs, where we implement the base OTs using [45] instantiated with ML-KEM, and expand these base OTs using the extension protocol from [42].

Batching: The OPRF is benchmarked for a range of different batch sizes. Since multiple building blocks rely on the cut-and-choose approach for security, the implementation selected the best parameters for 40-bit statistical security.

Hardware: All our experiments are executed on a computer running Manjaro Linux with an AMD Ryzen 7 PRO 4750U 8-core processor with 24 GiB of RAM. The implementation utilizes AVX2 for improved performance. Client and server run on the same CPU in different threads.

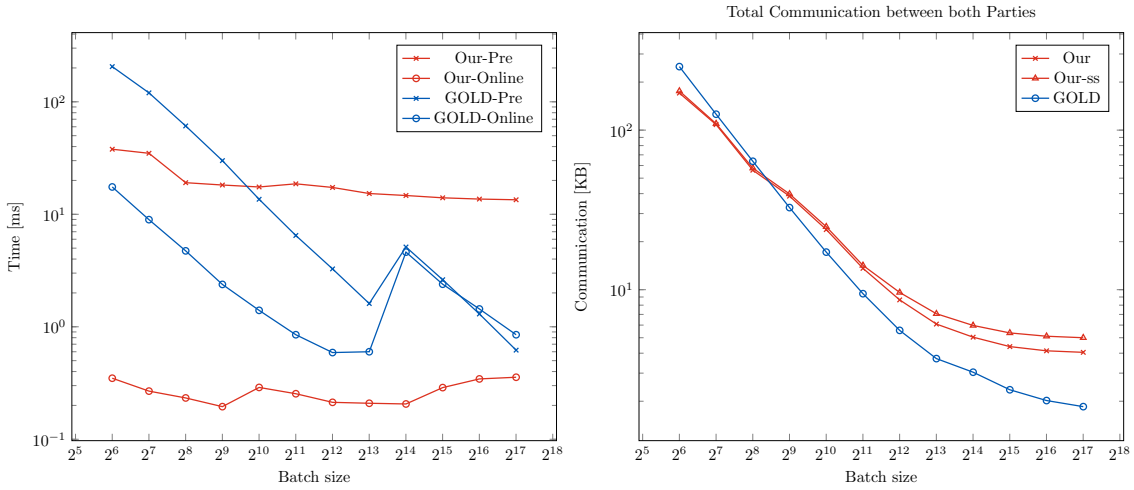
Security (S-C)	Batch Size	Comm.	Time
◐ - ●	1	2019 KB	$125 + 5.87$ ms
	2^6	68.2 KB	$3.69 + 0.46$ ms
	2^{17}	553 B	$1.85 + 0.26$ ms
● - ●	1	3824 KB	$239 + 3.09$ ms
	2^6	171 KB	$37.9 + 0.35$ ms
	2^{17}	4.03 KB	$13.5 + 0.36$ ms
● - ● Secret-shared output	1	4536 KB	$271 + 3.11$ ms
	2^6	176 KB	$43.6 + 0.33$ ms
	2^{17}	5.00 KB	$16.5 + 0.37$ ms

Table 2: Performance of our OPRF implementations. ◐ denotes semi-honest security where ● denotes malicious security. The timing is presented as (Offline + Online) per evaluation. The communication is the total communication per evaluation.

[5] https://github.com/OpenMPCC/Crypto-Dark-Matter-OPRF-public

7.2 Benchmarks

We now present the benchmarks for our OPRF implementation. We divide the running time of the evaluation into two distinct phases. The preprocessing phase, which is input independent, and the online phase. We focus only on the case of a uniformly random server key. The key can be selected by the server by sending an additional $\mathbb{F}_{2^{256}}$ element. In Table 2 the results of three batch sizes are shown for the malicious client, the fully malicious, and the secret-shared output OPRF. The small batch sizes have significantly higher cost per evaluation due to the one-time setup of base OT's. The large batch size shows the amortized cost of each evaluation, as in this setting the setup cost is negligible. The secret-shared output variant is more expensive, since it cannot utilize the optimization from Remark 2. As the OPRF is black-box in the OTs, VOLE and daBits, any future improvement in these protocols will have a direct impact on the performance of our OPRF.



(a) Runtime comparison

(b) Communication comparison

Fig. 18: Comparison of amortized offline time (x) and online time (o) (Fig. 18a) and comparison of amortized total communication (Fig. 18b) for a range of batch sizes between our OPRF(● - ●) and the GOLD OPRF(● - ●) [59]. Our secret shared output variant is excluded in (Fig. 18a) as it performs similarly.

7.3 Comparison to Other OPRF

Comparing with the state of the art OPRF, GOLD OPRF [59], we perform benchmarks on the same setup to have directly comparable numbers. In Figure 18a a detailed comparison of timings is shown for different batch sizes. The figure shows that our OPRF have a faster total time for batch sizes below 2^{10}, and have a faster online phase for all batch sizes. It is important to note the our OPRF has an almost flat performance curve, showing that the per-evaluation time is almost independent of the batch size. In Figure 18b a similar detailed comparison of communication is shown. Here, it becomes clear that the large batch size cost of GOLD, 1.85KB, is 2.2× less than

our construction. However, for batch sizes below 2^9 our construction becomes more communication efficient.

Comparing with OPRFs based on isogeny assumptions, the best known malicious OPRF is [9], requiring 8.7MB of communication, which is 2.3× more communication than ours. There is no implementation to compare timings with, however, the isogeny based semi-honest OPRF by [36] runs in ≈ 9s which is almost 3 order of magnitude slower. We expect a similar result for the malicious case.

Our OPRF becomes the new state-of-the-art malicious OPRF based on the mod(2,3) assumption. The OPRF by [3] using FHE requires 10KB and 151ms per evaluation, which we outperform by 2.5× and 8×, respectively.

7.4 Potential Improvement Based on Recent Work

The main contributor to communication of our OPRF is the protocol that generates authenticated OLE in Figure 21. Amortizing over large enough batch sizes, each authenticated OLE requires approximately 47 $\mathbb{F}_3$ elements and 27 OLEs, giving a communication cost of 13B. As 256 are consumed in each evaluation as daBits, this constitutes 82% of the communication, specifically, 3.3KB. Recent work by [44] shows a protocol for generating authenticated multiplication triples using only 1.1B per triple over $\mathbb{F}_2$. The communication scales linearly with $\log p$ for general p, thus, for triples over $\mathbb{F}_3$ a cost of 1.8B is expected. Using this protocol, our batched OPRF communication reduces to 1.11KB.

For small batch sizes, the cost of our OPRF is dominated by the one-time setup. By changing the base OT protocol to [8], the initial setup cost can be reduced by 197KB. Furthermore, utilizing the OT extension from [51] the setup can be further reduced by 537KB, at a small increase in computation time. These changes will not have any impact on the amortized cost of our OPRF.

Acknowledgements.

This work has been supported by funding from *Agentur für Innovation in der Cybersicherheit GmbH* under the project "Multi-Party Computation in the Confidential Cloud (MPCC)" within the Encrypted Computing – Analysis & Processing of Encrypted Data (EC2) Program.

References

1. Agarwal, A., Baum, C., Braun, L., Scholl, P.: Low-bandwidth mixed arithmetic in VOLE-based ZK from low-degree PRGs. In: Fehr, S., Fouque, P.A. (eds.) Advances in Cryptology – EUROCRYPT 2025, Part IV. Lecture Notes in Computer Science, vol. 15604, pp. 396–426. Springer, Cham, Switzerland, Madrid, Spain (May 4–8, 2025). https://doi.org/10.1007/978-3-031-91134-7_14
2. Alamati, N., Policharla, G.V., Raghuraman, S., Rindal, P.: Improved alternating-moduli PRFs and post-quantum signatures. In: Reyzin, L., Stebila, D. (eds.) Advances in Cryptology – CRYPTO 2024, Part VIII. Lecture Notes in Computer Science, vol. 14927, pp. 274–308. Springer, Cham, Switzerland, Santa Barbara, CA, USA (Aug 18–22, 2024). https://doi.org/10.1007/978-3-031-68397-8_9
3. Albrecht, M.R., Davidson, A., Deo, A., Gardham, D.: Crypto dark matter on the torus - oblivious PRFs from shallow PRFs and TFHE. In: Joye, M., Leander, G. (eds.) Advances in Cryptology – EUROCRYPT 2024, Part VI. Lecture Notes in Computer Science, vol. 14656, pp. 447–476. Springer, Cham, Switzerland, Zurich, Switzerland (May 26–30, 2024). https://doi.org/10.1007/978-3-031-58751-1_16

4. Albrecht, M.R., Davidson, A., Deo, A., Smart, N.P.: Round-optimal verifiable oblivious pseudorandom functions from ideal lattices. In: Garay, J. (ed.) PKC 2021: 24th International Conference on Theory and Practice of Public Key Cryptography, Part II. Lecture Notes in Computer Science, vol. 12711, pp. 261–289. Springer, Cham, Switzerland, Virtual Event (May 10–13, 2021). https://doi.org/10.1007/978-3-030-75248-4_10
5. Albrecht, M.R., Gür, K.D.: Verifiable oblivious pseudorandom functions from lattices: Practical-ish and thresholdisable. In: Chung, K.M., Sasaki, Y. (eds.) Advances in Cryptology – ASIACRYPT 2024, Part IV. Lecture Notes in Computer Science, vol. 15487, pp. 205–237. Springer, Singapore, Singapore, Kolkata, India (Dec 9–13, 2024). https://doi.org/10.1007/978-981-96-0894-2_7
6. Aly, A., Orsini, E., Rotaru, D., Smart, N.P., Wood, T.: Zaphod: Efficiently combining LSSS and garbled circuits in SCALE. In: WAHC@CCS. pp. 33–44. ACM (2019). https://doi.org/10.1145/3338469.3358943
7. Ayala, I.M., Raddum, H.: Zeroed out: Cryptanalysis of weak PRFs in alternating moduli. IACR Transactions on Symmetric Cryptology **2025**(2), 1–15 (2025). https://doi.org/10.46586/tosc.v2025.i2.1-15
8. Badrinarayanan, S., Masny, D., Mukherjee, P.: Efficient and tight oblivious transfer from PKE with tight multi-user security. In: Ateniese, G., Venturi, D. (eds.) ACNS 2022: 20th International Conference on Applied Cryptography and Network Security. Lecture Notes in Computer Science, vol. 13269, pp. 626–642. Springer, Cham, Switzerland, Rome, Italy (Jun 20–23, 2022). https://doi.org/10.1007/978-3-031-09234-3_31
9. Basso, A.: A post-quantum round-optimal oblivious PRF from isogenies. In: Carlet, C., Mandal, K., Rijmen, V. (eds.) SAC 2023: 30th Annual International Workshop on Selected Areas in Cryptography. Lecture Notes in Computer Science, vol. 14201, pp. 147–168. Springer, Cham, Switzerland, Fredericton, Canada (Aug 14-18, 2024). https://doi.org/10.1007/978-3-031-53368-6_8
10. Baum, C., Beullens, W., Mukherjee, S., Orsini, E., Ramacher, S., Rechberger, C., Roy, L., Scholl, P.: One tree to rule them all: Optimizing GGM trees and OWFs for post-quantum signatures. In: Chung, K.M., Sasaki, Y. (eds.) Advances in Cryptology – ASIACRYPT 2024, Part I. Lecture Notes in Computer Science, vol. 15484, pp. 463–493. Springer, Singapore, Singapore, Kolkata, India (Dec 9–13, 2024). https://doi.org/10.1007/978-981-96-0875-1_15
11. Baum, C., Braun, L., Munch-Hansen, A., Razet, B., Scholl, P.: Appenzeller to brie: Efficient zero-knowledge proofs for mixed-mode arithmetic and Z2k. In: Vigna, G., Shi, E. (eds.) ACM CCS 2021: 28th Conference on Computer and Communications Security. pp. 192–211. ACM Press, Virtual Event, Republic of Korea (Nov 15–19, 2021). https://doi.org/10.1145/3460120.3484812
12. Beaver, D.: Efficient multiparty protocols using circuit randomization. In: Feigenbaum, J. (ed.) Advances in Cryptology – CRYPTO'91. Lecture Notes in Computer Science, vol. 576, pp. 420–432. Springer Berlin Heidelberg, Germany, Santa Barbara, CA, USA (Aug 11–15, 1992). https://doi.org/10.1007/3-540-46766-1_34
13. Bendlin, R., Damgård, I., Orlandi, C., Zakarias, S.: Semi-homomorphic encryption and multiparty computation. In: Paterson, K.G. (ed.) Advances in Cryptology – EUROCRYPT 2011. Lecture Notes in Computer Science, vol. 6632, pp. 169–188. Springer Berlin Heidelberg, Germany, Tallinn, Estonia (May 15–19, 2011). https://doi.org/10.1007/978-3-642-20465-4_11
14. Beullens, W., Dodgson, L., Faller, S.H., Hesse, J.: The 2Hash OPRF framework and efficient post-quantum instantiations. In: Fehr, S., Fouque, P.A. (eds.) Advances in Cryptology – EUROCRYPT 2025, Part VIII. Lecture Notes in Computer Science, vol. 15608, pp. 332–362. Springer, Cham, Switzerland, Madrid, Spain (May 4–8, 2025). https://doi.org/10.1007/978-3-031-91101-9_12
15. Boneh, D., Ishai, Y., Passelègue, A., Sahai, A., Wu, D.J.: Exploring crypto dark matter: New simple PRF candidates and their applications. In: Beimel, A., Dziembowski, S. (eds.) TCC 2018: 16th Theory of Cryptography Conference, Part II. Lecture Notes in Computer Science, vol. 11240, pp. 699–729. Springer, Cham, Switzerland, Panaji, India (Nov 11–14, 2018). https://doi.org/10.1007/978-3-030-03810-6_25
16. Boneh, D., Kogan, D., Woo, K.: Oblivious pseudorandom functions from isogenies. In: Moriai, S., Wang, H. (eds.) Advances in Cryptology – ASIACRYPT 2020, Part II. Lecture Notes in Computer

Science, vol. 12492, pp. 520–550. Springer, Cham, Switzerland, Daejeon, South Korea (Dec 7–11, 2020). https://doi.org/10.1007/978-3-030-64834-3_18
17. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y.: Compressing vector OLE. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) ACM CCS 2018: 25th Conference on Computer and Communications Security. pp. 896–912. ACM Press, Toronto, ON, Canada (Oct 15–19, 2018). https://doi.org/10.1145/3243734.3243868
18. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., Rindal, P., Scholl, P.: Efficient two-round OT extension and silent non-interactive secure computation. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) ACM CCS 2019: 26th Conference on Computer and Communications Security. pp. 291–308. ACM Press, London, UK (Nov 11–15, 2019). https://doi.org/10.1145/3319535.3354255
19. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: 42nd Annual Symposium on Foundations of Computer Science. pp. 136–145. IEEE Computer Society Press, Las Vegas, NV, USA (Oct 14–17, 2001). https://doi.org/10.1109/SFCS.2001.959888
20. Casacuberta, S., Hesse, J., Lehmann, A.: SoK: Oblivious pseudorandom functions. In: 2022 IEEE European Symposium on Security and Privacy. pp. 625–646. IEEE Computer Society Press, Genoa, Italy (Jun 6–10, 2022). https://doi.org/10.1109/EuroSP53844.2022.00045
21. Cheon, J.H., Cho, W., Kim, J.H., Kim, J.: Adventures in crypto dark matter: Attacks and fixes for weak pseudorandom functions. In: Garay, J. (ed.) PKC 2021: 24th International Conference on Theory and Practice of Public Key Cryptography, Part II. Lecture Notes in Computer Science, vol. 12711, pp. 739–760. Springer, Cham, Switzerland, Virtual Event (May 10–13, 2021). https://doi.org/10.1007/978-3-030-75248-4_26
22. Cheon, J.H., Jang, D.: Cryptanalysis on lightweight verifiable homomorphic encryption. In: Advances in Cryptology – ASIACRYPT 2025, Part VII. pp. 366–397. Lecture Notes in Computer Science, Springer, Singapore, Singapore (Dec 7–11, 2025). https://doi.org/10.1007/978-981-95-5122-4_12
23. Cui, H., Guo, C., Guo, X., Wang, X., Yang, K., Yu, Y.: Simulation-based security notion of correlation robust hashing with applications to MPC. Cryptology ePrint Archive, Report 2025/1818 (2025), https://eprint.iacr.org/2025/1818
24. Davidson, A., Goldberg, I., Sullivan, N., Tankersley, G., Valsorda, F.: Privacy pass: Bypassing internet challenges anonymously. Proceedings on Privacy Enhancing Technologies **2018**(3), 164–180 (Jul 2018). https://doi.org/10.1515/popets-2018-0026
25. Dinur, I., Goldfeder, S., Halevi, T., Ishai, Y., Kelkar, M., Sharma, V., Zaverucha, G.: MPC-friendly symmetric cryptography from alternating moduli: Candidates, protocols, and applications. In: Malkin, T., Peikert, C. (eds.) Advances in Cryptology – CRYPTO 2021, Part IV. Lecture Notes in Computer Science, vol. 12828, pp. 517–547. Springer, Cham, Switzerland, Virtual Event (Aug 16–20, 2021). https://doi.org/10.1007/978-3-030-84259-8_18
26. Dittmer, S., Ishai, Y., Ostrovsky, R.: Line-point zero knowledge and its applications. In: Tessaro, S. (ed.) ITC 2021: 2nd Conference on Information-Theoretic Cryptography. Leibniz International Proceedings in Informatics (LIPIcs), vol. 199, pp. 5:1–5:24. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Seattle, WA, USA (Jul 23–26, 2021). https://doi.org/10.4230/LIPIcs.ITC.2021.5
27. Doerner, J., Haitner, I., Ishai, Y., Makriyannis, N.: From OT to OLE with subquadratic communication. In: ACM CCS 2025: 32nd Conference on Computer and Communications Security. pp. 2249–2263. ACM Press (Nov 2025). https://doi.org/10.1145/3719027.3765225
28. Escudero, D., Ghosh, S., Keller, M., Rachuri, R., Scholl, P.: Improved primitives for MPC over mixed arithmetic-binary circuits. In: Micciancio, D., Ristenpart, T. (eds.) Advances in Cryptology – CRYPTO 2020, Part II. Lecture Notes in Computer Science, vol. 12171, pp. 823–852. Springer, Cham, Switzerland, Santa Barbara, CA, USA (Aug 17–21, 2020). https://doi.org/10.1007/978-3-030-56880-1_29
29. Esgin, M.F., Steinfeld, R., Tairi, E., Xu, J.: LeOPaRd: Towards practical post-quantum oblivious PRFs via interactive lattice problems. Cryptology ePrint Archive, Report 2024/1615 (2024), https://eprint.iacr.org/2024/1615
30. Faller, S.H., Ottenhues, A., Ottenhues, J.: Composable oblivious pseudo-random functions via garbled circuits. In: Aly, A., Tibouchi, M. (eds.) Progress in Cryptology - LATINCRYPT 2023: 8th International

Conference on Cryptology and Information Security in Latin America. Lecture Notes in Computer Science, vol. 14168, pp. 249–270. Springer, Cham, Switzerland, Quito, Ecuador (Oct 3–6, 2023). https://doi.org/10.1007/978-3-031-44469-2_13
31. Furukawa, J., Lindell, Y., Nof, A., Weinstein, O.: High-throughput secure three-party computation for malicious adversaries and an honest majority. In: Coron, J.S., Nielsen, J.B. (eds.) Advances in Cryptology – EUROCRYPT 2017, Part II. Lecture Notes in Computer Science, vol. 10211, pp. 225–255. Springer, Cham, Switzerland, Paris, France (Apr 30 – May 4, 2017). https://doi.org/10.1007/978-3-319-56614-6_8
32. Galois, Inc.: swanky: A suite of rust libraries for secure computation. https://github.com/GaloisInc/swanky (2019)
33. Han, K., Kim, S., Lee, B., Son, Y.: Revisiting OKVS-based OPRF and PSI: Cryptanalysis and better construction. In: Chung, K.M., Sasaki, Y. (eds.) Advances in Cryptology – ASIACRYPT 2024, Part VIII. Lecture Notes in Computer Science, vol. 15491, pp. 266–296. Springer, Singapore, Singapore, Kolkata, India (Dec 9–13, 2024). https://doi.org/10.1007/978-981-96-0944-4_9
34. Hazay, C., Scholl, P., Soria-Vazquez, E.: Low cost constant round MPC combining BMR and oblivious transfer. In: Takagi, T., Peyrin, T. (eds.) Advances in Cryptology – ASIACRYPT 2017, Part I. Lecture Notes in Computer Science, vol. 10624, pp. 598–628. Springer, Cham, Switzerland, Hong Kong, China (Dec 3–7, 2017). https://doi.org/10.1007/978-3-319-70694-8_21
35. Heimberger, L.: The big table of post-quantum friendly OPRFs. https://heimberger.xyz/oprfs.html (2025)
36. Heimberger, L., Hennerbichler, T., Meisingseth, F., Ramacher, S., Rechberger, C.: OPRFs from isogenies: Designs and analysis. In: Zhou, J., Quek, T.Q.S., Gao, D., Cárdenas, A.A. (eds.) ASIACCS 24: 19th ACM Symposium on Information, Computer and Communications Security. ACM Press, Singapore (Jul 1–5, 2024). https://doi.org/10.1145/3634737.3645010
37. Heimberger, L., Kales, D., Lolato, R., Mir, O., Ramacher, S., Rechberger, C.: Leap: A fast, lattice-based OPRF with application to private set intersection. In: Fehr, S., Fouque, P.A. (eds.) Advances in Cryptology – EUROCRYPT 2025, Part VII. Lecture Notes in Computer Science, vol. 15607, pp. 254–283. Springer, Cham, Switzerland, Madrid, Spain (May 4–8, 2025). https://doi.org/10.1007/978-3-031-91098-2_10
38. Hu, K., Leander, G., Raddum, H., Sandrib, A., Udovenko, A.: Cryptanalysis of two alternating moduli weak PRFs. Cryptology ePrint Archive, Paper 2026/482 (2026), https://eprint.iacr.org/2026/482
39. Ishai, Y., Kelkar, M., Narayanan, V., Zafar, L.: One-message secure reductions: On the cost of converting correlations. In: Handschuh, H., Lysyanskaya, A. (eds.) Advances in Cryptology – CRYPTO 2023, Part I. Lecture Notes in Computer Science, vol. 14081, pp. 515–547. Springer, Cham, Switzerland, Santa Barbara, CA, USA (Aug 20–24, 2023). https://doi.org/10.1007/978-3-031-38557-5_17
40. Jarecki, S., Krawczyk, H., Xu, J.: OPAQUE: An asymmetric PAKE protocol secure against pre-computation attacks. In: Nielsen, J.B., Rijmen, V. (eds.) Advances in Cryptology – EUROCRYPT 2018, Part III. Lecture Notes in Computer Science, vol. 10822, pp. 456–486. Springer, Cham, Switzerland, Tel Aviv, Israel (Apr 29 – May 3, 2018). https://doi.org/10.1007/978-3-319-78372-7_15
41. Keller, M.: MP-SPDZ: A versatile framework for multi-party computation. In: Ligatti, J., Ou, X., Katz, J., Vigna, G. (eds.) ACM CCS 2020: 27th Conference on Computer and Communications Security. pp. 1575–1590. ACM Press, Virtual Event, USA (Nov 9–13, 2020). https://doi.org/10.1145/3372297.3417872
42. Keller, M., Orsini, E., Scholl, P.: Actively secure OT extension with optimal overhead. In: Gennaro, R., Robshaw, M.J.B. (eds.) Advances in Cryptology – CRYPTO 2015, Part I. Lecture Notes in Computer Science, vol. 9215, pp. 724–741. Springer Berlin Heidelberg, Germany, Santa Barbara, CA, USA (Aug 16–20, 2015). https://doi.org/10.1007/978-3-662-47989-6_35
43. Laur, S., Talviste, R., Willemson, J.: From oblivious AES to efficient and secure database join in the multiparty setting. In: Jacobson, Jr., M.J., Locasto, M.E., Mohassel, P., Safavi-Naini, R. (eds.) ACNS 2013: 11th International Conference on Applied Cryptography and Network Security. Lecture Notes in Computer Science, vol. 7954, pp. 84–101. Springer Berlin Heidelberg, Germany, Banff, AB, Canada (Jun 25–28, 2013). https://doi.org/10.1007/978-3-642-38980-1_6
44. Li, Z., Xing, C., Yao, Y., Yuan, C.: Efficient pseudorandom correlation generators for any finite field. In: Fehr, S., Fouque, P.A. (eds.) Advances in Cryptology – EUROCRYPT 2025, Part V. Lecture Notes

in Computer Science, vol. 15605, pp. 145–175. Springer, Cham, Switzerland, Madrid, Spain (May 4–8, 2025). https://doi.org/10.1007/978-3-031-91092-0_6
45. Masny, D., Rindal, P.: Endemic oblivious transfer. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) ACM CCS 2019: 26th Conference on Computer and Communications Security. pp. 309–326. ACM Press, London, UK (Nov 11–15, 2019). https://doi.org/10.1145/3319535.3354210
46. Mohassel, P., Rindal, P., Rosulek, M.: Fast database joins and PSI for secret shared data. In: Ligatti, J., Ou, X., Katz, J., Vigna, G. (eds.) ACM CCS 2020: 27th Conference on Computer and Communications Security. pp. 1271–1287. ACM Press, Virtual Event, USA (Nov 9–13, 2020). https://doi.org/10.1145/3372297.3423358
47. Naor, M., Reingold, O.: Number-theoretic constructions of efficient pseudo-random functions. J. ACM **51**(2), 231–262 (2004)
48. Nielsen, J.B., Nordholt, P.S., Orlandi, C., Burra, S.S.: A new approach to practical active-secure two-party computation. In: Safavi-Naini, R., Canetti, R. (eds.) Advances in Cryptology – CRYPTO 2012. Lecture Notes in Computer Science, vol. 7417, pp. 681–700. Springer Berlin Heidelberg, Germany, Santa Barbara, CA, USA (Aug 19–23, 2012). https://doi.org/10.1007/978-3-642-32009-5_40
49. Raghuraman, S., Rindal, P., Shah, H.: Two party secret shared joins. Cryptology ePrint Archive, Paper 2025/702 (2025), https://eprint.iacr.org/2025/702
50. Rotaru, D., Wood, T.: MArBled circuits: Mixing arithmetic and Boolean circuits with active security. In: Hao, F., Ruj, S., Sen Gupta, S. (eds.) Progress in Cryptology - INDOCRYPT 2019: 20th International Conference in Cryptology in India. Lecture Notes in Computer Science, vol. 11898, pp. 227–249. Springer, Cham, Switzerland, Hyderabad, India (Dec 15–18, 2019). https://doi.org/10.1007/978-3-030-35423-7_12
51. Roy, L.: SoftSpokenOT: Quieter OT extension from small-field silent VOLE in the minicrypt model. In: Dodis, Y., Shrimpton, T. (eds.) Advances in Cryptology – CRYPTO 2022, Part I. Lecture Notes in Computer Science, vol. 13507, pp. 657–687. Springer, Cham, Switzerland, Santa Barbara, CA, USA (Aug 15–18, 2022). https://doi.org/10.1007/978-3-031-15802-5_23
52. Seres, I.A., Horváth, M., Burcsi, P.: The legendre pseudorandom function as a multivariate quadratic cryptosystem: security and applications. Appl. Algebra Eng. Commun. Comput. **36**(2), 223–253 (2025)
53. Sidem, A., Wang, Q.: General key recovery attack on pointwise-keyed functions - application to alternating moduli weak PRFs. In: Advances in Cryptology – ASIACRYPT 2025, Part I. pp. 131–154. Lecture Notes in Computer Science, Springer, Singapore, Singapore (Dec 7–11, 2025). https://doi.org/10.1007/978-981-95-5018-0_5
54. van Baarsen, A., Stevens, M.: Amortizing circuit-PSI in the multiple sender/receiver setting. IACR Communications in Cryptology (CiC) **1**(3), 2 (2024). https://doi.org/10.62056/a0fhsgvtw
55. Weng, C., Yang, K., Katz, J., Wang, X.: Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. In: 2021 IEEE Symposium on Security and Privacy. pp. 1074–1091. IEEE Computer Society Press, San Francisco, CA, USA (May 24–27, 2021). https://doi.org/10.1109/SP40001.2021.00056
56. Weng, C., Yang, K., Xie, X., Katz, J., Wang, X.: Mystique: Efficient conversions for zero-knowledge proofs with applications to machine learning. In: Bailey, M., Greenstadt, R. (eds.) USENIX Security 2021: 30th USENIX Security Symposium. pp. 501–518. USENIX Association (Aug 11–13, 2021), https://www.usenix.org/conference/usenixsecurity21/presentation/weng
57. Yang, K., Sarkar, P., Weng, C., Wang, X.: QuickSilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. In: Vigna, G., Shi, E. (eds.) ACM CCS 2021: 28th Conference on Computer and Communications Security. pp. 2986–3001. ACM Press, Virtual Event, Republic of Korea (Nov 15–19, 2021). https://doi.org/10.1145/3460120.3484556
58. Yang, Y., Liang, X., Song, X., Dong, Y., Huang, L., Ren, H., Dong, C., Zhou, J.: Maliciously secure circuit private set intersection via spdz-compatible oblivious PRF. Proc. Priv. Enhancing Technol. **2025**(2), 680–696 (2025). https://doi.org/10.56553/POPETS-2025-0082
59. Yang, Y., Benhamouda, F., Halevi, S., Krawczyk, H., Rabin, T.: Gold OPRF: Post-quantum oblivious power-residue PRF. In: 2025 IEEE Symposium on Security and Privacy. pp. 259–278. IEEE Computer Society Press, San Francisco, CA, USA (May 2025). https://doi.org/10.1109/SP61157.2025.00116

A Additional Functionalities

Our protocols require standard additional functionalities that we present here. As these are standard and commonly known we provide no realization or proof of security.

A.1 Randomness

We define a randomness functionality, which outputs unbiased randomness to both parties. The functionality can be realized in the UC framework using random oracles as in, for example, [50, Appendix B].

Fig. 19 Ideal functionality $\mathcal{F}_{\mathsf{random}}$

Subset: Upon receiving $(\mathsf{sid}, \texttt{subset}, t, X)$ from all parties, where X is a set such that $|X| \geq t$. The functionality samples the set $S \xleftarrow{\$} \{Y \subseteq X : |Y| = t\}$. Return S to all parties.

Buckets: Upon receiving $(\mathsf{sid}, \texttt{buckets}, t, X)$ from all parties, where X is a set such that $|X| \mid t$. Let $n = |X|/t$, the functionality will then for $i \in [n]$ do:

- Sample $S_i \xleftarrow{\$} \{Y \subset X : |Y| = t\}$
- Set $X = X \setminus S_i$

Return $(S_i)_{i=1}^n$ to all parties.

A.2 Secure Equality Check

We define a weak equality-check functionality, which reveals $\mathcal{P}_i$'s input to $\mathcal{P}_{i\oplus 1}$, but aborts in case the inputs are not equal, in Figure 20. This functionality is adapted from [55] and can be realized in the UC framework using random oracles as in, for example, [55, Appendix A].

The equality-check functionality $\mathcal{F}_{\mathsf{EQ}}$ can easily be extended to check equality of long vectors with communication independent of the vector length. That is, the parties first hash their inputs $\boldsymbol{x}^0, \boldsymbol{x}^1 \in \mathbb{F}_q^\ell$ using a random oracle $\mathsf{H} : \{0,1\}^* \to \{0,1\}^\lambda$. Since the probability that $\mathsf{H}(\boldsymbol{x}^0) = \mathsf{H}(\boldsymbol{x}^1)$ for $\boldsymbol{x}^0 \neq \boldsymbol{x}^1$ is negligible, we can treat the resulting functionality in the same way as Figure 20, with the main difference that $\mathcal{P}_{i\oplus 1}$ learns $\mathsf{H}(\boldsymbol{x}^i)$ instead of $\boldsymbol{x}^i$. For ease of exposition we will simply denote this as the parties using $\boldsymbol{x}^0$ and $\boldsymbol{x}^1$ as their inputs to $\mathcal{F}_{\mathsf{EQ}}$.

Fig. 20 Equality-Check Functionality $\mathcal{F}_{\mathsf{EQ}}$ (Adapted from [55])

Parameters: Two parties $\mathcal{P}_0$ and $\mathcal{P}_1$, a domain $\mathcal{X}$ and $i \in \{0,1\}$ indicating which party's input leaks to the other party.

Check: Upon receiving $(\mathsf{sid}, \texttt{check}, x_0, i)$ from $\mathcal{P}_0$ and $(\mathsf{sid}, \texttt{check}, x_1, i)$ from $\mathcal{P}_1$ with $x_0, x_1 \in \mathcal{X}$ and $i \in \{0,1\}$:

- Send $(x_0 \stackrel{?}{=} x_1)$ and x_i to $\mathcal{P}_{i\oplus 1}$.
- If $\mathcal{P}_{i\oplus 1}$ is honest and $x_0 = x_1$, or is corrupted and continues, send $(x_0 \stackrel{?}{=} x_1)$ to $\mathcal{P}_i$.
- If $\mathcal{P}_{i\oplus 1}$ is honest and $x_0 \neq x_1$, or is corrupted and aborts, send $\texttt{abort}$ to $\mathcal{P}_i$.

B Authenticated OLE

We present the fully-detailed protocol $\Pi_{\mathsf{aOLE}}^{3,r^{(3)}}$ in Figure 21, which realizes $\mathcal{F}_{\mathsf{mRand}}^{3,r^{(2)},r^{(3)}}(\textcircled{3})$ in the $\mathcal{F}_{\mathsf{sVOLE}}^{3,r^{(3)}}$-, $\mathcal{F}_{\mathsf{Random}}$-hybrid model.

Proposition 2. *Given a dishonest party $\mathcal{P}_i$ the protocol $\Pi_{\mathsf{aOLE}}^{3,r^{(3)}}$ with output size ℓ and bucketing parameters (B, C) has at most $\ell \cdot \binom{\ell B^2+C}{B}^{-1}$ probability of accepting an invalid authenticated OLE.*

Proof. $\mathcal{P}_0^*$ can make OLE i invalid by letting $c_{0_i}^* \neq c_{0_i}$. $\mathcal{P}_1^*$ can make OLE i invalid by letting $b_i^* \neq b_i, c_{1_i}^* \neq c_{1_i}$ or $\delta_i^* \neq \delta_i$. A corrupt δ_i is equivalent to corrupting c_{1_i} with $a^2(\delta_i^* - \delta_i)$. Define $\delta_i^* = \delta_i + x$ for some $x \in \{1, 2\}$

$$\begin{aligned} c_0 + c_1 &= m_a + \delta^* \cdot a^2 - m_0 \\ &= m_a + (\delta_i + x) \cdot a^2 - m_0 \\ &= c_0 + (c_1 + a^2 \cdot x) \end{aligned}$$

This corruption term is unknown to $\mathcal{P}_1^*$, thus the possible attacks by corrupting δ_i are a subset of corrupting c_{1_i}.

Let k be the number of corrupt OLEs. k must be a multiple of B otherwise $\Pr[\mathcal{A} \text{ wins}] = 0$. This can be seen as, during bucketing an invalid OLE is not detected if and only if it is within a bucket only containing invalid OLEs. Let (a, b, c_0, c_1) be a valid OLE in the same bucket as an invalid OLE (x, yz_0, z_1). By case distinction on the 3 point to corrupt we show that this invalid OLE is detected.

- Corrupt y': Define $y' = y + v$ where $v \in \{1, 2\}$ is the corrupting term.

  $$f_0 + f_1 = (c_0 + c_1) - (z_0 + z_1) + vx - ab + xy = vx$$

  The openings show the value $vx \neq 0$ for $x \in \{1, 2\}$ which is therefore an invalid OLE. Note that if $x = 0$ any value of y' creates a valid OLE.
- Corrupt z_0': Define $z_0' = z_0 + v$ where $v \in \{1, 2\}$.

  $$f_0 + f_1 = (c_0 + c_1) - (z_0 + z_1) + v - ab + xy = v$$

  This reveals the invalid OLE.
- Corrupt z_1': Define $z_1' = z_1 + v$ where $v \in \{1, 2\}$.

  $$f_0 + f_1 = (c_0 + c_1) - (z_0 + z_1) - v - ab + xy = -v$$

  This reveals the invalid OLE.

Thus, if an invalid OLE is in the same bucket as an honest OLE, the invalid OLE will always be detected.

From the above argument, let $k = tB$ for $t \in [1..B\ell]$ be the number of corrupt OLEs. An invalid OLE can be detected by being sacrificed or opened during bucketing. An OLE sacrificed that is invalid is detected by definition. Let E_1 be the event that no corrupt OLEs are opened

$$\Pr[E_1] = \frac{\binom{B^2\ell+C-tB}{C}}{\binom{B^2\ell+C}{C}}$$

Fig. 21 Authenticated OLE Protocol $\Pi_{\mathsf{aOLE}}^{3,r^{(3)}}$

Parameters: Two parties $\mathcal{P}_0$ and $\mathcal{P}_1$, number of authenticated OLEs ℓ, Cut-and-Choose parameters B, C such that $M = B^2\ell + C$ and extension field parameter $r^{(3)}$ for $\mathbb{F}_3$. $\mathsf{H} : \mathbb{F}_{3^{r^{(3)}}} \to \mathbb{F}_3$ is a random oracle.

Initializations: The parties send $(\mathsf{sid}, \mathtt{init}, 0)$ and $(\mathsf{sid}, \mathtt{init}, 1)$ to $\mathcal{F}_{\mathsf{sVOLE}}^{3,r^{(3)}}$. $\mathcal{P}_0$ receives $\Delta_0^{(3)} \in \mathbb{F}_{3^{r(3)}}$ and $\mathcal{P}_1$ receives $\Delta_1^{(3)} \in \mathbb{F}_{3^{r(3)}}$.

Generate Candidate Triples

- $\mathcal{P}_0$ samples $\boldsymbol{a} \xleftarrow{\$} \mathbb{F}_3^M$.
- Send $\mathsf{Commit}(\boldsymbol{a}, 3, r^{(3)}, \Delta_1)$ to $\mathcal{F}_{\mathsf{sVOLE}}^{3,r^{(3)}}$ and obtain $[\![\boldsymbol{a}]\!]_0^{(3)} = ((\boldsymbol{a}, \boldsymbol{M}), (\Delta_1, \boldsymbol{K}))$.
- $\mathcal{P}_0$ computes $m_{a_i,i} = \mathsf{H}(M_i, i)$ for each $j \in [M]$.
- $\mathcal{P}_1$ computes $m_{j,i} = \mathsf{H}(\Delta_1 \cdot j + K_i, i)$ for each $j \in \{0,1,2\}$ and $i \in [M]$.
- $\mathcal{P}_1$ sets $\delta_i = m_{0,i} + m_{1,i} + m_{2,i}$ for each $i \in [\ell]$ and sends $\boldsymbol{\delta}$ to $\mathcal{P}_0$.
- $\mathcal{P}_0$ lets $\boldsymbol{c}_0 = \boldsymbol{m}_{\boldsymbol{a}} + \boldsymbol{\delta} \odot \boldsymbol{a}^2$. $\mathcal{P}_1$ lets $\boldsymbol{b} = 2 \cdot \boldsymbol{m}_1 + \boldsymbol{m}_2$ and $\boldsymbol{c}_1 = -\boldsymbol{m}_0$.
- Send $\mathsf{Commit}(\boldsymbol{b}, 3, r^{(3)}, \Delta_0^{(3)})$, $\mathsf{Commit}(\boldsymbol{c}_0, 3, r^{(3)}, \Delta_1^{(3)})$ and $\mathsf{Commit}(\boldsymbol{c}_1, 3, r^{(3)}, \Delta_0^{(3)})$ to $\mathcal{F}_{\mathsf{sVOLE}}^{3,r^{(3)}}$ to obtain $[\![\boldsymbol{b}]\!]_1^{(3)}, [\![\boldsymbol{c}_0]\!]_0^{(3)}, [\![\boldsymbol{c}_1]\!]_1^{(3)}$.
- Note that $[\![\boldsymbol{c}_0]\!]_0^{(3)}$ and $[\![\boldsymbol{c}_1]\!]_1^{(3)}$ define a doubly authenticated sharing of $\boldsymbol{c}$ where $\boldsymbol{c} := \boldsymbol{c}_0 + \boldsymbol{c}_1 \bmod 3$, which we denote as $\langle\!\langle \boldsymbol{c} \rangle\!\rangle^{(3)}$.

Sacrificing Openings:

- Send $(\mathsf{sid}, \mathtt{subset}, C, M)$ to $\mathcal{F}_{\mathsf{random}}$ to obtain a subset $S \subset [M]$ of size C.
- Open $([\![a_i]\!]_0^{(3)}, [\![b_i]\!]_1^{(3)}, \langle\!\langle c_i \rangle\!\rangle^{(3)})_{i \in S}$ using $\mathsf{BatchOpen}$ and verify that $c_i = a_i \cdot b_i \bmod 3$. If the verification fails for any $i \in S$ the parties abort.

Checking Correctness:

- Send $(\mathsf{sid}, \mathtt{buckets}, B, B^2\ell)$ to $\mathcal{F}_{\mathsf{random}}$ to obtain sets $S_i \subset [M] \setminus S$ of size B for $i \in [B\ell]$. The remaining $B^2\ell$ elements of $[\![\boldsymbol{a}]\!]_0^{(3)}, [\![\boldsymbol{b}]\!]_1^{(3)}, \langle\!\langle \boldsymbol{c} \rangle\!\rangle^{(3)}$ are put into buckets according to these sets.
- For each bucket $i \in [B\ell]$: let the elements in S_i be relabelled from 1 to B
  - $\mathcal{P}_0$ opens $\boldsymbol{d}$ using $\mathsf{BatchOpen}\left([\![a_2 - a_1]\!]_0^{(3)}, \ldots, [\![a_B - a_1]\!]_0^{(3)}\right)$.
  - $\mathcal{P}_1$ opens $\boldsymbol{e}$ using $\mathsf{BatchOpen}\left([\![b_2 - b_1]\!]_1^{(3)}, \ldots, [\![b_B - b_1]\!]_1^{(3)}\right)$.
  - For $j = 2, \ldots, B$, let $f_j := \langle\!\langle c_1 \rangle\!\rangle^{(3)} - \langle\!\langle c_j \rangle\!\rangle^{(3)} + d_j \cdot \langle\!\langle b_1 \rangle\!\rangle^{(3)} + e_j \cdot \langle\!\langle a_1 \rangle\!\rangle^{(3)} + d_j \cdot e_j$.
  - The parties open $\boldsymbol{f}$ using $\mathsf{BatchOpen}\left((\langle\!\langle f_2 \rangle\!\rangle^{(3)}, \ldots, \langle\!\langle f_B \rangle\!\rangle^{(3)}), \{0,1\}\right)$, and check that $\boldsymbol{f} = 0$, otherwise abort.
  - The parties relabel $([\![a_i]\!]_0^{(3)}, [\![b_i]\!]_1^{(3)}, \langle\!\langle c_i \rangle\!\rangle^{(3)}) \leftarrow ([\![a_1]\!]_0^{(3)}, [\![b_1]\!]_1^{(3)}, \langle\!\langle c_1 \rangle\!\rangle^{(3)})$

Removing Leakage:

- Send $(\mathsf{sid}, \mathtt{buckets}, B, B\ell)$ to $\mathcal{F}_{\mathsf{random}}$ to obtain sets $T_i \subset [B\ell]$ of size B for $i \in [\ell]$. The remaining $B\ell$ elements of $[\![\boldsymbol{a}]\!]_0^{(3)}, [\![\boldsymbol{b}]\!]_1^{(3)}, \langle\!\langle \boldsymbol{c} \rangle\!\rangle^{(3)}$ are bucketed according to T_i.
- For each bucket $i \in [\ell]$: let the elements in T_i be relabeled from 1 to B.
  - $\mathcal{P}_1$ opens $\boldsymbol{d}$ using $\mathsf{BatchOpen}\left([\![b_2 - b_1]\!]_1^{(3)}, \ldots, [\![b_B - b_1]\!]_1^{(3)}\right)$.
  - For $j = 2, \ldots, B$:
    - Let $\langle\!\langle c'_j \rangle\!\rangle^{(3)} = d_j \cdot \langle\!\langle a_j \rangle\!\rangle^{(3)} + \langle\!\langle c_1 \rangle\!\rangle^{(3)} - \langle\!\langle c_j \rangle\!\rangle^{(3)}$, $[\![a'_j]\!]_0^{(3)} = [\![a_1]\!]_0^{(3)} + [\![a_j]\!]_0^{(3)}$.
    - Replace $([\![a_1]\!]_0^{(3)}, [\![b_1]\!]_1^{(3)}, \langle\!\langle c_1 \rangle\!\rangle^{(3)}) \leftarrow ([\![a'_j]\!]_0^{(3)}, [\![b_1]\!]_1^{(3)}, \langle\!\langle c'_j \rangle\!\rangle^{(3)})$.
  - Relabel $([\![a_i]\!]_0^{(3)}, [\![b_i]\!]_1^{(3)}, \langle\!\langle c_i \rangle\!\rangle^{(3)}) \leftarrow ([\![a_1]\!]_0^{(3)}, [\![b_1]\!]_1^{(3)}, \langle\!\langle c_1 \rangle\!\rangle^{(3)})$.
- The parties output $([\![a_i]\!]_0^{(3)}, [\![b_i]\!]_1^{(3)}, \langle\!\langle c_i \rangle\!\rangle^{(3)})_{i \in [\ell]}$

This can be seen because there exist $\binom{B^2\ell+C-tB}{C}$ different ways of selecting C non-corrupt OLEs out of $\binom{B^2\ell+C}{C}$ ways of selecting C OLEs.

Let E_2 be the event that all corrupt OLEs fall into buckets only with corrupt OLEs, which by the argument from [31, Theorem 5.2] gives

$$\Pr[E_2] = \frac{\binom{\ell}{t}(tB)!(\ell B^2 - tB)!}{(\ell B^2)!}$$

By the independence of E_1 and E_2 we get

$$\Pr[\mathcal{A} \text{ wins}] = \Pr[E_1] \cdot \Pr[E_2] = \binom{\ell}{t}\binom{\ell B^2 + C}{tB}^{-1}$$

This probability is maximized when $n = 1$:

$$\Pr[\mathcal{A} \text{ wins}] = \ell \cdot \binom{\ell B^2 + C}{B}^{-1}$$

□

Proposition 3. *Given a malicious $\mathcal{P}_1$ the protocol $\Pi^{3,r^{(3)}}_{\mathsf{aOLE}}$ has at most a $3^{-B} \cdot \frac{\ell(B)!(\ell B^2 - B)!}{(\ell B^2)!}$ probability of revealing a_i to $\mathcal{P}_1$ for some i.*

Proof. Let k be the number of corrupt OLEs. For $\mathcal{P}_1^*$ to leak the value $a_i = 0$ for some i the i'th OLE must be corrupted by setting $b_i^* \neq b_i$. If dishonest b_i^* is not detected during the check of correctness step, $a_i = 0$ is revealed to $\mathcal{P}_1^*$, since all values for b_i are a valid OLE. However, if $a_i \neq 0$ the check of correctness step fails with high probability. Hence the probability $\mathcal{P}_1^*$ to successfully corrupt k OLEs with the intention to leak is 3^{-k}.

Assume that all corrupt b_i^* are corruptions on OLEs with $a_i = 0$. During the check of correctness step, $B - 1$ OLEs in each bucket are revealed. If a bucket contains 1 corrupt OLE, $\mathcal{P}_1^*$ learns the value of a_1 in that bucket since $a_j - a_1 = -a_1$ when a_j is known to be 0. Hence $\mathcal{P}_1^*$ will not corrupt more than $B\ell$ OLEs. The best scenario for $\mathcal{P}_1^*$ is that each corrupt OLE falls in a different bucket. If two or more corrupt OLEs land in the same bucket, only 1 surviving value is learned.

The bucketing in the remove leakage step ensures the probability of leakage is negligible. Let a be the combination of all a_i for $i \in T_j$. For $\mathcal{P}_1^*$ to retain the knowledge of a, it requires that $\mathcal{P}_1^*$ knows all a_i. Thus the number of corrupt OLE must be at least B, hence let $k = t + B$ for $t \in [1, \ldots, B(\ell - 1)]$.

Since all a_i in a bucket must be leaked to leak a the bucket must be full of corrupt OLEs. However, if there are more than B corrupt OLEs, only B of them need to fall into the same bucket to leak a. The remaining corrupts OLEs can fall in any other bucket. The probability that $\mathcal{P}_1^*$ successfully leaks a value a is the probability that there exists a bucket containing only corrupt OLEs. By the inclusion-exclusion principle:

$$\Pr[\exists \text{ Corrupt bucket}] = \sum_{j=1}^{\ell}(-1)^{j+1}\binom{\ell}{j}\frac{\binom{t+B}{jB}}{\binom{B\ell}{jB}},$$

where $\binom{\ell}{j}$ are a selection of j buckets to fill and $\frac{\binom{t+B}{jB}}{\binom{B\ell}{jB}}$ is the probability that all these j chosen buckets are full of corrupt OLEs. By independence of the two events we have

$$\Pr[\mathcal{A} \text{ wins}] = 3^{-(t+B)} \cdot \sum_{j=1}^{\ell}(-1)^{j+1}\binom{\ell}{j}\frac{\binom{t+B}{jB}}{\binom{B\ell}{jB}}$$

This probability is maximized when $t = 0$, thus this is equivalent to all corrupt OLEs landing in the same bucket. The is described as E_2 in Proposition 2, thus

$$\Pr[\mathcal{A} \text{ wins}] = 3^{-B} \cdot \frac{\ell(B)!(\ell B^2 - B)!}{(\ell B^2)!}$$

□

Theorem 11. *The protocol $\Pi^{3,r^{(3)}}_{\mathsf{aOLE}}$ securely realize the ideal functionality $\mathcal{F}^{3,r^{(2)},r^{(3)}}_{\mathsf{mRand}}(③)$ against malicious adversaries in the $(\mathcal{F}^{3,r^{(3)}}_{\mathsf{sVOLE}}, \mathcal{F}_{\mathsf{Random}})$-hybrid model and H as a random oracle. $\Pi^{3,r^{(3)}}_{\mathsf{aOLE}}$ have a failure probability $\ell \cdot \binom{\ell B^2 + C}{B}^{-1}$ for bucketing parameters (B, C).*

Proof. If both parties behave honestly, it is straightforward to show that the parties obtain correct authenticated OLE triples. First, we show that the values $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c_0}, \boldsymbol{c_1}$ are computed correctly by looking at one of the entries

$$a \cdot b = c^0 + c^1$$
$$a \cdot (2 \cdot m_1 + m_2) = m_a + (m_0 + m_1 + m_2) \cdot a^2 - m_0$$

Now for $a \in \{0, 1, 2\}$, it is easy to see that the equality holds as $m_a = \mathsf{H}(M) = \mathsf{H}(\Delta^{(3)}_1 \cdot a + K)$. Each value of the authenticated OLE is correctly authenticated by definition of $\mathcal{F}^{3,r^{(3)}}_{\mathsf{sVOLE}}$. As all authenticated OLE are valid the sacrificial openings opens correctly. The checking correctness step also succeeds, as each party computes

$$\begin{aligned}
\mathcal{P}_0 : [\![f^0_j]\!] &= [\![c^0_1]\!] - [\![c^0_j]\!] + (b_j - b_1) \cdot [\![a_1]\!] + (b_j - b_1) \cdot (a_j - a_1) \\
\mathcal{P}_1 : [\![f^1_j]\!] &= [\![c^1_1]\!] - [\![c^1_j]\!] + (a_j - a_1) \cdot [\![b_1]\!] \\
[\![f^0_j]\!] + [\![f^1_j]\!] &= [\![c^0_1]\!] + [\![c^1_1]\!] - ([\![c^0_j]\!] + [\![c^1_j]\!]) - a_1 b_1 + a_j b_j \\
&= [\![a_1]\!][\![b_1]\!] - [\![a_1]\!][\![b_1]\!] + [\![a_j]\!][\![b_j]\!] - [\![a_j]\!][\![b_j]\!] = 0
\end{aligned}$$

This shows that the opening is 0 when the triples are valid. As the commitments are linear homomorphic the all openings will be accepted by verification. Lastly, the leakage removal produces valid triples as the replaced triple is consistent

$$\begin{aligned}
[\![a'_j]\!] \cdot [\![b_1]\!] &= [\![c'^0_j]\!] + [\![c'^1_j]\!] \\
([\![a_1]\!] + [\![a_j]\!]) \cdot [\![b_1]\!] &= d_j \cdot [\![a_j]\!] + ([\![c^0_1]\!] + [\![c^1_1]\!]) + ([\![c^0_j]\!] + [\![c^1_j]\!]) \\
[\![a_1]\!] \cdot [\![b_1]\!] + [\![a_j]\!] \cdot [\![b_1]\!] &= [\![a_1]\!] \cdot [\![b_1]\!] + [\![a_j]\!] \cdot [\![b_1]\!]
\end{aligned}$$

The parties output correctly authenticated OLEs.

Let $\mathcal{P}_0$ be maliciously corrupted by adversary $\mathcal{A}$. We define the simulator $\mathcal{S}$ as:

- Receives $\Delta^{(3)}_0 \in \mathbb{F}_{3^{r(3)}}$ from $\mathcal{A}$ when forwarding the call $(\mathsf{sid}, \texttt{init}, 0)$ to $\mathcal{F}^{3,r(3)}_{\mathsf{sVOLE}}$.
- Receives $\boldsymbol{a}, \boldsymbol{c}_0$ and $\boldsymbol{M_a}, \boldsymbol{M_{c_0}}$ from $\mathcal{A}$ when forwarding the calls $\mathsf{Commit}(\boldsymbol{a}, r, r^{(3)}, \Delta_1)$ and $\mathsf{Commit}(\boldsymbol{c}_0, r, r^{(3)}, \Delta_1)$ to $\mathcal{F}^{3,r(3)}_{\mathsf{sVOLE}}$.
- Receives $\boldsymbol{K_b}, \boldsymbol{K_{c_1}}$ from $\mathcal{A}$ when forwarding the calls $\mathsf{Commit}(\boldsymbol{b}, r, r^{(3)}, \Delta_0)$ and $\mathsf{Commit}(\boldsymbol{c}_1, r, r^{(3)}, \Delta_0)$ to $\mathcal{F}^{3,r(3)}_{\mathsf{sVOLE}}$.
- Samples $\boldsymbol{\delta} \xleftarrow{\$} \mathbb{F}_3^M$ and gives $\boldsymbol{\delta}$ to $\mathcal{A}$.
- $\mathcal{S}$ defines the set $S_{\Delta_1} = \emptyset$. If at some point $|S_{\Delta_1}| > 1$ $\mathcal{S}$ aborts.

– To simulate the sacrifice step, $\mathcal{S}$ does the following
  - Receives the set S from $\mathcal{A}$ when forwarding the call $(\mathsf{sid}, \mathsf{subset}, C, M)$ to $\mathcal{F}_{\mathsf{random}}$.
  - For $i \in S$: Receives opening (a_i, M_{a_i}) and $(c_{0_i}, M_{c_{0_i}})$ from $\mathcal{A}$.
  - If any of the openings deviate from protocol specification, let this deviation be (x', M') where (x, M) is the honest openings, the simulator checks. (1) If $x = x'$: $\mathcal{S}$ abort. (2) If $x \neq x'$: compute $\Delta_1' = (M - M')(x - x')$ and add Δ_1' to S_{Δ_1}
  - For $i \in S$: Samples $b_i \xleftarrow{\$} \mathbb{F}_3$ and set $M_{b_i} = K_{b_i} + b \cdot \Delta_1$. Let

$$c_{1_i} = \begin{cases} -H(M_{a_i}) & \text{if } a_i = 0 \\ b_i - \delta_i - H(M_{a_i}) & \text{if } a_i = 1 \\ 2H(M_{a_i}) - \delta_i - b_i & \text{if } a_i = 2 \end{cases} \tag{7}$$

    and set $M_{c_{1_i}} = K_{c_{1_i}} + c_{1_i} \cdot \Delta_0$.
  - If $c_{0_i} + c_{1_i} \neq 0$ abort.
– To simulate the checking correctness step, $\mathcal{S}$ does the following
  - Receives the sets S_i from $\mathcal{A}$ when forwarding the call $(\mathsf{sid}, \mathsf{buckets}, B, B^2\ell)$ to $\mathcal{F}_{\mathsf{random}}$.
  - For $i \in [B\ell]$, for $j \in S_i$: $\mathcal{S}$ receives the openings d_j, f_j and the corresponding MACs M_{d_j}, M_{f_j}.
  - If any of the openings deviate from protocol specification, let this deviation be (x', M') where (x, M) is the honest openings, the simulator checks. (1) If $x = x'$ abort. (2) If $x \neq x'$ compute $\Delta_1' = (M - M')(x - x')$ and add Δ_1' to S_{Δ_1}.
  - For $i \in [B\ell]$, for $j \in S_i$: $\mathcal{S}$ sets $(b_j, M_{b_j}) = (b_j \xleftarrow{\$} \mathbb{F}_3, M_{b_j} = K_{b_j} + b_j \cdot \Delta_0)$. Let c_{1_j} be defined by Equation 7 and let $M_{c_{1_j}} = K_{c_{1_j}} + c_{1_j} \cdot \Delta_0$.
  - Opens e_j and f_j as per protocol specification to $\mathcal{A}$. If any openings of $f_j \neq 0$ abort.
– To simulate the removing leakage step, $\mathcal{S}$ does the following
  - Receives the sets T_i from $\mathcal{A}$ when forwarding the call $(\mathsf{sid}, \mathsf{buckets}, B, B\ell)$ to $\mathcal{F}_{\mathsf{random}}$.
  - For $i \in [\ell]$, for $j \in T_i$: $\mathcal{S}$ sets $(b_j, M_{b_j}) = (b_j \xleftarrow{\$} \mathbb{F}_3, M_{b_j} = K_{b_j} + b_j \cdot \Delta_0)$, and opens d_j to $\mathcal{A}$.
– If $|S_{\Delta_1}| = 1$, send $(\mathsf{sid}, \mathsf{guess}, S_{\Delta_1}, 3)$ to $\mathcal{F}_{\mathsf{mRand}}^{3,r^{(2)},r^{(3)}}$ (③). If the response is $(\mathtt{success}, \Delta_1)$ send $\boldsymbol{a}^0 = [a_i], \boldsymbol{w}_{\boldsymbol{a}^0} = [M_{a_i}], \boldsymbol{c}^0 = [\mathsf{H}(a_i) + \delta_i \cdot a_i^2], \boldsymbol{w}_{\boldsymbol{c}^0} = [\Delta_1 \cdot (c_{0_i} - \mathsf{H}(a_i) - \delta_i \cdot a_i^2) + M_{c_{0_i}}]$ for $i \in [\ell]$ to $\mathcal{F}_{\mathsf{mRand}}^{3,r^{(2)},r^{(3)}}$ (③).
– Send the remaining $\boldsymbol{a}^0 = [a_i], \boldsymbol{w}_{\boldsymbol{a}^0} = [M_{a_i}], \boldsymbol{c}^0 = [c_{0_i}], \boldsymbol{w}_{\boldsymbol{c}^0} = [M_{c_{0_i}}]$ for $i \in [\ell]$ to $\mathcal{F}_{\mathsf{mRand}}^{3,r^{(2)},r^{(3)}}$ (③).

We now argue for the indistinguishability between the real and ideal worlds. Two types of messages are received from $\mathcal{S}$, the first type is the $\boldsymbol{\delta}$ message, and the second type is the openings performed during the protocol steps. The message $\boldsymbol{\delta}$ is distributed uniformly in the real world, as the output of H is uniformly random according to the definition. The openings of $\mathcal{S}$ contain two values b and c_1. The distribution of b is uniform random in both worlds, as $\mathcal{A}$ cannot know both m_1 and m_2, so the simulation is identical. The distribution of c_1 cannot just be sampled since the equation $a \cdot b = c_0 + c_1$ must be satisfied. Thus, by the definition of Equation 7 the value of c_1 is set according to the protocol specification. This ensures that openings that are not valid triples will be detected in the ideal world, as they will be in the real world.

If any MACs from $\mathcal{A}$ are not according to the protocol specification, the real world always aborts except if $\mathcal{A}$ guesses Δ_1. This matches with the ideal world distribution since $\mathcal{S}$ constructs the set S_{Δ_1} of $\mathcal{A}$ guesses and sends this to $\mathcal{F}_{\mathsf{mRand}}^{3,r^{(2)},r^{(3)}}$ (③). If the guess from $\mathcal{A}$ is incorrect, both worlds abort. Otherwise, the real world $\mathcal{P}_1$ will output $([\![\boldsymbol{a}]\!]_0^{(3)}, [\![\boldsymbol{b}]\!]_1^{(3)}, [\![\boldsymbol{c}_0]\!]_0^{(3)})$, which is $(\boldsymbol{K}_{\boldsymbol{a}}, (\boldsymbol{b}, \boldsymbol{M}_{\boldsymbol{b}}), \boldsymbol{K}_{\boldsymbol{c}_0})$. To ensure that the simulator outputs consistent values with the real world in the case a $\mathcal{A}$ guessing Δ_1 the simulator alters the input to $\mathcal{F}_{\mathsf{mRand}}^{3,r^{(2)},r^{(3)}}$ (③). In the real world $\mathcal{A}$ cannot interfere with the

output values of the honest party after the commitment of $\boldsymbol{a}$, thus generating the corresponding inputs to $\mathcal{F}_{\mathsf{mRand}}^{3,r^{(2)},r^{(3)}}$ (③) based on the value of $\boldsymbol{a}$, will make the ideal world produce the same output as the real world.

$\mathcal{A}$ can try to distinguish the real world from the ideal world by committing to inconsistent parts of the authenticated OLE. In the real world these inconsistent parts will result in an invalid authenticated OLE, however, the ideal world will not produce invalid authenticated OLEs. $\mathcal{A}$ has at most probability $\ell \cdot \binom{\ell B^2+C}{B}^{-1}$ per Proposition 2 to make such an inconsistency not be detected.

After the execution of the protocol $\mathcal{Z}$ learn the output of all parties, and hence learn Δ_1. Since the input of $\mathcal{A}$,$\boldsymbol{a}$, determines the values produced in the real world, $\mathcal{Z}$ can compute all expected openings from the honest party using $\mathsf{H}(\Delta_1 \cdot j + \boldsymbol{K})$ for $j \in \{0,1,2\}$. Since δ_i, b_i are sampled in the ideal world and set based on H in the real world, $\mathcal{Z}$ can query H to learn the expected values. We handle this as follows. Each time $\mathcal{Z}$ queries H on with input (j_i, Q_i): go over all previous queries, (j_k, Q_K) for $k < i$ and let $\Delta_1' = Q_i - Q_k, \Delta_1'' = Q_k - Q_i$. Then query the global key of $\mathcal{F}_{\mathsf{mRand}}^{3,r^{(2)},r^{(3)}}$ (③) to determine if $\Delta_1 = \Delta_1'$ and $\Delta_1 = \Delta_1''$. If the functionality returns $(\texttt{success}, (\boldsymbol{K_a}, \boldsymbol{b}, \boldsymbol{M_b}, \boldsymbol{c}_1, \boldsymbol{M_{c_1}}))$, $\mathcal{S}$ also learns the honest party's output. Using the knowledge of Δ_1 and $\boldsymbol{a}$, $\mathcal{S}$ derives the value of $\boldsymbol{K} = \boldsymbol{M} - \Delta_1 \cdot \boldsymbol{a}$.

For a given index j where b_j, c_{1_j} or δ_j will be verified by $\mathcal{Z}$, the simulator will program H to give consistent outputs. We show how to program H by case distinction on a_j

- If $a_j = 1$: $\mathcal{S}$ knows one query to H, namely $M_j = \Delta_1 \cdot 1 + K_j$, then at the point of the second query, $\mathcal{S}$ uses the above strategy to learn Δ_1 and the honest party's outputs. Since $m_1 = \mathsf{H}(M_j)$ is decided before $\mathcal{S}$ learns this knowledge, $\mathcal{S}$ will program $\mathsf{H}(K_j) = m_0 = -c_{1_j}$ and $\mathsf{H}(\Delta_1 \cdot 2 + K_j) = m_2 = b_j - 2m_1$. This ensures the consistency of the honest party's output c_{1_j} and b_j. From these programmings, the expected value of δ_j is

$$\begin{aligned}
\delta_j &= m_0 + m_1 + m_2 \\
&= -c_{1_j} + m_1 + b_j - 2m_1 \\
&= b_j - m_1 - c_{1_j} \\
&= b_j - m_1 - (a_j \cdot b_j - c_{0_j}) \\
&= b_j - m_1 - b_j + c_{0_j} \\
&= c_{0_j} - m_1
\end{aligned}$$

  Since c_{0_j} is honest with overwhelming probability as discussed above, we have $c_{0_j} = m_1 + \delta_j$, thus the consistency of δ_j is ensured as

$$\delta_j = c_{0_j} - m_1 = m_1 + \delta_j - m_1$$

- If $a_j = 2$: $\mathcal{S}$ knows the query $\mathsf{H}(M_j) = \mathsf{H}(\Delta_1 \cdot 2 + K_j) = m_2$. $\mathcal{S}$ learns Δ_1 and the honest party's output when a second query to H is made, using the above strategy. Since m_2 is decided before this knowledge $\mathcal{S}$ must program $\mathsf{H}(\Delta_1 \cdot 1 + K_j) = m_1 = 2b_j - 2m_2$ and $\mathsf{H}(K_j) = m_0 = -c_{0_j}$ to ensure consistency of the honest party's output. The expected value of δ_j will be

$$\begin{aligned}
\delta_j &= m_0 + m_1 + m_2 \\
&= c_{0_j} - m_2
\end{aligned}$$

  Using the same argument as in the above case, we have $c_{0_j} = m_2 + \delta_j$, which ensures the consistency of δ_j

– If $a_j = 0$: $\mathcal{S}$ knows the query $\mathsf{H}(M_j) = \mathsf{H}(K_j) = m_0$. As in the other cases, $\mathcal{S}$ learns Δ_1 and the honest parties output when a second query is made. Here m_0 cannot be programmed to $-c_{1_j}$, however, due to the assumption $-c_{1_j} = c_{0_j}$, and $c_{0_j} = m_0$ if $\mathcal{A}$ created an honest authenticated OLE. As shown above, this happens except with negligible probability. Since both m_1 and m_2 are undecided at this point, $\mathcal{S}$ will program these according to the order of their query. Let m_1 be queries first, setting $m_1 = -\delta_j + b_j + c_{0_j}$ and $m_2 = b_j - 2m_1$, ensure consistency of b_j and δ_j will look as follows

$$\begin{aligned}\delta &= m_0 + m_1 + m_2 \\ &= -c_{1_j} + m_1 + b_j - 2m_1 \\ &= c_{0_j} + b_j - m_1 \\ &= c_{0_j} + b_j - (-\delta_j + b_j + c_{0_j}) \\ &= \delta_j\end{aligned}$$

If m_2 is first queried, setting $m_1 = 2b_j - 2m_2$ and $m_2 = -\delta + 2b_j + c_{0_j}$ as a similar derivation will show that δ_j is consistent.

Let $\mathcal{P}_1$ be maliciously corrupted by adversary $\mathcal{A}$. We define the simulator $\mathcal{S}$ as:

– Receives $\Delta_1^{(3)} \in \mathbb{F}_{3^{r(3)}}$ from $\mathcal{A}$ when forwarding the call $(\mathsf{sid}, \mathtt{init}, 1)$ to $\mathcal{F}_{\mathsf{sVOLE}}^{3,r(3)}$.
– Receives $\boldsymbol{K_a}$ from $\mathcal{A}$ when forwarding the call $\mathsf{Commit}(\boldsymbol{a}, 3, r^{(3)}, \Delta_1)$ to $\mathcal{F}_{\mathsf{sVOLE}}^{3,r(3)}$
– Receives $\boldsymbol{\delta}$ from $\mathcal{A}$
– Receives $(\boldsymbol{b}, \boldsymbol{M_b})$, $\boldsymbol{K_{c_0}}$, and $(\boldsymbol{c_1}, \boldsymbol{M_{c_1}})$ from $\mathcal{A}$ when forwarding the calls $\mathsf{Commit}(\boldsymbol{b}, 3, r^{(3)}, \Delta_0)$, $\mathsf{Commit}(\boldsymbol{c_0}, 3, r^{(3)}, \Delta_1)$ and $\mathsf{Commit}(\boldsymbol{c_1}, 3, r^{(3)}, \Delta_0)$, respectively.
– $\mathcal{S}$ defines the set $S_{\Delta_0} = \emptyset$. If at some point $|S_{\Delta_0}| > 1$ $\mathcal{S}$ aborts.
– To simulate the sacrifice step, $\mathcal{S}$ does the following
  - Receives the set S from $\mathcal{A}$ when forwarding the call $(\mathsf{sid}, \mathsf{subset}, C, M)$ to $\mathcal{F}_{\mathsf{random}}$.
  - For $i \in S$: Receives the openings (b_i, M_{b_i}) and $(c_{1_i}, M_{c_{1_i}})$ from $\mathcal{A}$.
  - If any of the openings deviate from protocol specification, let this deviation be (x', M') where (x, M) is the honest openings, the simulator checks. (1) If $x = x'$ abort. (2) If $x \neq x'$ compute $\Delta_1' = (M - M')(x - x')$ and add Δ_0' to S_{Δ_0}.
  - For $i \in S$: Samples $a_i \xleftarrow{\$} \mathbb{F}_3$ and set $M_{a_i} = \Delta_1 \cdot a_i + K_{a_i}$. Let $c_{0_i} = \mathsf{H}(M_{a_i}) + \delta_i \cdot a_i^2$ and $M_{c_{0_i}} = \Delta_1 \cdot c_{0_i} + K_{c_{0_i}}$.
  - If $c_{0_i} + c_{1_i} \neq 0$ abort.
– To simulate the checking correctness step, $\mathcal{S}$ does the following
  - Receives the sets S_i from $\mathcal{A}$ when forwarding the call $(\mathsf{sid}, \mathsf{buckets}, B, B^2\ell)$ to $\mathcal{F}_{\mathsf{random}}$.
  - For $i \in [B\ell]$, for $j \in S_i$: $\mathcal{S}$ receives the openings e_j, f_j and the corresponding MACs M_{e_j}, M_{f_j}.
  - If any of the openings deviate from protocol specification, let this deviation be (x', M') where (x, M) is the honest openings, the simulator checks. (1) If $x = x'$ abort. (2) If $x \neq x'$ compute $\Delta_0' = (M - M')(x - x')$ and add Δ_0' to S_{Δ_0}.
  - For $i \in [B\ell]$, for $j \in S_i$: $\mathcal{S}$ sets $(a_i, M_{a_i}) = (a_i \xleftarrow{\$} \mathbb{F}_3, \Delta_1 \cdot a_i + K_{a_i})$ and set $(c_{1_j}, M_{c_{1_j}}) = (\mathsf{H}(M_{a_i}) + \delta_i \cdot a_i^2, \Delta_1 \cdot c_{0_i} + K_{c_{0_i}})$
  - Opens d_j and f_j a protocol specification to $\mathcal{A}$. If any openings of $f_j \neq 0$ abort.
– To simulate the removing leakage step, $\mathcal{S}$ only check the openings from $\mathcal{A}$ and if any of them deviate from the protocol specification, add Δ_0' to S_{Δ_0} as previously defined or abort.

– If $|S_{\Delta_0}| = 1$, send $(\mathsf{sid}, \mathsf{guess}, S_{\Delta_0}, 3)$ to $\mathcal{F}_{\mathsf{mRand}}^{3,r^{(2)},r^{(3)}}$ (③). If the response is $(\mathtt{success}, \Delta_1)$ send $\boldsymbol{a}^1 = [2\mathsf{H}(\Delta_1 + K_{a_i}) + \mathsf{H}(2 \cdot \Delta_1 + K_{a_i})], \boldsymbol{w}_{\boldsymbol{a}^1} = [(a_j^1 - b_i) \cdot \Delta_1 + M_{b_i}], \boldsymbol{c}^1 = [-\mathsf{H}(K_{a_i})], \boldsymbol{w}_{\boldsymbol{c}^1} = [(c_{1_i} + \mathsf{H}(K_{a_i})) \cdot \Delta_1 + M_{c_{1_i}}]$ to $\mathcal{F}_{\mathsf{mRand}}^{3,r^{(2)},r^{(3)}}$ (③), otherwise abort.
– Otherwise, send the remaining $\boldsymbol{a}^1 = [b_i], \boldsymbol{w}_{\boldsymbol{a}^1} = [M_{b_i}], \boldsymbol{c}^1 = [c_{1_i}], \boldsymbol{w}_{\boldsymbol{c}^1} = [M_{c_{1_i}}]$ for $i \in [\ell]$ to $\mathcal{F}_{\mathsf{mRand}}^{3,r^{(2)},r^{(3)}}$ (③).

We now argue for the indistinguishability between the real and ideal worlds. The only messages sent from $\mathcal{S}$ to $\mathcal{A}$ are openings. In the ideal world, these openings are uniformly random since a_j is sampled uniformly random. This is the same distribution as in the real world. The value of $c_{0,j}$ depends on the value of a_j using the same formula in both the ideal world and the real world.

$\mathcal{A}$ can try to distinguish the worlds by using b_j to leak the value of a_j. If a successful leak occurs, $\mathcal{Z}$ can tell that the leaked values are different from the output values in the ideal world with good probability. By proposition 3 this happens with statistically low probability.

When the protocol finishes, the environment learns the honest party's output, and hence it can check whether the output values from the honest party are consistent with the real world. Since a_j is sampled in the real world, the output value of the ideal world will always be consistent. Given the value of a_j, the value of $c_{0,j}$ must be equal to $m_{a,j} + \delta_j \cdot a_j^2$. From the definition of the ideal functionality we show that this is indeed the case, since with very high probability the value of b_j and $c_{1,j}$ from $\mathcal{A}$ follows the protocol.

$$\begin{aligned} c_{0,j} &= a_j \cdot b_j - c_{1,j} \\ &= a_j \cdot (2 \cdot m_{1,j} + m_{2,j}) + m_{0,j} \\ &= m_{0,j} + 2a_j \cdot m_{1,j} + a_j \cdot m_{2,j} \end{aligned}$$

Setting $a_j \in \{0, 1, 2\}$ shows that $c_{0,j}$ is consistent with the real world, making $\mathcal{Z}$ unable to distinguish the two worlds. □

C Full Proofs

C.1 Local Doubly-Authenticated Bits

Proof. We first consider correctness when both parties behave honestly. As $\mathcal{P}_i$ commits to valid $\boldsymbol{b}$, the initial C opened entries will be verified correctly. Each bucket will also successfully verify, since the protocol $\Pi_{\mathsf{XORCheck}}^{p}$ emulates the exclusive-or operation correctly both mod 2 and mod p.

Next, consider a malicious $\mathcal{P}_i$, we construct the simulator as follows:

– Receives $\boldsymbol{b}^{(2)}, \boldsymbol{b}^{(p)}, \boldsymbol{M}_{\boldsymbol{b}^{(2)}}, \boldsymbol{M}_{\boldsymbol{b}^{(p)}}$ from $\mathcal{A}$ when simulating the calls $\mathsf{RandCommit}(\boldsymbol{b}, 2)$ and $\mathsf{Commit}(\boldsymbol{b}, p)$.
– Upon the call $\mathsf{Prove}([\![\mathsf{b}]\!]_{\mathsf{i}}^{(\mathsf{p})}, \mathsf{C})$, wait for $\mathcal{A}$ to send $(\mathsf{sid}_{\mathsf{VOLE}}, \mathsf{guess}, S_{\Delta^{(p)}})$ with $|S_{\Delta^{(p)}}| \leq 2$ and set $S_{\Delta^{(p)}} = \emptyset$ otherwise. Send $(\mathsf{sid}_{\mathsf{VOLE}}, \mathsf{guess}, \Delta_i^{(p)}, p)$ to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$ (⓪) for each $\Delta_i^{(p)} \in S_{\Delta^{(p)}}$. If the functionality return `success` send `success` to $\mathcal{A}$. Otherwise return `success` to $\mathcal{A}$ if $\boldsymbol{b}^{(2)} = \boldsymbol{b}^{(p)}$, else return `abort` to $\mathcal{A}$ and abort.
– Samples $S \subset [M]$ of size C and send S to $\mathcal{A}$.
– For each $j \in S$ wait for $\mathcal{A}$ to send $(\tilde{b}_j, M')$.
  • If $\tilde{b} = b_j$ and $M' = M_{b_j}$ continue
  • If $\tilde{b} = b_j$ and $M' \neq M_{b_j}$ abort, since the MAC does not verify
  • If $\tilde{b} \neq b_j$, the simulator computes $\Delta' = (M_{b_j} - M')(b_j - \tilde{b})$ and sends $(\mathsf{sid}_2, \mathsf{guess}, \Delta', 2)$ to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$ (①). If it returns $(\mathtt{success}, \Delta')$ the simulator puts $b_j \leftarrow \tilde{b}, M_{b_j} \leftarrow M'$ and continues otherwise abort.

– To simulate $\mathsf{BatchVer}((\llbracket z_j \rrbracket_i^{(p)})_{j\in S})$, the simulator samples $(\chi_j)_{j\in S} \in \mathbb{F}_{p^{r^{(p)}}}$ and send these to $\mathcal{A}$. Wait for $\mathcal{A}$ to respond with M' and check for all $j \in S$:

$$b_j^{(p)} - b_j^{(2)} = 0$$

and

$$M' \stackrel{?}{=} \sum_{j\in S} \chi_j \cdot M_{b_j^{(2)}} := M$$

If this does not hold we are in one of the following two cases

1. If the b_j verifies for all j, but M' does not, the simulator aborts
2. If there exists $j \in S$ such that the b_j does not verify the simulator computes

$$(M' - M) \cdot \left(\sum_{j\in S} \chi_j \cdot (b_j^{(2)}) - b_j^{(p)} \right)^{-1}$$

and send $(\mathsf{sid}_p, \mathtt{guess}, \Delta^{(p)})$ to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}(①)$. It `success` is returned continue, otherwise abort.

– Sample random permutation, π and send π to $\mathcal{A}$. Assign the remaining $\boldsymbol{b}^{(2)}, \boldsymbol{b}^{(p)}, \boldsymbol{M}_{\boldsymbol{b}^{(2)}}, \boldsymbol{M}_{\boldsymbol{b}^{(p)}}$ into buckets according to π.

– To simulate $\Pi^p_{\mathsf{XORCheck}}((\llbracket b_\nu \rrbracket^{(2)}, \llbracket b_\nu \rrbracket^{(p)})_{\nu\in P_j})$ wait for $\mathcal{A}$ to send $(c', M_{c'})$, check if $c' \stackrel{?}{=} \sum_{\nu\in P_j} b_\nu^{(2)} := c$ and $M_{c'} \stackrel{?}{=} \sum_{\nu\in P_j} M_{b_\nu^{(2)}} := M_c$

- If $c' = c$ and $M' = M_c$ continue.
- If $c' = c$ and $M' \neq M_c$ abort, since the MAC does not verify.
- If $c' \neq c$, the simulator computes $\Delta' = (M_c - M')(c - c')$ and sends $(\mathsf{sid}_2, \mathtt{guess}, \Delta', 2)$ to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}(①)$. If it returns $(\mathtt{success}, \Delta')$ the simulator puts $c \leftarrow c', M_c \leftarrow M'$ and continues otherwise abort.

Upon the $\mathsf{Prove}(\llbracket \boldsymbol{y} \rrbracket^{(p)}, C)$ call to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}(①)$, wait for $\mathcal{A}$ to send $(\mathsf{sid}_p, \mathtt{guess}, S_{\Delta^{(p)}})$ with $|S_{\Delta^{(p)}}| \leq m/2$ and set $S_{\Delta^{(p)}} = \emptyset$ otherwise. Send $(\mathsf{sid}_{\mathsf{VOLE}}, \mathtt{guess}, \Delta_i^{(p)}, p)$ to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}(⓪)$ for each $\Delta_i^{(p)} \in S_{\Delta^{(p)}}$. If the functionality return `success` return `success` to $\mathcal{A}$. Otherwise return `success` to $\mathcal{A}$ if $(-1)^c \prod_{\nu\in P_j}(2 \cdot b_\nu^{(p)} - 1) = c$, else return `abort` to $\mathcal{A}$ and abort.

– For each bucket add $(b_1^{(2)}, \ldots b_k^{(2)})$, $(b_1^{(p)}, \ldots b_k^{(p)})$, $(M_{b_1^{(2)}}, \ldots, M_{b_k^{(2)}})$, and $(M_{b_1^{(p)}}, \ldots, M_{b_k^{(p)}})$ to L

– Send L to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}(①)$.

We now argue for indistinguishability between the real and the ideal world. The messages sent from $\mathcal{P}_{i\oplus 1}$ to $\mathcal{A}$ during the protocol are all locally sampled, hence the ideal and the real world are identically distributed. So what remains it show the both world have the same aborting behaviour and consistent outputs.

During the initial to Prove both worlds abort if the proof is invalid unless $\mathcal{A}$ guesses $\Delta^{(p)}$. During all openings the worlds have the same aborting behaviour, if the MAC is incorrect the world abort, except if $\mathcal{A}$ guesses the global key $\Delta^{(2)}$. Likewise the call to $\mathsf{BatchVer}$ have same aborting behaviour, as both world abort if the MAC is incorrect, except if $\mathcal{A}$ guesses $\Delta^{(p)}$. The simulator The last call to Prove during the $\Pi^p_{\mathsf{XORCheck}}$ sub-protocol the ideal world abort if any of $b_\nu^{(p)} \neq b_\nu^{(2)}$. The real world aborts similarly except if $\mathcal{A}$ succeed in getting d or more corrupt $b^{(3)}$ into a bucket. By Lemma 2 this makes the ideal and the real distinguishable with statistically small probability.

Now, consider a malicious $\mathcal{P}_{i\oplus 1}$, we construct the simulator as follows:

– Receives $\Delta^{(2)}, \Delta^{(p)}$ from $\mathcal{A}$ when simulating calls $\mathsf{sid}_2, \mathtt{init}, i \oplus 1)$ and $\mathsf{sid}_p, \mathtt{init}, i \oplus 1)$.
– Receives $\boldsymbol{K}_{\boldsymbol{b}^{(2)}}$ and $\boldsymbol{K}_{\boldsymbol{b}^{(p)}}$ from $\mathcal{A}$ when simulating the calls $\mathsf{RandCommit}(\boldsymbol{b}, 2)$ and $\mathsf{Commit}(\boldsymbol{b}, p)$.
– Upon the $\mathsf{Prove}([\![\mathsf{b}]\!]_{\mathsf{i}}^{(\mathsf{p})}, \mathsf{C})$ send `success` to $\mathcal{A}$.
– Wait for $\mathcal{A}$ to send S. For $j \in S$ sample $b_j \xleftarrow{\$} \mathbb{F}_2$ and send $b_j, M_{b_j} = b_j \cdot \Delta_2 + K_{b_j^{(2)}}$ to $\mathcal{A}$.
– To simulate $\mathsf{BatchVer}((\llbracket z_j \rrbracket_i^{(p)})_{j \in S})$ the simulator computes $K_{z_j} = K_{b_j^{(2)}} + K_{b_j^{(p)}}$ for $j \in S$. Wait for $\mathcal{A}$ to send $(\chi_j)_{j \in S}$ and respond with

$$M = \sum_{j \in S} \chi_j \cdot K_{z_j}.$$

– Wait for $\mathcal{A}$ to send a permutation π.
– Assign the remaining $\boldsymbol{K}_{\boldsymbol{b}^{(2)}}$ and $\boldsymbol{K}_{\boldsymbol{b}^{(p)}}$ into buckets according to π.
– The simulator does the following for each bucket. For each P_j for $j \in [B-k]$ the simulator samples $c_j \xleftarrow{\$} \mathbb{F}_2$ and set $M_{c_j^{(2)}} = c_j \cdot \Delta^{(2)} + \sum_{\nu \in P_j} K_{b_\nu^{(2)}}$. The simulator sends $(c_j, M_{c_j^{(2)}})$ to $\mathcal{A}$ to simulate the opening, and sends `success` to $\mathcal{A}$ on the $\mathsf{Prove}(\llbracket y \rrbracket^{(p)}, C)$

We now argue for indistinguishability between the ideal and the real world. It is clear that the simulator perfectly simulates the ideal functionalities. The openings of b_j for $j \in S$ are perfectly simulated, as in the real world b_j is sampled uniformly random. The corresponding opened MAC is computed from the knowledge from $\mathcal{A}$ as the valid MAC. During batched verification, each z_j is expected to be 0 hence $M_{z_j} = K_{z_j}$. The simulator can compute this value and make the verification succeed. Each bucket is also simulated perfectly. Each b in sampled uniformly random in the real world and for $P = [P' | I_{B-k}]$ both P and P' is full row rank the value c_j opened will be uniformly random for $\mathcal{A}$. Thus, sampling c_j in the ideal world have identical distribution. Furthermore after the protocol gives output to the environment, it cannot compute the expected value of c_j since P' is full row rank, thus the masking of the revealed bits are also uniformly random to the environment. □

C.2 Doubly Authenticated Bits

Proof. If both parties behave honestly, correctness of the protocol can be seen as follows. First note that for each $i \in \{0,1\}$, $\llbracket \boldsymbol{b}^i \rrbracket_i^{(2)}$ and $\llbracket \boldsymbol{b}^i \rrbracket_i^{(p)}$ are consistent commitments to the same vector of random bits $\boldsymbol{b}^i \xleftarrow{\$} \{0,1\}^\ell$ by definition of $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$ (①). Moreover, $\llbracket \boldsymbol{a}^0 \rrbracket_0^{(p)}$ and $\llbracket \boldsymbol{a}^1 \rrbracket_1^{(p)}$ are commitments to random $\boldsymbol{a}^0, \boldsymbol{a}^1 \xleftarrow{\$} \mathbb{F}_p^\ell$ and $\langle\!\langle \boldsymbol{c} \rangle\!\rangle^{(p)}$ forms an authenticated secret sharing of $\boldsymbol{c} := \boldsymbol{a}^0 \odot \boldsymbol{a}^1 \bmod p$ by definition of $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$ (③). Hence $\mathsf{Mult}\left((\llbracket \boldsymbol{b}^0 \rrbracket_0, \llbracket \boldsymbol{b}^1 \rrbracket_1), (\llbracket \boldsymbol{a}^0 \rrbracket_0, \llbracket \boldsymbol{a}^1 \rrbracket_1, \langle\!\langle \boldsymbol{c} \rangle\!\rangle)\right)$ outputs a correct sharing $\langle\!\langle \boldsymbol{b}^0 \odot \boldsymbol{b}^1 \rangle\!\rangle^{(p)}$ as detailed in Section 3.3 and finally $\langle\!\langle \boldsymbol{b} \rangle\!\rangle^{(p)} = \langle\!\langle \boldsymbol{b}^0 \rangle\!\rangle^{(p)} + \langle\!\langle \boldsymbol{b}^1 \rangle\!\rangle^{(p)} - 2 \cdot \langle\!\langle \boldsymbol{b}^0 \odot \boldsymbol{b}^1 \rangle\!\rangle^{(p)}$ forms a sharing of the same bits as $\langle\!\langle \boldsymbol{b} \rangle\!\rangle^{(2)}$ as it emulates the XOR of $\boldsymbol{b}^0$ and $\boldsymbol{b}^1$ modulo p.

Now let party $\mathcal{P}_0$ be maliciously corrupted by the adversary $\mathcal{A}$ (the case where $\mathcal{P}_1$ is corrupted will be analogous as the protocol is symmetric in the role of the parties). The simulator:

– Receives $\Delta_0^{(2)} \in \mathbb{F}_2^{r^{(2)}}$ and $\Delta_0^{(p)} \in \mathbb{F}_p^{r^{(p)}}$ from $\mathcal{A}$ after the $(\mathsf{sid}, \mathtt{init}, 0, q)$ calls to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$ for $q \in \{2, p\}$.
– Receives $\boldsymbol{b}^0 \in \{0,1\}^\ell$, $\boldsymbol{M}_{\boldsymbol{b}^0}^{(2)}, \boldsymbol{K}_{\boldsymbol{b}^1}^{(2)} \in \mathbb{F}_{2^{r^{(2)}}}^\ell$ and $\boldsymbol{M}_{\boldsymbol{b}^0}^{(p)}, \boldsymbol{K}_{\boldsymbol{b}^1}^{(p)} \in \mathbb{F}_{p^{r^{(p)}}}^\ell$ from $\mathcal{A}$ after the $(\mathsf{sid}, \mathtt{laBits}, \ell, 0)$ and $(\mathsf{sid}, \mathtt{laBits}, \ell, 1)$ calls to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$.
– Receives $\boldsymbol{a}^0, \boldsymbol{c}^0 \in \mathbb{F}_p^\ell$ and $\boldsymbol{M}_{\boldsymbol{a}^0}^{(p)}, \boldsymbol{M}_{\boldsymbol{c}^0}^{(p)}, \boldsymbol{K}_{\boldsymbol{a}^1}^{(p)}, \boldsymbol{K}_{\boldsymbol{c}^1}^{(p)} \in \mathbb{F}_{p^{r^{(p)}}}^\ell$ from $\mathcal{A}$ after the $(\mathsf{sid}, \mathtt{aOLE}, p, \ell)$ call to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$.

– To simulate $\mathsf{BatchOpen}\left(\llbracket d_1 \rrbracket_0^{(p)}, \ldots, \llbracket d_\ell \rrbracket_0^{(p)}\right)$:
  - Sample $(\chi_1, \ldots, \chi_\ell) \xleftarrow{\$} \mathbb{F}_{p^{r^{(p)}}}^{\ell}$ and send these to $\mathcal{P}_0$.
  - Wait for $\mathcal{P}_0$ to send $\boldsymbol{d}' \in \mathbb{F}_p^\ell$ and $M' \in \mathbb{F}_{p^r}$ and check if $\boldsymbol{d}' \stackrel{?}{=} \boldsymbol{d}$ and $M' \stackrel{?}{=} \sum_{i=1}^{\ell} \chi_i \cdot M_{d_i}$, where $\boldsymbol{d} := \boldsymbol{a}^0 - \boldsymbol{b}^0$ and $\boldsymbol{M}_{\boldsymbol{d}}^{(p)} := \boldsymbol{M}_{\boldsymbol{a}^0}^{(p)} - \boldsymbol{M}_{\boldsymbol{b}^0}^{(p)}$, and continue if this holds. Otherwise, we are in one of the following cases:
    1. If $\boldsymbol{d}' = \boldsymbol{d}$ but $M' \neq \sum_{i=1}^{\ell} \chi_i \cdot M_{d_i}$, the simulator aborts since this means the MACs do not verify.
    2. If $\boldsymbol{d}' \neq \boldsymbol{d}$, the simulator computes $\Delta' := \left(\sum_{i=1}^{\ell} \chi_i \cdot M_{d_i}^{(p)} - M'\right) \cdot \left(\sum_{i=1}^{\ell} \chi_i \cdot (d_i - d_i')\right)^{-1}$ and sends the guess $(\mathsf{sid}, \mathtt{guess}, \Delta', p)$ to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$. If $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$ returns $(\mathtt{success}, \Delta')$, the simulator puts $\boldsymbol{M}_{\boldsymbol{d}}^{(p)} \leftarrow \Delta' \cdot (\boldsymbol{d}' - \boldsymbol{d}) + \boldsymbol{M}_{\boldsymbol{d}}^{(p)}$ and $\boldsymbol{d} \leftarrow \boldsymbol{d}'$, and continues. Otherwise, the simulator aborts.

– To simulate $\mathsf{BatchOpen}\left(\llbracket e_1 \rrbracket_1^{(p)}, \ldots, \llbracket e_\ell \rrbracket_1^{(p)}\right)$:
  - Wait for $\mathcal{P}_0$ to send $(\chi_1, \ldots, \chi_\ell) \in \mathbb{F}_{p^{r^{(p)}}}^{\ell}$.
  - Sample $\boldsymbol{e} \xleftarrow{\$} \mathbb{F}_p^\ell$ and send $\boldsymbol{e}$, $M' := \sum_{i=1}^{\ell} \chi_i \cdot \Delta_0^{(p)} \cdot e_i + \sum_{i=1}^{\ell} \chi_i \cdot K_{e_i}^{(p)}$ to $\mathcal{P}_0$, where $K_{e_i}^{(p)} := K_{a_i^1}^{(p)} - K_{b_i^1}^{(p)}$.

– Sends $\boldsymbol{b}_{(2)}^0 := \boldsymbol{b}^0$, $\boldsymbol{M}_{\boldsymbol{b}_{(2)}^0}^{(2)} := \boldsymbol{M}_{\boldsymbol{b}^0}^{(2)}$ and $\boldsymbol{b}_{(p)}^0 := \boldsymbol{b}^0 + \boldsymbol{e} \odot \boldsymbol{d} - \boldsymbol{e} \odot \boldsymbol{a}^0 + \boldsymbol{c}^0$, $\boldsymbol{M}_{\boldsymbol{b}_{(p)}^0}^{(p)} := \boldsymbol{M}_{\boldsymbol{b}^0}^{(p)} - \boldsymbol{e} \odot \boldsymbol{M}_{\boldsymbol{a}^0}^{(p)} + \boldsymbol{M}_{\boldsymbol{c}^0}^{(p)}$ as well as $\boldsymbol{K}_{\boldsymbol{b}_{(2)}^1}^{(2)} := \boldsymbol{K}_{\boldsymbol{b}^1}^{(2)}$ and $\boldsymbol{K}_{\boldsymbol{b}_{(p)}^1}^{(p)} := \boldsymbol{K}_{\boldsymbol{b}^1}^{(p)} - \boldsymbol{d} \odot \boldsymbol{K}_{\boldsymbol{a}^1}^{(p)} + \boldsymbol{K}_{\boldsymbol{c}^1}^{(p)}$ to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$ (②).

We now argue for indistinguishability between the real world and the ideal world using the simulator. $\mathcal{S}$ sends two messages outside the ideal functionalities, i.e., the challenges $(\chi_1, \ldots, \chi_\ell)$ and the openings $(\boldsymbol{e}, M')$. The challenges are sampled uniformly random in both worlds and thus have identical distribution. The openings of $\llbracket \boldsymbol{e} \rrbracket_1^{(p)}$ contain two values, first the value of $\boldsymbol{e}$ is sampled uniformly random in the ideal world, where in the real world $\llbracket \boldsymbol{e} \rrbracket_1^{(p)} = \llbracket \boldsymbol{a}^1 \rrbracket_1^{(p)} - \llbracket \boldsymbol{b}^1 \rrbracket_1^{(p)}$. The environment will learn $\llbracket \boldsymbol{b}^1 \rrbracket_1^{(p)}$, since $\llbracket \boldsymbol{b}^1 \rrbracket_1^{(2)}$ is part of the output and by definition of `laBits`, $\boldsymbol{b}^{1,(2)} = \boldsymbol{b}^{1,(p)}$. However, since $\llbracket \boldsymbol{a}^1 \rrbracket_1^{(p)}$ is uniformly random by definition and remains unknown to the environment, the value of $\boldsymbol{e}$ is identically distributed. The valid corresponding MAC, M' for $\boldsymbol{e}$ is computed using the knowledge of $\Delta_0^{(p)}$, $\boldsymbol{K}_{\boldsymbol{a}^1}^{(p)}$, and $\boldsymbol{K}_{\boldsymbol{b}^1}^{(p)}$. Since $\boldsymbol{e}$ is uniformly random, M' will also be uniformly random, hence have identical distribution with the real world. □

C.3 Authenticated VOLE

Proof. When both parties behave honestly, it is simple to see that the protocol is correct. The parties obtain shares $\boldsymbol{v} + \boldsymbol{w} = \gamma \cdot \boldsymbol{y}$ from the $\mathsf{Commit}(\boldsymbol{y}, q^r, 1, \gamma)$ call to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$. Subsequently, $\mathsf{Commit}(\Delta_0 \cdot \boldsymbol{y}, q^r, 1, \gamma)$ and $\mathsf{Commit}(\boldsymbol{y}, q^r, 1, \Delta_1 \cdot \gamma)$ give shares

$$\tilde{\boldsymbol{w}} + \tilde{\boldsymbol{v}} = \gamma \cdot (\Delta_0 \cdot \boldsymbol{y}) = \Delta_0 \cdot (\boldsymbol{w} + \boldsymbol{v})$$
$$\bar{\boldsymbol{w}} + \bar{\boldsymbol{v}} = (\Delta_1 \cdot \gamma) \cdot \boldsymbol{y} = \Delta_1 \cdot (\boldsymbol{w} + \boldsymbol{v})$$

Rewriting the above equations shows that this gives valid MACS $\llbracket \boldsymbol{w} \rrbracket_{0,\mathsf{MAC}}^{(q^r)} := ((\boldsymbol{w}, \bar{\boldsymbol{w}}), (\Delta_1, \Delta_1 \cdot \boldsymbol{v} - \bar{\boldsymbol{v}}))$ and $\llbracket \boldsymbol{v} \rrbracket_{1,\mathsf{MAC}}^{(q^r)} := ((\Delta_0, \Delta_0 \cdot \boldsymbol{w} - \tilde{\boldsymbol{w}}), (\boldsymbol{v}, \tilde{\boldsymbol{v}}))$. The rest of the protocol steps are there to enforce correct behaviour from the parties, and all checks will trivially pass if the parties are honest.

Consider a maliciously corrupted receiver $\mathcal{P}_0$. We consider a simulator as follows:

– Receive $\Delta_0 \in \mathbb{F}_{q^r}$ from $\mathcal{A}$ upon the $(\mathsf{sid}_{\mathsf{MAC}}, \mathtt{init}, 0, q)$ call to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$.
– Receive $\tilde{\Delta}_0, a \in \mathbb{F}_{q^r}$ from $\mathcal{A}$ upon the $\mathsf{Commit}(\Delta_0, q^r, 1, \gamma)$ call to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$ and $c \in \mathbb{F}_{q^r}$ upon the $\mathsf{Commit}(\gamma, q^r, 1, \Delta_0)$ call to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$.
– Wait for $\mathcal{P}_0$ to send $(\mathsf{sid}, \mathtt{check}, \tilde{a}, 0)$ to $\mathcal{F}_{\mathsf{EQ}}$:
  - If $\tilde{\Delta}_0 = \Delta_0$ and $\tilde{a} = a - c$, the simulator sends 1 to $\mathcal{P}_0$ and continues.
  - If $\tilde{\Delta}_0 = \Delta_0$ and $\tilde{a} \neq a - c$, the simulator sends $\mathtt{abort}$ to $\mathcal{P}_0$ and aborts.
  - If $\tilde{\Delta}_0 \neq \Delta_0$, the simulator computes $\gamma' := (\tilde{a} - a + c) \cdot (\Delta_0 - \tilde{\Delta}_0)^{-1}$ and sends $(\mathsf{sid}_{\mathsf{VOLE}}, \mathsf{guess}, \gamma', q)$ to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$. If the functionality returns $\mathtt{success}$, the simulator sends 1 to $\mathcal{P}_0$ and continues. Otherwise, the simulator sends $\mathtt{abort}$ to $\mathcal{P}_0$ and aborts.
– Receive $(\boldsymbol{y} \| s), (\boldsymbol{w} \| e) \in \mathbb{F}_{q^r}^{\ell+1}$ from $\mathcal{A}$ upon the $\mathsf{Commit}((\boldsymbol{y} \| s), q^r, 1, \gamma)$ call, $(\boldsymbol{w}' \| e'), (\boldsymbol{t} \| \tau) \in \mathbb{F}_{q^r}^{\ell+1}$ upon the $\mathsf{Commit}((\boldsymbol{w} \| e), q^r, 1, \Delta_1)$ call and $\tilde{\boldsymbol{y}}, \tilde{\boldsymbol{w}} \in \mathbb{F}_{q^r}^{\ell}$ upon the $\mathsf{Commit}(\Delta_0 \cdot \boldsymbol{y}, q^r, 1, \gamma)$ call to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$.
– Upon the $(\mathsf{sid}_{\mathsf{VOLE}}, \mathtt{prove}, \mathsf{id}_1, \ldots, \mathsf{id}_{2\ell+1}, C_1, \ldots, C_\ell, q)$ call to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$, wait for $\mathcal{A}$ to send $(\mathsf{sid}_{\mathsf{VOLE}}, \mathsf{guess}, S_\gamma)$ with $|S_\gamma| \leq 2$, and set $S_\gamma := \emptyset$ otherwise. Send $(\mathsf{sid}_{\mathsf{VOLE}}, \mathsf{guess}, \gamma', q)$ to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$ for each $\gamma' \in S_\gamma$. If the functionality returns $\mathtt{success}$, return $\mathtt{success}$ to $\mathcal{P}_0$. Otherwise, continue and return $\mathtt{success}$ to $\mathcal{P}_0$ if $\Delta_0 \cdot \boldsymbol{y} = \tilde{\boldsymbol{y}}$, and return $\mathtt{abort}$ and abort otherwise.
– Receive $(\bar{\boldsymbol{y}} \| \bar{s}), (\bar{\boldsymbol{w}} \| \bar{e}) \in \mathbb{F}_{q^r}^{\ell+1}$ from $\mathcal{A}$ upon the $\mathsf{Commit}((\boldsymbol{y} \| s), q^r, 1, \Delta_{\mathsf{Mask}} \to \Delta_1 \cdot \gamma)$ call to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$.
– Wait for $\mathcal{P}_0$ to send $(\mathsf{sid}, \mathtt{check}, \tilde{t}, 0)$ to $\mathcal{F}_{\mathsf{EQ}}$. The simulator recovers the first $\boldsymbol{z} \in \mathbb{F}_{q^r}^{\ell+1}$ such that $\mathsf{H}(\boldsymbol{z}) = \tilde{t}$ from the list of $\mathcal{P}_0$'s random oracle queries (if such a $\boldsymbol{z}$ does not exist, the simulator sends 0 to $\mathcal{P}_0$, sends $\mathtt{abort}$ to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$ (④) and aborts) and does:
  - If $\boldsymbol{z} = (\bar{\boldsymbol{w}} \| \bar{e}) - (\boldsymbol{t} \| \tau)$ and $(\boldsymbol{w}' \| e') = (\boldsymbol{w} \| e)$ and $(\bar{\boldsymbol{y}} \| \bar{s}) = (\boldsymbol{y} \| s))$, the simulator sends 1 to $\mathcal{P}_0$ and continues.
  - If $\boldsymbol{z} \neq (\bar{\boldsymbol{w}} \| \bar{e}) - (\boldsymbol{t} \| \tau)$ and $(\boldsymbol{w}' \| e') = (\boldsymbol{w} \| e)$ and $(\bar{\boldsymbol{y}} \| \bar{s}) = (\boldsymbol{y} \| s))$, the simulator sends 0 to $\mathcal{P}_0$, sends $\mathtt{abort}$ to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$ (④) and aborts.
  - If $((\boldsymbol{w}' \| e') \neq (\boldsymbol{w} \| e)$ or $(\bar{\boldsymbol{y}} \| \bar{s}) \neq (\boldsymbol{y} \| s))$, the simulator puts $\boldsymbol{a}' := (\boldsymbol{w}' \| e') - (\boldsymbol{w} \| e)$, $\boldsymbol{b}' := (\bar{\boldsymbol{y}} \| \bar{s}) - (\boldsymbol{y} \| s)$ and $\boldsymbol{c}' := \boldsymbol{z} - (\bar{\boldsymbol{w}} \| \bar{e}) + (\boldsymbol{t} \| \tau)$ and sends $(\mathsf{sid}_{\mathsf{VOLE}}, \mathsf{sid}_{\mathsf{MAC}}, \mathtt{guess}^*, (a'_i, b'_i, c'_i), q)$ to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$ (④) for each $i \in [\ell + 1]$. If the functionality returns $\mathtt{success}$ on all instances, the simulator sends 1 to $\mathcal{P}_0$, and continues if $\mathcal{P}_0$ does. Otherwise, if the functionality returns $\mathtt{fail}$ on any instance, the simulator sends $\mathtt{abort}$ to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$ (④) and aborts.
– Sends $\boldsymbol{y}, \boldsymbol{w}$, $\boldsymbol{M}_{\boldsymbol{w}} := \bar{\boldsymbol{w}}$ and $\boldsymbol{K}_{\boldsymbol{v}} := \Delta_0 \cdot \boldsymbol{w} - \tilde{\boldsymbol{w}}$ to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$ (④).

It is clear that the messages $\mathcal{A}$ receive from the ideal functionalities are indistinguishable from those of the simulator's. Thus, it remains to argue that the simulator perfectly simulates the message and aborting behaviour of the honest $\mathcal{P}_1$.

The protocol can abort at three different points during execution, one for each call to $\mathcal{F}_{\mathsf{EQ}}$ and during the evaluation of $C_1, \ldots, C_\ell$. During the first call to $\mathcal{F}_{\mathsf{EQ}}$ the real world aborts when the input $\tilde{a} \neq d - b$. In particular, this happens if $\mathcal{A}$ commits to a $\tilde{\Delta}_0 \neq \Delta_0$, unless $\mathcal{A}$ can guess γ, and therefore can alter their commitment to produce the correct input to $\mathcal{F}_{\mathsf{EQ}}$. The simulator simulates this perfectly, as a commitment to $\tilde{\Delta}_0 \neq \Delta_0$ together with $\tilde{a}$ can be used to extract the guess on γ and return 1 in the case this is correct.

During the evaluation of $C_1, \ldots, C_\ell$ the real world abort when $[\![\Delta_0]\!]_{1,0,\mathsf{VOLE}}^{(q^1)} \cdot [\![\boldsymbol{y}]\!]_{1,0,\mathsf{VOLE}}^{(q^1)} \neq [\![\tilde{\boldsymbol{y}}]\!]_{1,0,\mathsf{VOLE}}^{(q^r)}$. This happens when $\tilde{\boldsymbol{y}} \neq \Delta_0 \cdot \boldsymbol{y}$, unless $\mathcal{A}$ can guess γ and thus alter their commitment

during the evaluation to ensure success. The simulator simulates this perfectly, as if $\mathcal{A}$ guesses γ it returns success, and otherwise only returns success if $\tilde{\boldsymbol{y}} = \Delta_0 \cdot \boldsymbol{y}$.

During the second call to $\mathcal{F}_{\mathsf{EQ}}$ the real world aborts when $\mathcal{A}$ inputs $\boldsymbol{z} \neq (\boldsymbol{u}\|\sigma) - (\bar{\boldsymbol{v}}\|\bar{f}) + \Delta_1 \cdot (\boldsymbol{v}\|f)$ as input to $\mathcal{F}_{\mathsf{EQ}}$. In particular, this happens when $\mathcal{A}$ commits to values $(\boldsymbol{w}'\|e') \neq (\boldsymbol{w}\|e)$ or $(\bar{\boldsymbol{y}}\|\bar{s}) \neq (\boldsymbol{y}\|s)$, except if $\mathcal{A}$ can guess a combination (a, b, c) such that $c = a \cdot \Delta_1 + b \cdot (\Delta_1 \cdot \gamma)$. That is, guessing such a linear combination allows them to adapt their input to $\mathcal{F}_{\mathsf{EQ}}$ as $\boldsymbol{z} := (\bar{\boldsymbol{w}}\|\bar{e}) - (\boldsymbol{t}\|\tau) + \Delta_1 \cdot ((\boldsymbol{w}'\|e') - (\boldsymbol{w}\|e)) + (\Delta_1 \cdot \gamma) \cdot ((\bar{\boldsymbol{y}}\|\bar{s}) - (\boldsymbol{y}\|s))$. The simulator perfectly simulates this as it aborts in the case the input to $\mathcal{F}_{\mathsf{EQ}}$ or the committed values are wrong, except if $\mathcal{A}$ guesses (a, b, c) correctly for all adapted entries, which can be checked by the simulator using the special guess command.

Lastly, the environment can check whether the output from the parties is consistent across the worlds. First, the shares $(\boldsymbol{w}, \boldsymbol{v})$ of the product $\gamma \cdot \boldsymbol{y}$ are trivially consistent by calling the ideal functionality. The authentication of $\boldsymbol{v}$ is also consistent, as in the ideal world $\tilde{\boldsymbol{v}}$ is calculated from $\boldsymbol{K}_{\boldsymbol{v}}$. $\boldsymbol{K}_{\boldsymbol{v}}$ can only be wrong if $\tilde{\boldsymbol{w}}$ is wrong, which happens with negligible probability when evaluating $C_1, \ldots, C_\ell$. The authentication of $\boldsymbol{w}$, is calculated from $\boldsymbol{w}$ and $\bar{\boldsymbol{w}}$. The authentication can only be wrong if $\bar{\boldsymbol{w}}$ is wrong, which following the second call to $\mathcal{F}_{\mathsf{EQ}}$ only occurs with negligible probability.

Now consider a maliciously corrupted sender $\mathcal{P}_1$. The simulator behaves as follows:

- Receive $\Delta_1, \gamma \in \mathbb{F}_{q^r}$ from $\mathcal{A}$ upon the $(\mathsf{sid}_{\mathsf{MAC}}, \mathtt{init}, 1, q)$ and $(\mathsf{sid}_{\mathsf{VOLE}}, \mathtt{init}, 1, q)$ calls to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$, respectively.
- Receive $\Delta_{\mathsf{Mask}} \in \mathbb{F}_{q^r}$ from $\mathcal{A}$ upon the $(\mathsf{sid}_{\mathsf{Mask}}, \mathtt{init}, 1, q)$ call to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$.
- Receive $b \in \mathbb{F}_{q^r}$ from $\mathcal{A}$ upon the $\mathsf{Commit}(\Delta_0, q^r, 1, \gamma)$ call to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$ and $\tilde{\gamma}, d \in \mathbb{F}_{q^r}$ upon the $\mathsf{Commit}(\gamma, q^r, 1, \Delta_0)$ call to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$.
- Wait for $\mathcal{P}_1$ to send $(\mathsf{sid}, \mathtt{check}, \tilde{d}, 0)$ to $\mathcal{F}_{\mathsf{EQ}}$:
  - If ($\tilde{\gamma} = \gamma$ and $\tilde{d} = d - b$), the simulator sends 1 and $\tilde{c} := \tilde{d}$ to $\mathcal{P}_1$ and continues if $\mathcal{P}_0$ does.
  - If ($\tilde{\gamma} = \gamma$ and $\tilde{d} \neq d - b$), the simulator sends 0 and $\tilde{c} := d - b$ to $\mathcal{P}_1$, sends $\mathtt{abort}$ to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$(④) and aborts.
  - If $\tilde{\gamma} \neq \gamma$ the simulator computes $\Delta_0' := (\tilde{d} - d + b) \cdot (\gamma - \tilde{\gamma})^{-1}$ and sends $(\mathsf{sid}_{\mathsf{MAC}}, \mathtt{guess}, \Delta_0', q)$ to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$. If the functionality returns $\mathtt{success}$, the simulator puts $\Delta_0 := \Delta_0'$ and sends 1 and $\tilde{c} := \tilde{d}$ to $\mathcal{P}_0$, and continues if $\mathcal{P}_0$ does. If the functionality returns $\mathtt{fail}$, the simulator sends $(\mathsf{sid}_{\mathsf{VOLE}}, \mathsf{sid}_{\mathsf{MAC}}, \mathtt{abort}^*, q)$ to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$(④) and receives Δ_0. The simulator sends 0 and $\tilde{c} := d - b + \Delta_0 \cdot (\gamma - \tilde{\gamma})$ to $\mathcal{P}_1$ and aborts.
- Receive $(\boldsymbol{v}\|f) \in \mathbb{F}_{q^r}^{\ell+1}$ from $\mathcal{A}$ upon the $\mathsf{Commit}((\boldsymbol{y}\|s), q^r, 1, \gamma)$ call, $(\boldsymbol{u}, \sigma) \in \mathbb{F}_{q^r}^{(\ell+1)}$ upon the $\mathsf{Commit}((\boldsymbol{w}\|e), q^r, 1, \Delta_1)$ call and $\tilde{\boldsymbol{v}} \in \mathbb{F}_{q^r}^{\ell}$ upon the $\mathsf{Commit}(\Delta_0 \cdot \boldsymbol{y}, q^r, 1, \gamma)$ call to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$.
- Upon the $(\mathsf{sid}, \mathtt{prove}, \mathsf{id}_1, \ldots, \mathsf{id}_{2\ell+1}, C_1, \ldots, C_\ell, q)$ call to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$, return $\mathtt{success}$ to $\mathcal{P}_1$.
- Receive $\tilde{\Delta}_1 \in \mathbb{F}_{q^r}$ and $(\bar{\boldsymbol{v}}\|\bar{f}) \in \mathbb{F}_{q^r}^{\ell+1}$ from $\mathcal{A}$ upon the $\mathsf{Commit}((\boldsymbol{y}\|s), q^r, 1, \Delta_{\mathsf{Mask}} \to \Delta_1 \cdot \gamma)$ call to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$.
- Wait for $\mathcal{P}_1$ to send $(\mathsf{sid}, \mathtt{check}, \tilde{u}, 0)$ to $\mathcal{F}_{\mathsf{EQ}}$. The simulator recovers the first $\boldsymbol{z} \in \mathbb{F}_{q^r}^{\ell+1}$ such that $\mathsf{H}(\boldsymbol{z}) = \tilde{u}$ from $\mathcal{P}_1$'s list of random oracle queries (if such a $\boldsymbol{z}$ does not exist, the simulator sends 0 to $\mathcal{P}_1$, sends $\mathtt{abort}$ to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$(④) and aborts) and does:
  - If $\boldsymbol{z} = (\boldsymbol{u}\|\sigma) - (\bar{\boldsymbol{v}}\|\bar{f}) + \Delta_1 \cdot (\boldsymbol{v}\|f)$ and $\tilde{\Delta}_1 = \Delta_1 \cdot \gamma$, send 1 and $\tilde{t} := \tilde{u}$ to $\mathcal{P}_1$ and continue.
  - If $\boldsymbol{z} \neq (\boldsymbol{u}\|\sigma) - (\bar{\boldsymbol{v}}\|\bar{f}) + \Delta_1 \cdot (\boldsymbol{v}\|f)$ and $\tilde{\Delta}_1 = \Delta_1 \cdot \gamma$, send 0 and $\tilde{t} := \mathsf{H}((\boldsymbol{u}\|\sigma) - (\bar{\boldsymbol{v}}\|\bar{f}) + \Delta_1 \cdot (\boldsymbol{v}\|f))$ to $\mathcal{P}_1$, send $\mathtt{abort}$ to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$(④) and abort.

- If $\tilde{\Delta}_1 \neq \Delta_1 \cdot \gamma$, sample $\tilde{t} \xleftarrow{\$} \{0,1\}^\lambda$, send 0 and $\tilde{t}$ to $\mathcal{P}_1$, send `abort` to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$(④) and abort.

– Send $\boldsymbol{v}$, $\boldsymbol{M_v} := \tilde{\boldsymbol{v}}$ and $\boldsymbol{K_w} := \Delta_1 \cdot \boldsymbol{v} - \bar{\boldsymbol{v}}$ to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$(④).

Unlike the simulator for the malicious $\mathcal{P}_0$ we now need to argue that both the aborting behaviour and messages sent during the two calls to $\mathcal{F}_{\mathsf{EQ}}$ are indistinguishable from the real protocol execution. During the first call to $\mathcal{F}_{\mathsf{EQ}}$, the real protocol aborts if $a - c \neq d - b$. However, if the real protocol aborts the difference between the two values can be used to extract the secret Δ_0 if $\gamma \neq \tilde{\gamma}$. This is perfectly simulated by using the special abort command that reveals Δ_0 to the simulator. This ensures in the event that $\gamma \neq \tilde{\gamma}$ the simulator can abort, but also reveal Δ_0 to $\mathcal{A}$ by sending the expected message. The last exception is that when $\mathcal{A}$ guesses Δ_0 correctly, the equality check can succeed in the real protocol. This is also perfectly simulated by first issuing a guess to the ideal functionality in case $\tilde{\gamma} \neq \gamma$.

The evaluation of $C_1, \ldots, C_\ell$ is trivial to simulate by sending `success`, which is identical to the real world, as per assumption $\mathcal{P}_0$ behaves honestly.

The second call to $\mathcal{F}_{\mathsf{EQ}}$ aborts in the real world if $\boldsymbol{z} \neq (\bar{\boldsymbol{w}} \| \bar{e}) - (\boldsymbol{t} \| \tau)$. In particular, this happens when $\mathcal{A}$ commits to $\tilde{\Delta}_1 \neq \Delta_1 \cdot \gamma$, except if $\mathcal{A}$ can guess $(\boldsymbol{y} \| s)$, in which case they can adapt their input to $\mathcal{F}_{\mathsf{EQ}}$ as $\boldsymbol{z} = (\tilde{\Delta}_1 - \Delta_1 \cdot \gamma) \cdot (\boldsymbol{y} \| s) + (\boldsymbol{u} \| \sigma) - (\bar{\boldsymbol{v}} \| \bar{f}) + \Delta_1 \cdot (\boldsymbol{v} \| f)$. However, since s is chosen uniformly at random by $\mathcal{P}_0$, and unknown to the environment $\mathcal{Z}$, the simulator can just abort in the case the equality is wrong and send a random $\tilde{t} \xleftarrow{\$} \{0,1\}^\lambda$ to simulate the hash.

Lastly, the environment can check the consistency across the two worlds. As for the other simulator, the value produced by the adversary is correct except with negligible probability. Therefore, the two worlds are consistent and the environment cannot distinguish them except with negligible probability. □

C.4 Maliciously-Secure OPRF

Proof. First we consider correctness when both parties behave honestly. During the input-encoding steps, $\mathcal{P}_0$ computes $y^j = \mathsf{elt}_\alpha(\boldsymbol{x'}^j \| 1) \in \mathbb{F}_{2^n}$ where $\boldsymbol{x'}^j := \mathsf{decomp}(\boldsymbol{G} \cdot_3 \boldsymbol{x}^j) \in \mathbb{F}_2^{d+2(n'-d)}$ for each $j \in [m]$ as per the description in (5), and commits to this value using the authenticated VOLE functionality with sender key $k \in \mathbb{F}_{2^n}$, sender MAC key $\Delta_1^{(2)} \in \mathbb{F}_{2^n}$ and receiver MAC key $\Delta_0^{(2)}$, obtaining $\langle\!\langle k \cdot y^j \rangle\!\rangle^{(2^n)} \leftarrow (\llbracket w^j \rrbracket_{0,\mathsf{MAC}}^{(2^n)}, \llbracket v^j \rrbracket_{1,\mathsf{MAC}}^{(2^n)})$. In particular, $\llbracket w^j \rrbracket_{0,\mathsf{MAC}}^{(2^n)}$ defines a commitment of y^j under the sender's key k as $\llbracket y^j \rrbracket_{0,\mathsf{Key}}^{(2^n)} := \big((y^j, -w^j), (k, v^j)\big)$. The parties then use the local daBits obtained during preprocessing to convert these to modulo 3 VOLE commitments of $\llbracket \boldsymbol{y}^j \rrbracket_{0,\mathsf{Key}}^{(3)} := \llbracket \mathsf{vec}_\alpha(y^j) \rrbracket_{0,\mathsf{Key}}^{(3)}$ with sender key $\Delta_k^{(3)} \in \mathbb{F}_{3^{r^{(3)}}}$. Using the linear properties of VOLE commitments, the parties can now verify that

$$(\boldsymbol{H} \cdot \boldsymbol{G}_{\text{gadget}} \cdot \boldsymbol{y}^j_{[1,n-1]} \equiv 0 \mod 3) \quad \wedge \quad \boldsymbol{y}^j_n = 1,$$

which guarantees that the input is encoded correctly as described in Section 4.1.

Now using the authenticated secret sharing $\langle\!\langle k \cdot y^j \rangle\!\rangle^{(2^n)}$, the parties can convert these to an authenticated secret sharing of $\mathsf{vec}_\alpha(k \cdot y^j)$ over $\mathbb{F}_3$ using the doubly authenticated bits obtained during preprocessing, obtaining $\langle\!\langle \mathsf{vec}_\alpha(k \cdot y^j) \rangle\!\rangle^{(3)}$. Finally, they locally compute the matrix multiplication $\boldsymbol{B} \cdot_3 \mathsf{vec}_\alpha(k \cdot y^j)$ and reconstruct this value, which corresponds to $F_k(\boldsymbol{x}^j)$ as defined in (5).

First consider a maliciously corrupted receiver $\mathcal{P}_0$. The simulator behaves as follows:

– Receives $\Delta_0^{(2)} \in \mathbb{F}_{2^{r^{(2)}}}$, $\Delta_0^{(3)} \in \mathbb{F}_{3^{r^{(3)}}}$ from $\mathcal{A}$ upon the $(\mathsf{sid}_{\mathsf{MAC}}, \mathtt{init}, 0, q)$ calls to $\mathcal{F}_{\mathsf{mRand}}^{3,r^{(2)},r^{(3)}}$ for $q \in \{2,3\}$. Receives $\boldsymbol{b'}^j \in \{0,1\}^n$ and $\boldsymbol{M}_{\boldsymbol{b'}^j}^{(2)} \in \mathbb{F}_{2^n}^n$, $\boldsymbol{M}_{\boldsymbol{b'}^j}^{(3)} \in \mathbb{F}_{3^{r^{(3)}}}^n$, for each $j \in [\ell]$, from $\mathcal{A}$ after the $(\mathsf{sid}_{\mathsf{Key}}, \mathtt{laBits}, (n-1)\cdot\ell, 0)$ call to $\mathcal{F}_{\mathsf{mRand}}^{3,n,r^{(3)}}$. Receives $\boldsymbol{b}_{(2)}^{0,j} \in \mathbb{F}_2^n$, $\boldsymbol{b}_{(3)}^{0,j} \in \mathbb{F}_3^n$, $\boldsymbol{K}_{\boldsymbol{b}_{(2)}^{1,j}}^{(2)}, \boldsymbol{M}_{\boldsymbol{b}_{(2)}^{0,j}}^{(2)} \in \mathbb{F}_{2^{r^{(2)}}}^n$ and $\boldsymbol{K}_{\boldsymbol{b}_{(3)}^{1,j}}^{(3)}, \boldsymbol{M}_{\boldsymbol{b}_{(3)}^{0,j}}^{(3)} \in \mathbb{F}_{3^{r^{(3)}}}^n$ from $\mathcal{A}$, all for each $j \in [\ell]$, upon the $(\mathsf{sid}_{\mathsf{MAC}}, \mathtt{daBits}, n\cdot\ell)$ call to $\mathcal{F}_{\mathsf{mRand}}^{3,r^{(2)},r^{(3)}}$.
– During input-encoding, receive $y^j, w^j \in \mathbb{F}_{2^n}$ and $M_{w^j}^{(2^n)}, K_{v^j}^{(2^n)} \in \mathbb{F}_{2^n}$ from $\mathcal{A}$ upon the $\mathsf{aCommit}$ call. If $\mathcal{A}$ issues a special guess command $\mathtt{guess}^*$, the simulator sends $\mathtt{abort}$ to $\mathcal{F}_{\mathsf{OPRF}}$ and aborts.
– To simulate $\mathsf{BatchOpen}\left([\![c'^1]\!]_{0,\mathsf{Key}}^{(2^n)}, \ldots, [\![c'^\ell]\!]_{0,\mathsf{Key}}^{(2^n)}\right)$: Wait for $\mathcal{P}_0$ to send $c'^1, \ldots, c'^\ell \in \mathbb{F}_{2^n}$, sample $(\chi_1, \ldots, \chi_\ell) \xleftarrow{\$} \mathbb{F}_{2^{r^{(2)}}}$ and send these to $\mathcal{P}_0$. Wait for $\mathcal{P}_0$ to send $M \in \mathbb{F}_{2^{r^{(2)}}}$, check if $c'^j \stackrel{?}{=} y^j + \mathsf{elt}_\alpha(\boldsymbol{b'}^j)$ for all $j \in [\ell]$ and check if

$$M \stackrel{?}{=} \sum_{j=1}^{\ell} \chi_j \cdot \left(\mathsf{elt}_\alpha(\boldsymbol{M}_{\boldsymbol{b'}^j}^{(2)}) + w^j\right) =: M'.$$

If this does not hold, we are in one of the following two cases:
1. If c'^j verifies for all $j \in [\ell]$, but M does not, the simulator aborts;
2. If there exists $j \in [\ell]$ for which c'^j does not verify , the simulator computes

$$k' := (M' - M) \cdot \left(\sum_{j=1}^{\ell} \chi_j \cdot (c'^j - y^j - \mathsf{elt}_\alpha(\boldsymbol{b'}^j))\right)^{-1},$$

and sends $(\mathsf{sid}, \mathtt{guess}, k')$ to $\mathcal{F}_{\mathsf{OPRF}}$. If $\mathcal{F}_{\mathsf{OPRF}}$ returns $\mathtt{success}$, the simulator continues using c'^j. Otherwise, the simulator aborts.
– Put $\boldsymbol{y}^j := \mathsf{vec}_\alpha(c'^j) + \boldsymbol{b'}^j + \mathsf{vec}_\alpha(c'^j) \odot \boldsymbol{b'}^j$, $\boldsymbol{x'}^j := \boldsymbol{y}_{[1,n-1]}^j$ and $\boldsymbol{x}^j := \boldsymbol{x'}_{[1,d]}^j$. The simulator sends $(\mathsf{sid}, \mathtt{eval}, (\boldsymbol{x}^j)_{j\in[\ell]})$ to $\mathcal{F}_{\mathsf{OPRF}}$ and receives $(F_k(\boldsymbol{x}^j))_{j\in[\ell]}$. Moreover, the simulator puts $\boldsymbol{M}_{\boldsymbol{y}^j}^{(3)} := (\boldsymbol{M}_{\boldsymbol{b'}^j}^{(3)} + \mathsf{vec}_\alpha(c'^j) \odot \boldsymbol{M}_{\boldsymbol{b'}^j}^{(3)})$
– To simulate $\mathsf{BatchVer}\left([\![\boldsymbol{z}^1]\!]_{0,\mathsf{Key}}^{(3)}, \ldots, [\![\boldsymbol{z}^\ell]\!]_{0,\mathsf{Key}}^{(3)}, [\![z^{\ell+1}]\!]_{0,\mathsf{Key}}^{(3)}, \ldots, [\![z^{2\ell}]\!]_{0,\mathsf{Key}}^{(3)}\right)$: The simulator samples $(\chi_1, \ldots, \chi_{(n'-d+1)\cdot\ell}) \xleftarrow{\$} \mathbb{F}_{3^{r^{(3)}}}^{(n'-d+1)\cdot\ell}$ and sends these to $\mathcal{P}_0$. Wait for $\mathcal{P}_0$ to send $M \in \mathbb{F}_{3^{r^{(3)}}}$, check, for all $j \in [\ell]$, if

$$(\boldsymbol{z^j} := \boldsymbol{H} \cdot \boldsymbol{G}_{\mathrm{gadget}} \cdot \boldsymbol{x'}^j \stackrel{?}{=} \boldsymbol{0}) \quad \wedge \quad \boldsymbol{y}_n^j = 1$$

and check if

$$M \stackrel{?}{=} \sum_{j=1}^{\ell} \sum_{i=1}^{n'-d} \chi_{i+(j-1)\cdot(n'-d)} \cdot M_{z_i^j}^{(3)} + \sum_{\nu=1}^{\ell} \chi_{(n'-d)\ell+\nu} \cdot M_{z^{\ell+\nu}}^{(3)},$$

where $\boldsymbol{M}_{\boldsymbol{z}^j}^{(3)} := \boldsymbol{H} \cdot \boldsymbol{G}_{\mathrm{gadget}} \cdot (\boldsymbol{M}_{\boldsymbol{y}^j}^{(3)})_{[1,n-1]}$ and $M_{z^{\ell+j}}^{(3)} := (\boldsymbol{M}_{\boldsymbol{y}^j}^{(3)})_n$ for $j \in [\ell]$. The simulator aborts if any of the checks do not hold.
– To simulate $\mathsf{BatchOpen}\left((\langle\!\langle c^1 \rangle\!\rangle^{(2^n)}, \ldots, \langle\!\langle c^\ell \rangle\!\rangle^{(2^n)}), 1\right)$, wait for $\mathcal{P}_0$ to send $c_0^1, \ldots, c_0^\ell \in \mathbb{F}_{2^n}$, sample $(\chi_1, \ldots, \chi_\ell) \xleftarrow{\$} \mathbb{F}_{2^n}^\ell$ and send to $\mathcal{P}_0$. Wait for $\mathcal{P}_0$ to send $M \in \mathbb{F}_{2^n}$ and check if $c_0^j \stackrel{?}{=} w^j + b_0^j$ for

all $j \in [\ell]$ and

$$M \stackrel{?}{=} \sum_{j=1}^{\ell} \chi_j \cdot \left(M^{(2^n)}_{w^j} + M^{(2^n)}_{b_0^j} \right),$$

where $b_0^j = \mathsf{elt}_\alpha(\boldsymbol{b}^{0,j}_{(2)})$ and $M^{(2^n)}_{b_0^j} := \mathsf{elt}_\alpha(\boldsymbol{M}^{(2)}_{\boldsymbol{b}^{0,j}_{(2)}})$, and abort if any of the checks do not hold.

– To simulate $\mathsf{BatchOpen}\left((\langle\!\langle c^1 \rangle\!\rangle^{(2^n)}, \ldots, \langle\!\langle c^\ell \rangle\!\rangle^{(2^n)}), 0\right)$, sample $c_1^1, \ldots, c_1^\ell \xleftarrow{\$} \mathbb{F}_2^n$ and send these to $\mathcal{P}_0$. Wait for $\mathcal{P}_0$ to send $(\chi_1, \ldots, \chi_\ell) \in \mathbb{F}^\ell_{2^n}$ and return

$$M := \sum_{j=1}^{\ell} \chi_j \cdot \left(\Delta_0^{(2)} \cdot c_1^j + K^{(2^n)}_{v^j} + K^{(2^n)}_{b_1^j} \right)$$

to $\mathcal{P}_0$, where $K^{(2^n)}_{b_1^j} := \mathsf{elt}_\alpha(\boldsymbol{K}_{\boldsymbol{b}^{1,j}_{(2)}})$.

– Finally, to simulate $\mathsf{BatchOpen}\left((\langle\!\langle F_k(\boldsymbol{x}^1) \rangle\!\rangle^{(3)}, \ldots, \langle\!\langle F_k(\boldsymbol{x}^\ell) \rangle\!\rangle^{(3)}), 0\right)$, the simulator puts $c^j := c_0^j + c_1^j$ and $\boldsymbol{f}_0^j := \boldsymbol{B} \cdot_3 \left(\mathsf{vec}_\alpha(c^j) + \boldsymbol{b}^{0,j}_{(3)} + \mathsf{vec}_\alpha(c^j) \odot \boldsymbol{b}^{0,j}_{(3)} \right)$, and sends $\boldsymbol{f}_1^j := F_k(\boldsymbol{x}^j) - \boldsymbol{f}_0^j \bmod 3$ to $\mathcal{P}_0$ for each $j \in [\ell]$. The simulator computes

$$\boldsymbol{M}^{(3)}_{\boldsymbol{f}_1^j} := \Delta_0^{(3)} \cdot \boldsymbol{f}_1^j + \boldsymbol{B} \cdot_3 \left(\boldsymbol{K}^{(3)}_{\boldsymbol{b}^{1,j}_{(3)}} + \mathsf{vec}_\alpha(c^j) \odot \boldsymbol{K}^{(3)}_{\boldsymbol{b}^{1,j}_{(3)}} \right),$$

waits for $\mathcal{P}_0$ to send $(\chi_1, \ldots, \chi_{t \cdot \ell}) \in \mathbb{F}^{t \cdot \ell}_{3^{r^{(3)}}}$ and returns

$$M := \sum_{j=1}^{\ell} \sum_{i=1}^{t} \chi_{i+(j-1)\cdot t} \cdot M^{(3)}_{f^j_{1,i}}.$$

We will now argue that the environment $\mathcal{Z}$ can not distinguish between the following joint distributions: (1) The transcript of $\mathcal{P}_0$ during a real execution of the protocol with an honest $\mathcal{P}_1$, together with the protocol output's of the honest $\mathcal{P}_1$; (2) The transcript of $\mathcal{P}_0$ during a simulated execution described above, together with both parties' outputs from the ideal functionality $\mathcal{F}_{\mathsf{OPRF}}$. In both cases the environment $\mathcal{Z}$ controls $\mathcal{P}_0$ and $\mathcal{A}$, can additionally choose the the honest $\mathcal{P}_1$'s input, and receives the output of both parties. First note that the simulator perfectly simulates all interactions with the ideal functionality $\mathcal{F}^{3,r^{(2)},r^{(3)}}_{\mathsf{mRand}}$. Here we want to highlight that the environment can not distinguish between $\mathcal{P}_0$ succeeding or failing upon a special guess call $\mathtt{guess}^*$ to $\mathcal{F}^{3,r^{(2)},r^{(3)}}_{\mathsf{mRand}}$(④), since a correct guess (a, b, c) needs to satisfy $\Delta_1^{(2)} \stackrel{?}{=} (a + b \cdot k)^{-1} \cdot c$. However, since it is not an output of the ideal $\mathcal{F}_{\mathsf{OPRF}}$ functionality, $\Delta_1^{(2)}$ is uniformly random in $\mathbb{F}_{2^{r^{(2)}}}$ from the point of view of $\mathcal{Z}$. Since $|\mathbb{F}_{2^{r^{(2)}}}| \geq 2^\kappa$, the probability of $\mathcal{Z}$ (or $\mathcal{P}_0$) guessing $\Delta_1^{(2)}$ is negligible. It remains to argue that the simulated $\mathsf{BatchOpen}$ and $\mathsf{BatchVer}$ are indistinguishable from the real protocol execution, and that the simulated transcript is consistent with the ideal $\mathcal{F}_{\mathsf{OPRF}}$ functionality outputs.

For the batch opening of $[\![c'^1]\!]^{(2^n)}_{0,\mathsf{Key}}, \ldots, [\![c'^\ell]\!]^{(2^n)}_{0,\mathsf{Key}}$, the simulator does of course not know the honest party $\mathcal{P}_1$'s key k, so can check correctness of the opening in two ways: (1) By checking if c'^j and M are consistent with the earlier committed values they obtained from $\mathcal{P}_0$; (2) If these do not match, the opening check should fail unless $\mathcal{P}_0$ managed to guess the key k correctly. So the simulator can forward a key-guess k' to $\mathcal{F}_{\mathsf{OPRF}}$ and continue if and only if the guess is correct. The simulation of the batch verification of $[\![\boldsymbol{z}^1]\!]^{(3)}_{0,\mathsf{Key}}, \ldots, [\![\boldsymbol{z}^\ell]\!]^{(3)}_{0,\mathsf{Key}}, [\![z^{\ell+1}]\!]^{(3)}_{0,\mathsf{Key}}, \ldots, [\![z^{2\ell}]\!]^{(3)}_{0,\mathsf{Key}}$ and the

batch opening of $\langle\!\langle c^1 \rangle\!\rangle^{(2^n)}, \ldots, \langle\!\langle c^\ell \rangle\!\rangle^{(2^n)}$ to $\mathcal{P}_1$ proceeds similarly, but since the environment has no knowledge about the honest $\mathcal{P}_1$'s MAC keys $\Delta_k^{(3)} \in \mathbb{F}_{3^{r^{(3)}}}$ and $\Delta_1^{(2)} \in \mathbb{F}_{2^{r^{(2)}}}$, the simulator can simply abort whenever the values $\mathcal{P}_0$ sends do not agree with the values they committed to before, and the probability of guessing $\Delta_k^{(3)}$ or $\Delta_1^{(2)}$ is negligible since $r^{(2)} \geq \kappa$ and $r^{(3)} \geq \kappa / \log_2 3$.

Now for the batch openings in the other direction. During the opening of $\langle\!\langle c^1 \rangle\!\rangle^{(2^n)}, \ldots, \langle\!\langle c^\ell \rangle\!\rangle^{(2^n)}$ to $\mathcal{P}_0$, the simulated $c_1^1, \ldots, c_1^\ell \xleftarrow{\$} \mathbb{F}_{2^n}$ are indistinguishable from the real protocol execution since the $\boldsymbol{b}_{(2)}^{1,j} \in \mathbb{F}_2^n$, and therefore $b_1^j = \mathsf{elt}_\alpha(\boldsymbol{b}_{(2)}^{1,j}) \in \mathbb{F}_{2^n}$, are uniformly random by definition of $\mathcal{F}_{\mathsf{mRand}}^{3,r^{(2)},r^{(3)}}$ (②), and the environment has no information about these values. Moreover, the linear combination of MACs M is consistent with the real protocol, since, implicitly defining b_1^j through the equation $c_1^j = v^j + b_1^j$, where $v^j = k \cdot y^j - w^j$, and writing out the corresponding MACs as $M_{v^j}^{(2^n)} = \Delta_0^{(2)} \cdot v^j + K_{v^j}^{(2^n)}$ and $M_{b_1^j}^{(2^n)} = \Delta_0^{(2)} \cdot b_1^j + K_{b_1^j}^{(2^n)}$, it holds that

$$\Delta_0^{(2)} \cdot c_1^j + K_{v^j}^{(2^n)} + K_{b_1^j}^{(2^n)} = M_{v^j}^{(2^n)} + M_{b_1^j}^{(2^n)}.$$

For the opening of $\langle\!\langle F_k(\boldsymbol{x}^1) \rangle\!\rangle^{(3)}, \ldots, \langle\!\langle F_k(\boldsymbol{x}^\ell) \rangle\!\rangle^{(3)}$, the simulated shares $\boldsymbol{f}_1^j := F_k(\boldsymbol{x}^j) - \boldsymbol{f}_0^j \bmod 3$ are picked to enforce consistency with the ideal functionality outputs $F_k(\boldsymbol{x}^j)$ on $\mathcal{P}_0$'s inputs $\boldsymbol{x}^j$ extracted after the input-encoding proof, and the $\boldsymbol{f}_0^j$ are computed identically to the share $\mathcal{P}_0$ should compute to get the correct output. Since we defined b_1^j through the equation $c_1^j = v^j + b_1^j$ and checked that $c_0^j = w^j + b_0^j$, it holds that

$$\begin{aligned} c^j &= c_0^j + c_1^j \\ &= w^j + v^j + b_0^j + b_1^j \\ &= k \cdot y^j + b^j \mod 2. \end{aligned}$$

So consistency between the real and the simulated shares follows, as

$$\begin{aligned} \boldsymbol{f}_1^j &:= F_k(\boldsymbol{x}^j) - \boldsymbol{f}_0^j && (8) \\ &= \boldsymbol{B} \cdot_3 \mathsf{vec}_\alpha(k \cdot y^j) - \boldsymbol{B} \cdot_3 \left(\mathsf{vec}_\alpha(c^j) + \boldsymbol{b}_{(3)}^{0,j} + \mathsf{vec}_\alpha(c^j) \odot \boldsymbol{b}_{(3)}^{0,j} \right) && (9) \\ &= \boldsymbol{B} \cdot_3 \left(\boldsymbol{b}^j + \mathsf{vec}_\alpha(c^j) + \boldsymbol{b}^j \odot \mathsf{vec}_\alpha(c^j) \right) - \boldsymbol{B} \cdot_3 \left(\mathsf{vec}_\alpha(c^j) + \boldsymbol{b}_{(3)}^{0,j} + \mathsf{vec}_\alpha(c^j) \odot \boldsymbol{b}_{(3)}^{0,j} \right) && (10) \\ &= \boldsymbol{B} \cdot_3 \left(\boldsymbol{b}_{(3)}^{1,j} + \mathsf{vec}_\alpha(c_j) \odot \boldsymbol{b}_{(3)}^{1,j} \right). && (11) \end{aligned}$$

The consistency of the linear combination of MACs sent by the simulator can now be seen similarly as before.

We construct the simulator for a maliciously corrupted sender $\mathcal{P}_1$ as follows:

- Receive $\Delta_1^{(2)}, k \in \mathbb{F}_{2^n}$ and $\Delta_1^{(3)}, \Delta_k^{(3)} \in \mathbb{F}_{3^{r^{(3)}}}$ from $\mathcal{A}$ upon the $(\mathsf{sid}_{\mathsf{MAC}}, \mathtt{init}, 1, q)$ and $(\mathsf{sid}_{\mathsf{Key}}, \mathtt{init}, 1, q)$ calls to $\mathcal{F}_{\mathsf{mRand}}^{3,r^{(2)},r^{(3)}}$ for $q \in \{2, 3\}$. Receive $\boldsymbol{K}_{\boldsymbol{b'}^j}^{(2)} \in \mathbb{F}_{2^{r^{(2)}}}^n$, $\boldsymbol{K}_{\boldsymbol{b'}^j}^{(3)} \in \mathbb{F}_{3^{r^{(3)}}}^n$ from $\mathcal{A}$ upon the $(\mathsf{sid}_{\mathsf{MAC}}, \mathtt{laBits}, n \cdot \ell, 1)$ call and $\boldsymbol{b}_{(2)}^{1,j} \in \mathbb{F}_2^n$, $\boldsymbol{b}_{(3)}^{1,j} \in \mathbb{F}_3^n$, $\boldsymbol{K}_{\boldsymbol{b}_{(2)}^{0,j}}^{(2)}, \boldsymbol{M}_{\boldsymbol{b}_{(2)}^{1,j}}^{(2)} \in \mathbb{F}_{2^{r^{(2)}}}^n$ and $\boldsymbol{K}_{\boldsymbol{b}_{(3)}^{0,j}}^{(3)}, \boldsymbol{M}_{\boldsymbol{b}_{(3)}^{1,j}}^{(3)} \in \mathbb{F}_{3^{r^{(3)}}}^n$ from $\mathcal{A}$, all for each $j \in [\ell]$, upon the $(\mathsf{sid}_{\mathsf{MAC}}, \mathtt{daBits}, n \cdot \ell)$ call to $\mathcal{F}_{\mathsf{mRand}}^{3,r^{(2)},r^{(3)}}$.
- Receive $v^j, M_{v^j}^{(2^n)}, K_{w^j}^{(2^n)} \in \mathbb{F}_{2^n}$ from $\mathcal{A}$ for each $j \in [\ell]$ during the $\mathsf{aCommit}$ call to $\mathcal{F}_{\mathsf{mRand}}^{3,r^{(2)},r^{(3)}}$. If $\mathcal{A}$ issues a special abort call $\mathtt{abort}^*$, the simulator samples $\Delta_0^{(2)} \xleftarrow{\$} \mathbb{F}_{2^n}$, returns it to $\mathcal{P}_1$, sends $\mathtt{abort}$ to $\mathcal{F}_{\mathsf{OPRF}}$ and aborts.

– To simulate $\mathsf{BatchOpen}\left([\![c'^1]\!]^{(2^n)}_{0,\mathsf{Key}}, \ldots, [\![c'^\ell]\!]^{(2^n)}_{0,\mathsf{Key}}\right)$, the simulator samples $c'^1, \ldots, c'^\ell \xleftarrow{\$} \mathbb{F}_{2^n}$ and sends these to $\mathcal{P}_1$. The simulator then waits for $\mathcal{P}_1$ to send $(\chi_1, \ldots, \chi_\ell) \in \mathbb{F}_{2^n}^\ell$ and responds with

$$M := \sum_{j=1}^{\ell} \chi_j \cdot (k \cdot c'^j + K^{(2^n)}_{b'^j} - v^j),$$

where $K^{(2^n)}_{b'^j} := \mathsf{elt}_\alpha(\boldsymbol{K}^{(2)}_{\boldsymbol{b}'^j})$.

– To simulate $\mathsf{BatchVer}\left([\![\boldsymbol{z}^1]\!]^{(3)}_{0,\mathsf{Key}}, \ldots, [\![\boldsymbol{z}^\ell]\!]^{(3)}_{0,\mathsf{Key}}, [\![z^{\ell+1}]\!]^{(3)}_{0,\mathsf{Key}}, \ldots, [\![z^{2\ell}]\!]^{(3)}_{0,\mathsf{Key}}\right)$, the simulator computes $\boldsymbol{K}^{(3)}_{\boldsymbol{y}^j} := \boldsymbol{K}^{(3)}_{\boldsymbol{b}'^j} + \mathsf{vec}_\alpha(c'^j) \odot \boldsymbol{K}^{(3)}_{\boldsymbol{b}'^j} - \Delta^{(3)}_k \cdot \mathsf{vec}_\alpha(c'^j)$ and $\boldsymbol{K}^{(3)}_{\boldsymbol{z}^j} := \boldsymbol{H} \cdot \boldsymbol{G}_{\text{gadget}} \cdot \boldsymbol{K}^{(3)}_{\boldsymbol{y}^j}$ for $j \in [\ell]$, $K^{(3)}_{z^{\ell+\nu}} := K^{(3)}_{y^\nu_n} - \Delta^{(3)}_k$ for $\nu \in [\ell]$. Wait for $\mathcal{P}_1$ to send $(\chi_1, \ldots, \chi_{(n'-d+1)\cdot\ell}) \in \mathbb{F}_{3^{r^{(3)}}}^{(n'-d+1)\cdot\ell}$ and respond with

$$M := \sum_{j=1}^{\ell} \sum_{i=1}^{n'-d} \chi_{i+(j-1)\cdot(n'-d)} \cdot K^{(3)}_{z^j_i} + \sum_{\nu=1}^{\ell} \chi_{(n'-d)\cdot\ell+\nu} \cdot K^{(3)}_{z^{\ell+\nu}}.$$

– To simulate $\mathsf{BatchOpen}\left((\langle\!\langle c^1 \rangle\!\rangle^{(2^n)}, \ldots, \langle\!\langle c^\ell \rangle\!\rangle^{(2^n)}), \{0,1\}\right)$:
  - The opening to $\mathcal{P}_0$ can be simulated as follows. Wait for $\mathcal{P}_1$ to send $c^1_1, \ldots, c^\ell_1 \in \mathbb{F}_{2^n}$. Sample $(\chi_1, \ldots, \chi_\ell) \xleftarrow{\$} \mathbb{F}_{2^n}^\ell$ and send these to $\mathcal{P}_1$. Wait for $\mathcal{P}_1$ to send M and check if $M \stackrel{?}{=} \sum_{j=1}^{\ell} \chi_j \cdot \left(M^{(2^n)}_{v^j} + M^{(2^n)}_{b^j_1}\right)$ and $c^j_1 \stackrel{?}{=} v^j + b^j_1$, where $b^j_1 := \mathsf{elt}_\alpha(\boldsymbol{b}^{1,j}_{(2)})$ and $M^{(2^n)}_{b^j_1} := \mathsf{elt}_\alpha(\boldsymbol{M}^{(2)}_{\boldsymbol{b}^{1,j}_{(2)}})$, and abort if any of these do not hold.
  - The opening to $\mathcal{P}_1$ can be simulated as follows. Sample $c^1_0, \ldots, c^\ell_0 \xleftarrow{\$} \mathbb{F}_{2^n}$ and send these to $\mathcal{P}_1$. Wait for $\mathcal{P}_1$ to send $(\chi_1, \ldots, \chi_\ell) \in \mathbb{F}_{2^n}$. Send $M := \sum_{j=1}^{\ell} \chi_j \cdot \left(\Delta^{(2)}_1 \cdot c^j_0 + K^{(2^n)}_{w^j} + K^{(2^n)}_{b^j_0}\right)$ to $\mathcal{P}_1$, where $K^{(2^n)}_{b^j_0} := \mathsf{elt}_\alpha(\boldsymbol{K}^{(2)}_{\boldsymbol{b}^{0,j}_{(2)}})$.

– To simulate $\mathsf{BatchOpen}\left(\langle\!\langle F_k(\boldsymbol{x}^1) \rangle\!\rangle^{(3)}, \ldots, \langle\!\langle F_k(\boldsymbol{x}^\ell) \rangle\!\rangle^{(3)}), 0\right)$, wait for $\mathcal{P}_1$ to send $\boldsymbol{f}^j_1 \in \mathbb{F}^t_3$ for each $j \in [t]$ and send $(\chi_1, \ldots, \chi_{t\cdot\ell}) \xleftarrow{\$} \mathbb{F}_{3^{r^{(3)}}}^{t\cdot\ell}$ to $\mathcal{P}_1$. Wait for $\mathcal{P}_1$ to send M and check if $\boldsymbol{f}^j_1 \stackrel{?}{=} \boldsymbol{B} \cdot_3 \left(\boldsymbol{b}^{1,j}_{(3)} + \mathsf{vec}_\alpha(c^j) \odot \boldsymbol{b}^{1,j}_{(3)}\right)$ and $M \stackrel{?}{=} \sum_{j=1}^{\ell} \sum_{i=1}^{t} \chi_{i+(j-1)\cdot t} \cdot M^{(3)}_{f^j_{1,i}}$, where $c^j := c^j_0 + c^j_1$ and $\boldsymbol{M}_{\boldsymbol{f}^j_1} := \boldsymbol{B} \cdot_3 \left(\boldsymbol{M}_{\boldsymbol{b}^{1,j}_{(3)}} + \mathsf{vec}_\alpha(c^j) \odot \boldsymbol{M}_{\boldsymbol{b}^{1,j}_{(3)}}\right)$, for all $j \in [\ell]$, and abort if any of these do not hold.

To argue indistinguishability between the real and simulated distribution, we again first note that the simulator perfectly simulates the interactions with the ideal $\mathcal{F}^{3,r^{(2)},r^{(3)}}_{\mathsf{mRand}}$ functionality. Here the only notable difference is that the simulator returns a uniformly random $\Delta^{(2)}_0 \xleftarrow{\$} \mathbb{F}_{2^n}$ upon the adversary issuing a special abort call $\texttt{abort}^*$, but the environment is not able to distinguish this value from the MAC key used by the honest client $\mathcal{P}_0$ since it is not part of the ideal $\mathcal{F}_{\mathsf{OPRF}}$ output, and thus remains uniformly random in $\mathbb{F}_{2^n}$ from their point of view. It remains to argue the indistinguishability of the simulated $\mathsf{BatchOpen}$ and $\mathsf{BatchVer}$ executions, as well as the consistency of the simulated transcript with the ideal $\mathcal{F}_{\mathsf{OPRF}}$ outputs.

For batch opening of the $[\![c'^1]\!]^{(2^n)}_{0,\mathsf{Key}}, \ldots, [\![c'^\ell]\!]^{(2^n)}_{0,\mathsf{Key}}$, the simulated $c'^1, \ldots, c'^\ell \xleftarrow{\$} \mathbb{F}_{2^n}$ are indistinguishable from the real protocol $c'^j := y^j + b'^j$ since the local doubly-authenticated bits $\boldsymbol{b}'^j$ are uniformly random in $\mathbb{F}_2^n$ by definition of $\mathcal{F}^{3,r^{(2)},r^{(3)}}_{\mathsf{mRand}}$ (①) and therefore the $b'^j = \mathsf{elt}_\alpha(\boldsymbol{b}'^j)$ are uniformly random in $\mathbb{F}_{2^n}$ as well, where additionally the environment has no information about these values.

The simulated MAC is consistent with what the honest receiver computes since, implicitly defining b'^j by the equation $c'^j = y^j + b'^j$, writing $M^{(2^n)}_{b'_j} = k \cdot b'^j + K^{(2^n)}_{b'_j}$, we see that $k \cdot c'^j + K^{(2^n)}_{b'_j} - v^j = w^j + M^{(2^n)}_{b'_j}$. Now for the batch verification of $[\![\boldsymbol{z}^1]\!]^{(3)}_{0,\mathsf{Key}}, \ldots, [\![\boldsymbol{z}^\ell]\!]^{(3)}_{0,\mathsf{Key}}, [\![z^{\ell+1}]\!]^{(3)}_{0,\mathsf{Key}}, \ldots, [\![z^{2\ell}]\!]^{(3)}_{0,\mathsf{Key}}$, it is straightforward to see that the simulator computes the MACs consistently with what the honest receiver $\mathcal{P}_0$ computes in the real protocol execution since all the commitments should open to zero and the simulator knows $\mathcal{P}_1$'s MAC keys.

The indistinguishability of the simulated openings of $\langle\!\langle c^1 \rangle\!\rangle^{(2^n)}, \ldots, \langle\!\langle c^\ell \rangle\!\rangle^{(2^n)}$ in both directions can be argued analogously to the proof for a malicious $\mathcal{P}_0$ since this part of the protocol is symmetric with respect to the role of the two parties. Finally, for the batch opening of $\langle\!\langle F_k(\boldsymbol{x}^1) \rangle\!\rangle^{(3)}, \ldots, \langle\!\langle F_k(\boldsymbol{x}^\ell) \rangle\!\rangle^{(3)}$ to $\mathcal{P}_0$, the simulator can check if the sender's shares $\boldsymbol{f}^j_1$ are computed according to the values the sender committed to before, and similarly check the linear combination of MACs M. If these do not match, it might be the case that $\mathcal{P}_1$ managed to guess the honest $\mathcal{P}_0$'s MAC key $\Delta^{(3)}_0$, but since $\mathcal{P}_0$ sampled $\Delta^{(3)}_0$ uniformly in $\mathbb{F}_{3^{r^{(3)}}}$, where $r^{(3)} \geq \kappa / \log_2 3$ and the environment does not receive any information about this value, this only happens with negligible probability and the environment can not distinguish. It remains to show that the simulated transcript is consistent with the ideal functionality outputs $F_k(\boldsymbol{x}^j)$. Implicitly defining $w^j = k \cdot y^j - v^j$ and $b^j_0 = c^j_0 - w^j$, consistency follows by following the equations (8)–(11) in the opposite direction. □

C.5 Half-Half-Authenticated Bits

Proof. First, we show that the protocol has correctness when both parties behave honestly. The parties sample $\boldsymbol{b}^j \xleftarrow{\$} \mathbb{F}_2^\ell$ for $j \in \{0, 1\}$, which trivially forms a secret sharing of $\boldsymbol{b} \bmod 2$. As $\mathcal{P}_0$ commits to their $\boldsymbol{b}^0$ to get the MAC $\boldsymbol{M}$ and $\mathcal{P}_1$ the key $(\Delta^{(2)}, \boldsymbol{K})$ the secret sharing of $\boldsymbol{b}$ is half authenticated on the $\mathcal{P}_0$ side. Next, we show that $\tilde{\boldsymbol{b}}^0 + \tilde{\boldsymbol{b}}^1 = \boldsymbol{b} \bmod p$. We show this for a single entry:

$$
\begin{aligned}
\tilde{b}^0 + \tilde{b}^1 &= m_{b^0} + b^0 \cdot \delta + b^0 + b^1 - m_0 \\
&= m_{b^0} + b^0 \cdot (m_0 - m_1 - 2b^1) + b^0 + b^1 - m_0 \\
&= m_{b^0} + b^0 \cdot m_0 - b^0 \cdot m_1 - 2b^0 \cdot b^1 + b^0 + b^1 - m_0 \\
&= \begin{cases}
m_0 - m_0 = 0 & \text{if } \; b^0 = 0, b^1 = 0 \\
m_0 + b^1 - m_0 = 1 & \text{if } \; b^0 = 0, b^1 = 1 \\
m_1 + m_0 - m_1 + b^0 - m_0 = 1 & \text{if } \; b^0 = 1, b^1 = 0 \\
m_1 + m_0 - m_1 - 2b^0 \cdot b^1 + b^0 + b^1 - m_0 = 0 & \text{if } \; b^0 = 1, b^1 = 1
\end{cases} \\
&= b \mod p
\end{aligned}
$$

From this, the parties have a secret sharing of $\boldsymbol{b}$ over $\mathbb{F}_2$ and over $\mathbb{F}_p$.

Now consider $\mathcal{P}_0$ to be maliciously corrupted by $\mathcal{A}$. We define the simulator $\mathcal{S}$ as follows:

- Receives $(\boldsymbol{b}^0, \boldsymbol{M})$ from $\mathcal{A}$ upon the $\mathsf{Commit}(\boldsymbol{b}^0, 2, r^{(2)}, \Delta^{(2)})$ call to $\mathcal{F}^{2,r^{(2)}}_{\mathsf{sVOLE}}$.
- Sample $\boldsymbol{\delta} \xleftarrow{\$} \mathbb{F}_p^\ell$ and $\tilde{\boldsymbol{b}}^0 \xleftarrow{\$} \mathbb{F}_p^\ell$. For $j \in [\ell]$: If $b^0_j = 0$ program $\mathsf{H}(j, M_j)$ to output $\tilde{b}^0_j$ else if $b^0_j = 1$ program $\mathsf{H}(j, M_j)$ to output $\tilde{b}^0_j - \delta_j - 1$.
- Send $\boldsymbol{\delta}$ to $\mathcal{A}$.
- Sends $\boldsymbol{b}^0$, $\boldsymbol{M}_{b^0} := \boldsymbol{M}$, and $\tilde{\boldsymbol{b}}^0$ to $\mathcal{F}^{p,r^{(2)},r^{(p)}}_{\mathsf{mRand}}$ (⑤).

We now argue for indistinguishability between the real world and the ideal world. From the point of view of $\mathcal{A}$ the ideal and the real world are identically distributed: $\boldsymbol{\delta}$ is uniformly random in the real world, since $m_{b^0_j \oplus 1, j}$ is uniform . $\boldsymbol{\delta}$ has the same distribution in the ideal world as it is sampled

uniformly. When the protocol outputs, $\mathcal{Z}$ learn the output of both parties, and hence the output of the honest party. The environment can now check if the values from the simulator are consistent with those in the real world.

For a given index j, the environment $\mathcal{Z}$ can check that both δ_j and $\tilde{b}_j^1$ are consistent between the two worlds. To check both values, $\mathcal{Z}$ needs to query both $\mathsf{H}(j, K_j)$ and $\mathsf{H}(j, \Delta^{(2)} + K_j)$. From this $\mathcal{S}$ can learn $\Delta^{(2)}$, by doing the following: Let the i'th query to H be (j_i, Q_i), $\mathcal{S}$ take all previous queries (j_l, Q_l) for $l < i$ and compute $\Delta' = Q_i - Q_l$ and send $(\mathsf{sid}, \mathsf{guess}, \Delta', 2)$ to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$ (⑤) if it returns $(\mathtt{success}, (\boldsymbol{b}^1, \tilde{\boldsymbol{b}^1},))$, $\mathcal{S}$ learns both $\Delta^{(2)}$ and the output of the honest party. We show how $\mathcal{S}$ can program the random oracle to ensure consistency between the real and the ideal world, by case distinction on b_j^0.

- Case $b_j^0 = 0$: $\mathcal{Z}$ can learn $\mathsf{H}(j, M_j) = m_{0,j}$ without $\mathcal{S}$ learning $\Delta^{(2)}$. Thus, $\mathcal{Z}$ can verify the consistency of $\tilde{b}_j^1$. By the programming of H during the simulation of the protocol we have $\mathsf{H}(j, M_j) = m_{0,j} = \tilde{b}_j^0$, which by the assumption of $b_j^0 = 0$ proves the consistency since

$$\begin{aligned}\tilde{b}_j^1 &= b_j - \tilde{b}_j^0 \mod p \\ &= b_j^1 - m_{0,j} \mod p.\end{aligned}$$

  $\mathcal{Z}$ can also check δ_j, however, to do this a query for $\mathsf{H}(\Delta^{(2)} + K_j)$ must be made, revealing $\Delta^{(2)}$ and the honest party's output to $\mathcal{S}$. Letting $\mathcal{S}$ program $\mathsf{H}(j, \Delta^{(2)} + K_j) = m_{1,j}$ to $m_{0,j} - \delta_j - 2 \cdot b_j^1$ the simulated value for δ_j is consistent.
- Case $b_j^0 = 1$: $\mathcal{Z}$ can learn $\mathsf{H}(j, M_j) = m_{1,j}$ without $\mathcal{S}$ learning $\Delta^{(2)}$. However, with this $\mathcal{Z}$ is unable to check any equations without querying $\mathsf{H}(j, K_j)$, which reveals $\Delta^{(2)}$ and the honest party's output to $\mathcal{S}$. Letting $\mathcal{S}$ program $\mathsf{H}(j, K_j) = m_{0,j}$ to $b_j^1 - \tilde{b}_j^1$ the simulated values are consistent. The consistency of $\tilde{b}_j^1$ follows directly from the programming. By using the programming of $\mathsf{H}(j, M_j)$ during the simulation of the protocol, we have $\delta_j = \tilde{b}_j^0 - \mathsf{H}(j, M_j) - 1$, which is consistent with the real world as follows:

$$\begin{aligned}\delta_j &= \tilde{b}_j^0 - \mathsf{H}(j, M_j) - 1 \mod p \\ &= \tilde{b}_j^0 - m_{1,j} - b_j - b_j^1 \mod p \\ &= b_j^1 - \tilde{b}_j^1 - m_{1,j} - 2 \cdot b_j^1 \mod p \\ &= m_{0,j} - m_{1,j} - 2 \cdot b_j^1 \mod p,\end{aligned}$$

  since by definition $\tilde{b}_j^0 + \tilde{b}_j^1 = b_j$ and by assumption $b_j = 1 - b_j^1 \bmod p$.

As $\mathcal{S}$ knows the value of b_j^0 and learns the index j on the query to H, $\mathcal{S}$ can select the correct case and program the oracle accordingly.

Now consider $\mathcal{P}_1$ to be semi-honestly corrupt by $\mathcal{A}$. We define the simulator $\mathcal{S}$ as follows:

- Receives $(\Delta^{(2)}, \boldsymbol{K})$ from $\mathcal{A}$ upon the $\mathsf{Commit}(\boldsymbol{b}^0, 2, r^{(2)}, \Delta^{(2)})$ call to $\mathcal{F}_{\mathsf{sVOLE}}^{2,r^{(2)}}$.
- Receive $\boldsymbol{\delta}$ from $\mathcal{A}$.
- Extract b_j^1 for all $j \in [\ell]$ as $b_j^1 = 2 \cdot (m_{0,j} - m_{1,j} - \delta_j)$, where $m_{0,j} = \mathsf{H}(j, K_j)$ and $m_{1,j} = \mathsf{H}(j, \Delta + K_j)$.
- Let $\tilde{b}_j^1 = b_j^1 - \mathsf{H}(K_j)$ for $j \in [\ell]$.
- Send $\boldsymbol{b}^1, \tilde{\boldsymbol{b}}^1$ to $\mathcal{F}_{\mathsf{mRand}}^{p,r^{(2)},r^{(p)}}$ (⑤)

The ideal and the real world are indistinguishable from the point of view of $\mathcal{A}$ since the simulator, and the real world does not send any messages outside of the ideal functionality $\mathcal{F}_{\mathsf{sVOLE}}^{2,r^{(2)}}$. From

the point of view of $\mathcal{Z}$, all the honest parties output are known. By the definition of the ideal functionality $\boldsymbol{b}^0 = \boldsymbol{b} - \boldsymbol{b}^1 \bmod 2$ and $\tilde{\boldsymbol{b}}^0 = \boldsymbol{b} - \tilde{\boldsymbol{b}}^1 \bmod p$. As $\boldsymbol{b}^0$ is selected uniformly randomly in the real world, it is identically distributed and consistent with the ideal world. $\mathcal{Z}$ can check if $\tilde{b}^0_j = m_{b^0,j} + b^0_j \cdot \delta_j + b^0_j$. By the following derivation we show that $\tilde{b}^0_j$ is consistent between the two worlds.

$$\begin{aligned}
\tilde{b}^0_j &= b_j - \tilde{b}^1_j \bmod p \\
&= b_j - b^1_j + m_{0,j} \bmod p \\
&= b_j - b^1_j + m_{0,j} + b^0(\delta_j - \delta_j) \bmod p \\
&= m_{0,j} + b^0\delta_j + b - b^1_j - b^0\delta_j \bmod p
\end{aligned}$$

By case distinction on b^0

- If $b^0_j = 0$ we have $\tilde{b}^0_j = m_{0,j}$
- If $b^0_j = 1$, from the assumption of the case distinction we have $b_j = 1 - b^1_j \bmod p$ hence

$$\begin{aligned}
\tilde{b}^0_j &= m_{0,j} + \delta_j + b_j - b^1_j - m_{0,j} + m_{1,j} + 2 \cdot b^1_j \\
&= \delta_j + 1 - b^1_j - b^1_j + m_{1,j} + 2 \cdot b^1_j \\
&= m_{1_j} + \delta_j + 1
\end{aligned}$$

From this the ideal and real world produce consistent output, and the transcript is identically distributed making the two worlds indistinguishable. □