Large-Scale Private Set Intersection in the Client-Server Setting

Yunqing Sun* Jonathan Katz† Mariana Raykova‡ Phillipp Schoppmann§
Xiao Wang¶

Abstract

Private set intersection (PSI) allows two parties to compute the intersection of their sets without revealing anything else. In some applications of PSI, a server holds a large set and needs to run PSI with many clients, each with its own small set. In this setting, however, all existing protocols fall short: they either incur too much cost to compute the intersections for many clients or cannot achieve the desired security requirements.

We design a protocol that particularly suits this setting with simulation-based security against malicious adversaries. In our protocol, the server publishes a one-time, linear-size encoding of its set. Then, multiple clients can each perform a cheap interaction with the server of complexity linear in the size of each client's set. A key ingredient of our protocol is an efficient instantiation of an oblivious verifiable unpredictable function, which could be of independent interest. To obtain the intersection, the client can download the encodings directly, which can be accelerated via content distribution networks or peer-to-peer networks since the same encoding is used by all clients; alternatively, clients can fetch only the relevant ones using verifiable private information retrieval.

Our implementation shows very high efficiency. For a server holding 10^8 elements and each client holding 10^3 elements, the size of the server's encoding is 800 MB; interacting with each client uses 60 MB of communication and runs in under 5 s in a WAN network with 120 Mbps bandwidth. Compared with the state-of-the-art PSI protocol, our scheme requires only 0.017 USD per client on an AWS server, which is 5x lower.

1 Introduction

Private set intersection (PSI) allows two parties to compute the intersection of their private sets without revealing anything else. It has found many applications, including genome testing [6], botnet detection [44], online advertising [31], compromised credential checking [47], contact discovery [36], etc. In many applications, each client holds a small set and wants to know the intersection with a large set from the server. E.g., this is the case for "password checkup" service, where the server holds a large set of compromised credentials while each client has its own credentials and wants to know if any has been compromised. Another application is contact discovery, where the server holds a large set of phone information of all users while each client has its own list of contacts and wants to know among them who are also using the app.

Theoretically, any secure PSI protocol could be used in the above setting. From a practical point of view, however, using existing protocols may be prohibitive since they are not tailored to the specific constraints encountered here. For example, Signal adopted an SGX-based solution;

*Northwestern University, yunqing.sun@northwestern.edu
†Google, jkatz2@gmail.com, portions of this work done while at the University of Maryland
‡Google, marianar@google.com
§Google, schoppmann@google.com
¶Northwestern University, wangxiao@northwestern.edu

Google adopted a solution with k-anonymity and other trade-offs for better efficiency. Existing PSI protocols can be categorized into the following classes.

1. Most PSI protocols [46, 52, 48, 19] require communication linear in the size of both parties' sets. If a server needs to run PSI with two different clients, two independent executions are needed, doubling the total cost. Furthermore, there exists a security risk regarding consistency in that, even with a full malicious secure protocol, a server might use different sets for different clients. In practical terms, even state-of-the-art malicious PSI protocols [51, 49] based on vector oblivious linear evaluation still require more than 16 bytes of communication per element in the parties' sets; for a server holding 10^8 elements, this translates to around 1.6 GB of communication when interacting with each client.

2. Several PSI protocols have been designed such that the communication is sublinear to one of the set [38, 15, 14, 16]. However, these protocols are not maliciously secure against a corrupted server and do not ensure that a single consistent set is used across all clients either. What's more, these protocols are often heavy in computation and will be increased proportionally when interacting with many clients.

3. Other works like Laconic PSI [5, 1, 4] also achieve small communication; however, they require clients performing heavy computation liner to the server's set size. [20] also allows cheaper clients but requires two non-colluding servers, and the servers need to perform computation linear to the servers' set for every service request.

To summarize, full maliciously secure protocols all require communication or computation linear to the larger set, which is costly as it needs to be repeated for all users. Alternatively, some protocols (OPRF-based or applying OPRF to FHE-based solutions) allow reusing the server-dependent communication and computation, but none of these protocols can protect against a malicious server without using heavy zero-knowledge proof techniques [34]. Security against a malicious server is particularly useful in reducing the liability of the server because the protocol ensures that the clients' privacy is protected regardless of the server's action. It also prevents the server from segregating clients, e.g., by using different subsets for PSI with different clients.

1.1 Our Contribution

In this paper, we design a PSI protocol with malicious security that is particularly suitable in the multi-client setting.

Reusable and asynchronous server encodings. Our protocol has a feature tailored towards the multi-client setting. The protocol consists of two stages. First, the server with set $X = \{x_i\}$ computes encodings of each elements, namely $EX = \{\mathsf{En}(\mathsf{sk}; x_i)\}$, where sk is a trapdoor used to encode any elements. Our protocol is maliciously secure, meaning that given the encodings, the set is unique and consistent across all clients. Then, when a client with set $Y = \{y_i\}$ comes in, the server runs an interactive protocol with this client with complexity linear to $|Y|$, which allows the clients to learn $EY = \{\mathsf{En}(\mathsf{sk}; y_i)\}$, *verifiable using* pk. The client with EX can compute $EX \cap EY$ to obtain the intersection between X and Y.

The protocol is highly flexible because the client could download the encoding anytime, even after the interaction, or could fetch only a subset of them using verifiable private information retrieval (PIR) [8, 18]. Security-wise, our protocol ensures that the server's set is fully extractable (by a simulator) given the encoding and that the interaction has to be consistent with the encoding. This prevents the server from launching any attacks mentioned in prior works with ad-hoc security against a corrupted server.

Our main observation is that PRF-based encodings and oblivious PRF are inherently not the tool for fully malicious PSI protocols: even using a malicious OPRF protocol, the underlying PSI cannot be made malicious with it because there is no way to ensure the same key is used over different OPRF calls. The main idea of our protocol is to design an encoding based on verifiable functions that ensure consistency and validity by verifying the consistency between the encodings and the public key. Utilizing the inherent verifiable property of the function, we avoid using generic zero-knowledge proofs to achieve fully malicious security, unlike prior works that construct committed OPRF [34].

Efficient oblivious verifiable unpredictable function. Although a verifiable random function could work, we further observe that verifiable unpredictable function (VUF), which is often simpler and easier to compute, is already sufficient to obtain a fully secure PSI. In particular, we adopt the verifiable unpredictable function proposed by Dodis and Yampolskiy [21] for encoding elements. We also designed an efficient oblivious VUF (OVUF) for obliviously encoding the elements, based on a common building block, namely multiplicative sharing to additive sharing conversion (MtA). This component has been designed in many prior works [22, 57, 13]. To further improve the efficiency, we also designed an "imperfect" MtA protocol that does not always return the correct relationship. However, we show that when ordering the protocol messages in a special way, we can still show that such an imperfect protocol is sufficient when computing OVUF. This is applicable in our setting because: 1) the output is only given to the client; and 2) the client can check the validity of the OVUF output, which is directly connected to the validity of the MtA outputs.

Practical Efficiency. We implemented our protocol incorporating state-of-the-art building blocks. The results show that our protocol is highly efficient. For example, performing a 10^8 v.s. 10^3 elements PSI requires downloading an 800 MB encoding asynchronously and online interaction of 0.5 second in the LAN setting (or 5 seconds in the WAN setting). The distribution of the encoding can be done via a content distribution network, and thus with significantly cheaper cost than a computational node sending the encoding. We discuss in Section 6 in how to distribute the encodings efficiently.

2 Overview

In this section, we briefly discuss our main technical contributions, leaving more details in the following sections.

2.1 Malicious Client-Server PSI from (O)VUF

Recall that a VUF is associated with a private key sk and a public key pk. It includes a keyed function $F_{\cdot}(\cdot)$ and a verification algorithm $\mathsf{Verify}(\mathsf{pk}, x, x')$ that outputs 1 if and only if $x' = F_{\mathsf{sk}}(x)$. An oblivious VUF (OVUF) allows a party with x to learn $F_{\mathsf{sk}}(x)$ from another party holding sk without any party learning anything else.

We first describe our PSI protocol from any OVUF. The server with a set X obtains a key pair $(\mathsf{sk}, \mathsf{pk})$ and can locally apply the VUF function on all elements: $x'_i = F_{\mathsf{sk}}(x_i)$. The final encoding is computed as $EX = \{H(x_i, x'_i)\}$ using a random oracle H. A simulator can extract elements from valid encodings with the help of a random oracle: given an encoding, the simulator can lookup the RO query list to find the tuple (x, x') corresponding to the output hash (the chance that there are more than one is negligible); then the server can use pk to validate if this tuple is a correct evaluation of the VUF function. The best that a server can do is to put in a $\perp$ element that (with overwhelming probability) will not match any item from the client.

To compute the intersection with a client with set Y, two parties use OVUF to let the client learn $y'_i = F_{\mathsf{sk}}(y_i)$ for each $y_i \in Y$. Then the client computes $EY = \{H(y_i, y'_i)\}$ and then computes

$EX \cap EY$, which can be used to obtain $X \cap Y$. To ensure that the encodings of EX and EY use a consistent key pair, the client needs to use the pk it fetched from PKI to check $\mathsf{Verify}(\mathsf{pk}, y_i, y_i')$ for every element. A malicious server could use a different sk, which will lead to an abort during the verification. We provide detailed protocol and proof in Section 4.

This protocol provides a lot of practical benefits. The server's set encoding is reusable across many clients. This means that such encoding can be distributed publicly using content distribution networks or peer-to-peer networks for improved accessibility. Furthermore, the interactive phase of OVUF only require the client to have the public key but not the server's encodings. Thus it provides huge flexibility in how a client access the encodings. It can download it at any point of time or can even access it in a streaming fashion. A client who does not want to read in the whole encoding can also use a verifiable PIR scheme to fetch the only the ones that can possibly in the intersection or use bucketization for a trade-off between efficiency and privacy. We discuss in Section 6 with more details.

A related but different concept is verifiable OPRF (VOPRF) [32, 2, 7, 55, 53, 33, 30, 17, 24, 39, 54, 10]. More specifically, there are two types of definitions for VOPRF. The original definition of VOPRF functionality by Jarecki, Kiayias, and Krawczyk [32] maintains the random function as part of the functionality and thus does not send the PRF key to any party. This also means that the simulator does not need to extract inputs from the client. Although schemes in this category are usually very efficient (e.g., many based on 2HashDH [17, 24, 33, 39, 54, 55] and some for post-quantum security [7, 10]), they cannot be directly used here because the server cannot locally encode the element for PSI in our case and that there is no way to extract client's set to PSI in the proof. The second type of VOPRF definition [2, 53] relies on standard MPC-like functionalities where all parties' input needs to be extractable in the proof by the simulator. While the second one can be securely composed to build a PSI protocol, all existing studies of concrete efficiency are restricted to the first category. Below, we show that OVUFs can instead be instantiated securely and efficiently.

2.2 Instantiating OVUF

In this work, we build an OVUF based on the VUF function by Dodis and Yampolskiy [21]. This construction works as $F_{\mathsf{sk}}(x) = g^{1/(\mathsf{sk}+x)} \in \mathbb{G}$, where sk is a secret key in $\mathbb{Z}_q$ held by the server. The public key $\mathsf{pk} = g^{\mathsf{sk}}$ and the verification can be done by checking if $e(F_{\mathsf{sk}}(x), \mathsf{pk} \cdot g^x) = e(g, g)$. The predictability can be reduced to the hardness of Diffie-Hellman Inversion. Note that the PRF version of this construction has been used for PSI [42] but without verifiability. One alternative is to use the Naor-Reingold PRF [45] also based on DDH and can be made oblivious. However, each invocation of NR-PRF requires $O(\kappa)$ public-key operations, not considering the cost to make it verifiable.

When running an OVUF protocol, the server has the secret key sk; the client has an element y and a public key pk. An OVUF protocol involves two parties running an interactive OVUF protocol such that the server learns nothing and the client learns $F_{\mathsf{sk}}(y) = g^{1/(\mathsf{sk}+y)}$; then the client can use the pk to verify the correctness of the VUF output using $\mathsf{Verify}(\cdot)$.

The high-level idea of the protocol works as follows:

1. Two parties sample additive secret sharing of a random value such that the server has ϕ and the client has ζ.

2. Two parties run an interactive multiplicative-to-additive conversion protocol, where the server uses sk and the client uses ζ; as the output two parties obtain c_1 and d_1 respectively such that $d_1 - c_1 = \mathsf{sk} \cdot \zeta$.

3. Similarly, two parties use the same protocol to obtain c_2 and d_2 respectively such that $d_2 - c_2 = \phi \cdot y$.

4. Server computes $m = \phi \cdot \mathsf{sk} - c_1 - c_2$ and the client computes $u = y \cdot \zeta - d_1 - d_2$. Two parties exchange these values and computes $v = m + u = (\mathsf{sk} + y)(\phi + \zeta)$.

5. The server sends $g^{\phi/v}$ to the client, who computes $g^{\phi/v} \cdot g^{\zeta/v} = g^{1/(\mathsf{sk}+y)}$.

Following this blueprint, the key step is efficiently computing multiplication to addition triples (MtA). Fully malicious MtA has been a key building block in threshold ECDSA constructions, and there have been different ways to compute it using oblivious transfer [23, 28], Paillier encryption [12, 25, 41], and Castagnos-Laguillaumie encryption [13]. See Xue et al. [57] for a more detailed summary. In this paper, we focus on the oblivious transfer based construction as it is the most computationally efficient.

The solution by Doerner et al. [23] can be viewed as the malicious version of the Gilboa protocol [26]. For two parties each with a and b as input, the high-level idea is to let them obtain additive secret sharing of $a \cdot b_i$, where b_i is the i-th bit of b. This step is done using OT. Then, two parties can obtain additive secret sharing of $a \cdot b$ by a linear combination of the shares obtained from OT. To make this protocol malicious, Doerner et al. made two changes: 1) each OT will select two sets of values, where the second set of values will be used solely for checking the correctness of the output; this way, any inconsistency can be caught by the checking. 2) instead of using b_i as an OT choice bit, they use encoded values that add extra randomness. This ensures that even if the adversary cheats and aborts with some probability during the checking, this probability does not depend on the secret b_i values. These changes lead to an overhead of $4\times$ to $5\times$ compared to the semi-honest counterpart in communication.

Our main idea is that since the outer protocol of OVUF checks the correctness of the output as part of $\mathsf{Verify}()$, one can save half of the communication by not sending the OT messages for checking. However, it is unclear in this case how to extract the adversary's cheating strategy, which is needed to show that it cannot learn anything from the randomized encoding of the bits. The randomized encoding works in the following way. First, two parties agree on a uniform and public value $\mathsf{g}^R \in \mathbb{Z}_q^\ell$. Then, to encode a value β, the client samples a random bit value $\gamma \in \{0,1\}^\ell$ and outputs $(\beta - \langle \mathsf{g}^R, \gamma \rangle) \| \gamma$. Doerner et al. designed a check such that every selective failure can be extracted; this encoding ensures that even if the adversary guessed s bits of information on the encoded value (i.e., the above output), where s is statistical security parameter, β is still hidden. Without a bit-by-bit check, it is hard to pin down the selective failure attack; thus, the proof cannot go through. In particular, the greatest challenge is to simulate the abort event when the attacker performs some selective failure attacks leading to abort with an observable probability.

We observed that by picking fresh vector g^R for every execution of the MtA protocol and sending it only after the corrupted server commits to its way of cheating, we can directly argue that any deviation will lead to an observable abort with all but negligible probability. This way, we avoid the above issue because now the adversary cannot cause abort with non-negligible probability and thus easy to simulate. Although this trick allows for the maximum efficiency, we sacrifice modularity. In particular, this weak MtA can no longer be modeled as an ideal functionality in a simple way since the attacker's cheating behavior becomes complicated. Therefore in Section 5, we describe the protocol in a modular way but prove them as a whole.

So far, it is still possible that a server use a wrong sk values during an OVUF execution. This could lead to an attack: if a sk^* is used, and $v = 0$, then the server knows that the element $y = -\mathsf{sk}^*$. To ensure that this does not happen, we add an extra check right after the second step. Essentially, we want to check that the server has c_1 such that $d_1 - c_1 = \mathsf{sk} \cdot \zeta$, where sk is the discrete log of

pk that the client knows. This is equivalent to checking $g^{d_1} \cdot g^{-c_1} = g^{\mathsf{sk}\cdot\zeta} = \mathsf{pk}^{\zeta}$. Therefore, we let the server send g^{c_1} and the client check if it equals to $g_1^d \cdot \mathsf{pk}^{-\zeta}$. This check does not let client learn anything new as the value that an honest server sends is already known to the client. On the other hand, since ζ is uniform, a cheating server would be caught with overwhelming probability. When executing multiple OVUFs, one can further hash g^{c_1} in all executions to further reduce the communication. Thus, this check incurs almost no communication and only a few exponentiations.

3 Preliminaries

3.1 Notation

We use κ as the computational security parameter and s as the statistical security parameter. Let $\mathbb{G}_1$, $\mathbb{G}_2$, and $\mathbb{G}_T$ be cyclic groups defined under a pairing-friendly elliptic curve with prime order q. Define $e : \mathbb{G}_1 \times \mathbb{G}_2$ as an efficiently computable bilinear map over pairing groups. This paper uses Type III pairing for implementation, where $\mathbb{G}_1 \neq \mathbb{G}_2$. For field element $a \in \mathbb{F}_q$, let $\frac{1}{a}$ denote the inverse of a in the field $\mathbb{F}_q$ under the multiplication operation. We use $H_\infty(\gamma)$ to denote the information entropy of γ. For consistency of notations, we use log to denote logarithm based on 2. We write $[n] = \{1, \cdots, n\}$. Bold lowercase letters like $\boldsymbol{a}$ represent row vectors, where $\boldsymbol{a}_i$ denotes the ith component of $\boldsymbol{a}$. We also write $\boldsymbol{a} \circ \boldsymbol{b}$ as Hadamard product. For $b \in \mathbb{Z}_q$, we use $Bits(b)$ to denote the bit decomposition of value b. Consider a is sampled uniformly from $\mathbb{Z}_q$, denoted as $a \leftarrow \mathbb{Z}_q$. For a set X, we use letter x_i to denote its set element, where $0 < i \leq |X|$. We define a coefficient vector g as $\mathsf{g} = \mathsf{g}^G || \mathsf{g}^R$, where $\mathsf{g}^G \in \mathbb{Z}_q^{\log q}$ is a gadget vector whose element $\mathsf{g}_i^G = 2^{i-1}$, and g^R is a uniformly sampled random vector that $\mathsf{g}^R \leftarrow \mathbb{Z}_q^{\log q + 2s}$.

3.2 Verifiable Unpredictable Function

3.2.1 Definition of VUF

The notion of verifiable unpredictable function (VUF) and verifiable random function (VRF) was first proposed in [43]. VRF is defined as a pseudo-random function that provides non-interactively verifiable proof for the correctness of its output. VUF is a weaker definition in which output is not necessarily pseudo-random but unpredictable. It contains three algorithms $(\mathsf{Gen}, \mathsf{F}, \mathsf{Verify})$. Define sets K_1, K_2, X, Y.

1. $\mathsf{Gen} : 1^\kappa \to K_1 \times K_2$. It generates a secret-public key pair (sk, pk) that $sk \in K_1$, $pk \in K_2$.
2. $\mathsf{F} : K_1 \times X \to \{Y, \perp\}$. It takes the secret key sk and an element $x \in X$ as input and outputs an element $y \in Y$ or $\perp$.
3. $\mathsf{Verify} : K_2 \times X \times Y \to \{1, 0\}$. It takes the public key pk and elements $x \in X, y \in Y$ as input and outputs 1 or 0.

A VUF shall satisfy the following properties:

Definition 1. *A function family $\mathsf{F}_{(\cdot)}(\cdot) : \{0,1\}^{a(\kappa)} \mapsto \{0,1\}^{b(\kappa)}$ is a family of VUFs, if there exists algorithms $(\mathsf{Gen}, \mathsf{F}, \mathsf{Verify})$ which satisfy the following properties:*

1. *Uniqueness: no values (pk, x, y_1, y_2) can satisfy $\mathsf{Verify}(pk, x, y_1) = \mathsf{Verify}(pk, x, y_2)$, where $pk \in K_2, x \in X, y_1, y_2 \in Y$.*
2. *Provability: if $y = \mathsf{F}_{sk}(x)$ where $x \in X, y \in Y$, then $\mathsf{Verify}(pk, x, y) = 1$.*
3. *Unpredictability: for any PPT algorithm A, which runs for a total of $s(\kappa)$ steps and does not query the oracle on x, the following is negligible:*

$$Pr\left[y = \mathsf{F}_{sk}(x) \middle| (pk, sk) \leftarrow \mathsf{Gen}(1^\kappa); (x, y) \leftarrow A^{\mathsf{F}_{sk}(\cdot)}(pk)\right] \tag{1}$$

Functionality $\mathcal{F}_{\mathsf{PSI}}$

There are a server S and multiple clients $R_1, \cdots, R_n$.

1. **Initialize:** Upon receiving $(\mathsf{server}, \mathsf{sid}, X)$ from server S, where each element x in set X satisfies $x \in U \cup \{\perp\}$, the functionality stores X and ignores subsequent request.
2. **Compute Intersection:** Upon receiving $(\mathsf{server}, \mathsf{sid}, \mathsf{compute})$ from server S and $(\mathsf{client}, \mathsf{sid}, Y)$ from any client $R_j, j \in [n]$, where element y in set Y satisfies $y \in U$, the functionality sends $X \cap Y$ to the client R_j.

Figure 1: Functionality for private set intersection.

3.2.2 DY-VUF Construction

Here, we introduce the efficient construction of VUF with short proofs and keys proposed by Dodis and Yampolskiy [21]. This construction is turned from a signature scheme proposed by Boneh and Boyen [9].

Algorithm 1. *($\mathsf{Gen}(\cdot)$, $F_{(\cdot)}(\cdot)$, $\mathsf{Verify}(\cdot)$)*

1. $\mathsf{Gen}(1^\kappa)$*:* $pk = g^{sk}$

2. $F_{sk}(x)$*:* $F_{sk}(x) = g^{\frac{1}{x+sk}}$*. If* $x + sk = 0$*,* $F_{sk}(x) = \perp$*.*

3. $\mathsf{Verify}(pk, x, y)$ *: If* $e(g^x \cdot pk, y) = e(g, g)$*, output 1; Otherwise, output 0.*

The security of this construction relies on two assumptions: q-Diffie-Hellman inversion assumption (q-DHI) and q-decisional bilinear Diffie-Hellman inversion assumption (q-DBDHI).

Assumption 1. *(q-DHI assumption) Given the* $(q+1)$*-tuple :* $(g, g^x, \cdots, g^{x^q}) \in (\mathbb{G}^*)^{q+1}$ *as input, to computes* $g^{\frac{1}{x}}$*. No t-time algorithm* $\mathcal{A}$ *has advantage at least* ε *in solving q-DHI in* $\mathbb{G}$*.*

Assumption 2. *(q-DBDHI assumption) Given the* $(q+1)$*-tuple :* $(g, g^x, \cdots, g^{x^q}) \in (\mathbb{G}^*)^{q+1}$ *as input, to distinguish* $e(g,g)^{\frac{1}{x}}$ *from random. No t-time algorithm* $\mathcal{A}$ *has an advantage at least* ε *in solving q-DBDHI in* $\mathbb{G}$*.*

3.3 Ideal Functionalities

We use the UC framework [11, 27] to prove security in the presence of a malicious, static adversary.

Private Set Intersection. Private set intersection (PSI) allows two distrusted parties to jointly compute the intersection of their private input without revealing any additional information about their input except their set size. In Figure 1, we describe the ideal functionality of our private set intersection (PSI), which allows a server to compute intersections with multiple clients. Note that this ideal functionality allows the malicious server to input invalid value $\perp$ to its set X.

Oblivious Verifiable Unpredictable Function. The functionality for oblivious verifiable unpredictable function (OVUF) is shown in Figure 2. $F_{sk}(x)$ is a VUF function that outputs verifiable and unpredictable results. OVUF involves two parties: a server and a client. The server holds a secret key sk that can compute $F_{sk}(x)$ for any input value x; the client inputs a set of private elements $\boldsymbol{y} \in \mathbb{Z}_q^n$ to this functionality. The functionality computes $F_{sk}(\boldsymbol{y}_i)$ for all inputs and lets both

Functionality $\mathcal{F}_{\mathsf{OVUF}}$

Parameters: A public VUF function F. A public parameter n.

Setup:

1. If $\mathcal{F}_{\mathsf{OVUF}}$ hasn't been set up by server S, upon receiving (init) from S, the functionality runs Gen() of F, obtains pair (sk, pk), and sends (sk, pk) to S. Otherwise, it ignores the (init) request.
2. Upon receiving (fetch) from any client R, if $\mathcal{F}_{\mathsf{OVUF}}$ has been set up by server S, the functionality sends (sid, pk) to R.

Compute:

1. Upon receiving (compute) from server S and $(\mathsf{compute}, \boldsymbol{y} \in \mathbb{Z}_q^n)$ from client R, the functionality computes $F_{sk}(\boldsymbol{y}_i)$. For each $F_{sk}(\boldsymbol{y}_i) = \perp$, insert i to a set O. The functionality sends O to both parties.
2. The functionality waits for (continue/abort) from server S. Then, it sends all the $F_{sk}(\boldsymbol{y}_i), i \notin O$ to R.

Figure 2: Functionality of OVUF.

Functionality $\mathcal{F}_{\mathsf{COT}}$

Upon receiving $(\mathsf{server}, \mathsf{sid}, \boldsymbol{\tau} \in \mathbb{G}^n, n)$ from server S and $(\mathsf{client}, \mathsf{sid}, \boldsymbol{w} \in \{0,1\}^n, n)$ from client R. For each $i \in [n]$, the functionality samples random pads $\boldsymbol{p}_i \leftarrow \mathbb{G}$ and sends it to the server. It then computes $\boldsymbol{q}_i = \boldsymbol{w}_i \cdot \boldsymbol{\tau}_i + \boldsymbol{p}_i \in \mathbb{G}$ and sends it to the client R.

Figure 3: Functionality of correlated OT.

Functionality $\mathcal{F}_{\mathsf{PKI}}$

1. Init: Upon receiving (init) from server, samples sk and computes $pk = g^{sk}$. The functionality returns (sk, pk) to server and maintains pk. The functionality ignores subsequent (init) request.
2. Fetch: Upon receiving (fetch) from any party, if the functionality has been initialized, it sends back the maintained pk; Otherwise, it does nothing.

Figure 4: Functionality of public-key infrastructure.

parties know the indices where the evaluation is $\perp$. Then, the functionality sends the remaining evaluations to the client.

Correlated Oblivious Transfer. Correlated oblivious transfer (COT) is an important variant of oblivious transfer. Our protocol adopts COT to achieve multiplicative to additive share operation. The functionality is shown in Figure 3.

Public-key Infrastructure. The public-key infrastructure (PKI) is shown in Figure 4 to help with maintaining one valid pair of (sk, pk) in system. Once it receives (init) from server, it generates a public key pair (sk, pk), returns it to server, and ignores subsequent (init) requests. Once it is initialized by server, for any (fetch) request, it sends the maintained public key pk.

Protocol Π_{PSI}

Input and parameters: The server S holds private set X and a client R_j holds private set Y, where $|X| > |Y|$. A cryptographic hash function $H : \mathbb{Z}_q \times \mathbb{G} \to \{0,1\}^\sigma$ modeled as a random oracle. Define $F_{sk}(x)$, where $x \in \mathbb{Z}_q, sk \in \mathbb{Z}_q, F_{sk}(x) \in \mathbb{G}$. Define a key pair (sk, pk), where $pk \in \mathbb{G}$.

Initialize:

1. Server S sends (init) to $\mathcal{F}_{\mathsf{OVUF}}$, and receives (sk, pk) if $\mathcal{F}_{\mathsf{OVUF}}$ hasn't been set up.
2. For each $x_i \in X$, S computes $ex_i = H(x_i, F_{sk}(x_i))$ and inserts ex_i to set EX; S publishes EX to public.

Compute Intersection:

1. Client R_j sends (fetch) to $\mathcal{F}_{\mathsf{OVUF}}$ and receives pk if $\mathcal{F}_{\mathsf{OVUF}}$ has been set up by S.
2. S sends (compute) to $\mathcal{F}_{\mathsf{OVUF}}$; R_j sends (compute, $\boldsymbol{y}$), where $\boldsymbol{y}$ is a vector constructed by the set Y, to $\mathcal{F}_{\mathsf{OVUF}}$. $\mathcal{F}_{\mathsf{OVUF}}$ sends back O to two parties. S sends (continue) to $\mathcal{F}_{\mathsf{OVUF}}$, which sends R_j $F_{sk}(y_i)$ for all $y_i \in Y$, $i \notin O$.
3. For each $i \notin O$, the client R_j computes $ey_i = H(y_i, F_{sk}(y_i))$ and inserts ey_i to EY, outputs $\{y_i \mid y_i \in Y, ey_i \in EX \cap EY\}$.

Figure 5: PSI protocol in the $\mathcal{F}_{\mathsf{OVUF}}$-hybrid model.

4 OVUF-based PSI

Given the functionality of $\mathcal{F}_{\mathsf{OVUF}}$ introduced in Section 3.3, we are able to build a fully malicious PSI. Recall from Section 2.1, our protocol starts by a server locally encoding all its private elements. Any client can interact with the server via $\mathcal{F}_{\mathsf{OVUF}}$ and obtain the VUF-evaluation of its input. Since OVUF functionality ensures correctness and consistency of the evaluation across all the inputs, the client can further compute the encodings of its own set and compute intersection accordingly. The detailed scheme is shown in Figure 5.

Given a maliciously secure $\mathcal{F}_{\mathsf{OVUF}}$, there is not much that a malicious sever can cheat. It could use wrong encodings EX for intersection but that would be equivalent to applying inconsistent x and $F_{sk}(x)$ to the random oracle, which essentially means that the server uses $\perp$ as this element. We prove that the OVUF-based PSI protocol is malicious secure in $\mathcal{F}_{\mathsf{OVUF}}$-hybrid model. The theorem is stated below with a full proof.

Theorem 1. *Protocol Π_{PSI} shown in Figure 5 UC-realizes $\mathcal{F}_{\mathsf{PSI}}$ in Figure 1 in the $\mathcal{F}_{\mathsf{OVUF}}$-hybrid model.*

Proof. Let $\mathcal{A}$ be a PPT adversary that can corrupt the server or the client. We construct a PPT simulator $\mathcal{S}$ with access to functionality $\mathcal{F}_{\mathsf{PSI}}$, which simulates the adversary's view. We consider the following two cases: malicious client and malicious server. We will prove that the joint distribution over the output of $\mathcal{A}$ and the honest party in the real world is indistinguishable from the joint distribution over the outputs of $\mathcal{S}$ and the honest party in the ideal world execution.

Corrupted server. Let $\mathcal{S}$ access the $\mathcal{F}_{\mathsf{PSI}}$ as an honest server and interact with $\mathcal{A}$ as an honest client. $\mathcal{S}$ passes all communication between $\mathcal{A}$ and environment $\mathcal{Z}$.

(1) $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{OVUF}}$. Once it receives init from $\mathcal{A}$ for the first time, it sends back (sk^*, pk^*), that $sk^* \leftarrow \mathbb{Z}_q, pk^* \leftarrow \mathbb{G}$. For subsequent init request from $\mathcal{A}$, $\mathcal{S}$ ignores them.

(2) $\mathcal{S}$ emulates random oracle H. Whenever $\mathcal{A}$ computes EX, it queries $H(x_i, q_i)$. $\mathcal{S}$ checks whether $q_i = F_{sk^*}(x_i)$. If it is, $\mathcal{S}$ records $(x_i, H(x_i, q_i))$. $\mathcal{S}$ receives EX from $\mathcal{A}$. For each $ex_i \in EX$, $\mathcal{S}$ checks whether ex_i equals to a specific $H(x_i, q_i)$ recorded. If it is, $\mathcal{S}$ inserts x_i to set X. Otherwise, $\mathcal{S}$ inserts $\perp$. $\mathcal{S}$ sends $(\mathsf{server}, \mathsf{sid}, X)$ to $\mathcal{F}_{\mathsf{PSI}}$.

$\mathcal{S}$ simulates **Compute Intersection** as follows:

(2) $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{OVUF}}$ and receives $(\mathsf{compute})$ and $(\mathsf{continue})$ from $\mathcal{A}$. Then, $\mathcal{S}$ sends $(\mathsf{server}, \mathsf{sid}, \mathsf{compute})$ to $\mathcal{F}_{\mathsf{PSI}}$.

Notice that given y_i, $F_{sk}(y_i) = \perp$ with negligible probability in real-world, the simulator didn't send O in simulation. Notice that given y_i, $F_{sk}(y_i) = \perp$ with negligible probability in real-world, the simulator didn't send O in simulation Notice that given y_i, $F_{sk}(y_i) = \perp$ with negligible probability in real-world, the simulator didn't send O in simulation

Corrupted client. Let $\mathcal{S}$ access to the $\mathcal{F}_{\mathsf{PSI}}$ as an honest client and interact with $\mathcal{A}$ as an honest server. $\mathcal{S}$ passes all communication between $\mathcal{A}$ and environment $\mathcal{Z}$.

(1) $\mathcal{S}$ samples EX^* which contains $|X|$ uniform values from $\{0,1\}^\sigma$ and sends EX^* to $\mathcal{A}$.

$\mathcal{S}$ simulates **Compute Intersection** as follows:

(1) $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{OVUF}}$. Once it receives fetch from $\mathcal{A}$, it sends back $pk^* \leftarrow \mathbb{G}$.

1. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{OVUF}}$ and records Y that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{OVUF}}$. $\mathcal{S}$ sends $(\mathsf{client}, \mathsf{sid}, Y)$ to $\mathcal{F}_{\mathsf{PSI}}$ and receives $Z = X \cap Y$ in response. $\mathcal{S}$ maintains a global record table of pairs $(y_i, F_{sk}(y_i)^*)$. For each $y_i \in Y$, $\mathcal{S}$ first checks whether y_i exists in table. If it is, $\mathcal{S}$ picks corresponding $F_{sk}(y_i)^*$ to $\mathcal{A}$. Otherwise, $\mathcal{S}$ randomly samples $F_{sk}(y_i)^* \leftarrow \mathbb{G}$, sends it to $\mathcal{A}$ in response to $\boldsymbol{y}_i$, and records the new pair of $(y_i, F_{sk}(y_i)^*)$. Note that $F_{sk}(y_i) = \perp$ with negligible probability in real-world, we didn't simulate sending O.

2. $\mathcal{S}$ emulates random oracle H. $\mathcal{S}$ records a global query-value table which contains each query (a_i, b_i) and corresponding value c_i. All the c_is in the table are inserted into a set C. Once $\mathcal{S}$ receives query (a_i, b_i), it first checks the query-value table and responds if this query exists in the table. Otherwise, it programs the random oracle as follows:

   (a) If (a_i, b_i) equals to a recorded $(y_i, F_{sk}(y_i)^*)$ pair and $a_i \in Z$. $\mathcal{S}$ chooses a random $c_i \in EX^*/C$ and sends it to $\mathcal{A}$. $\mathcal{S}$ inserts c_i to set C and inserts this query-value pair to the query-value table.

   (b) Otherwise, $\mathcal{S}$ samples random $c_i \notin EX^* \cup C$ and send it to $\mathcal{A}$. $\mathcal{S}$ inserts c_i to C and inserts this query-value pair to the query-value table.

In real-world execution, ex_i and ey_i are uniform, $F_{sk}(y_i)$ is unpredictable. $\mathcal{S}$ also uniformly samples EX^*, c_i, and $F_{sk}(y_i)^*$. $F_{sk}(y_i)^*$ satisfied the unpredictable property. By programming the random oracle in step 4, the output of $\mathcal{A}$ is same as interacting with an honest server.

□

5 Making DY-VUF Oblivious

In this section, we present an OVUF protocol secure against malicious adversaries, based on the DY-VUF construction described in Section 3.2.2. Our protocol works in the $(\mathcal{F}_{\mathsf{PKI}}, \mathcal{F}_{\mathsf{COT}})$-hybrid model with a sub-protocol named imperfect multiplicative to additive shares Π_{MtA}. In Section 5.1, we review a randomized encoding scheme. Then, in Section 5.2, we introduce the sub-protocol

Π_{MtA}, which leverages the encoding scheme. The OVUF protocol, described in Section 5.3, is constructed based on Π_{MtA}. Then, we give a complexity analysis of the proposed OVUF protocol in Section 5.4. We leave the discussion of further optimization in Section A.

5.1 Encoding for Coalesced Multiplication

We provide a brief recap of the randomized encoding scheme described by Doerner et al. [23]. However, we prove some slightly different properties of the encoding where we also take the randomness of the encoding vector g^{R}. This is valid in our protocol because, as shown in Figure 6, we sample g^{R} only after the adversary chooses where to cheat.

Single encoding. Define coefficient vector $\mathsf{g} = \mathsf{g}^{\mathsf{G}}||\mathsf{g}^{\mathsf{R}}$, where $\mathsf{g}^{\mathsf{G}} \in \mathbb{Z}_q^{\log q}$, $\mathsf{g}^{\mathsf{G}}{}_i = 2^{i-1}$, and $\mathsf{g}^{\mathsf{R}} \leftarrow \mathbb{Z}_q^{\log q+2s}$.

Algorithm 2. *Encode*$(\mathsf{g}^{\mathsf{R}} \in \mathbb{Z}_q^{\log q+2s}, \beta \in \mathbb{Z}_q)$

1. *Sample* $\gamma \leftarrow \{0,1\}^{\log q+2s}$
2. *Output* $Bits(\beta - \langle \mathsf{g}^{\mathsf{R}}, \gamma \rangle)||\gamma$

Lemma 1. *Given uniform* $\gamma \leftarrow \{0,1\}^{\log q+2s}$ *and* $\mathsf{g}^{\mathsf{R}} \leftarrow \mathbb{Z}_q^{\log q+2s}$, $h_{\mathsf{g}^R}(\gamma) := \langle \mathsf{g}^{\mathsf{R}}, \gamma \rangle$ *is statistically close to uniform distribution with a statistical distance of at most* 2^{-s}.

Proof. Let $\varepsilon = 2^{-s}$, then we have $\log q = \log q + 2s - 2\log(\frac{1}{\varepsilon})$. We can define a set of functions $\mathcal{H}$, such that for each $h \in \mathcal{H}$, it is the form of $\mathbb{Z}_q^{\log q+2s} \times \{0,1\}^{\log q+2s} \to \mathbb{Z}_q$. Each function, parameterized by g^R as $h_{\mathsf{g}^R}(\gamma) := \langle \mathsf{g}^{\mathsf{R}}, \gamma \rangle$ is a 2-universal hash function [22] with output bit length $\log q$. For input $\gamma \leftarrow \{0,1\}^{\log q+2s}$, its entropy is $H_\infty(\gamma) = \log q + 2s$. Thus, we have

$$\log q = H_\infty(\gamma) - 2\log(\frac{1}{\varepsilon}) \tag{2}$$

Equation 2 satisfies the Leftover Hash Lemma, that the output bit length of the 2-universal hash function equals the entropy of input minus $2\log(\frac{1}{\varepsilon})$. Thus, for any g^{R} uniformly distributed over $\mathbb{Z}_q^{\log q+2s}$ and independent of γ, we have

$$\sigma[(h_{\mathsf{g}^R}(\gamma), \gamma), (U, \gamma)] \leq \varepsilon$$

where U is uniform distributed over $\{0,1\}^{\log q}$ and independent of g^R. Thus, the statistical distance of $h_{\mathsf{g}^R}(\gamma)$ and U is at most 2^{-s}. □

Batch encoding. When encoding more than one element, it is possible to perform better than encoding each element independently.

Algorithm 3. *BatchEncode*$(\mathsf{g}^{\mathsf{R}} \in \mathbb{Z}_q^{\log q+2s}, \{\beta^1, \cdots, \beta^n\} \in \mathbb{Z}_q^n)$

1. *Sample* $\gamma^1 \leftarrow \{0,1\}^{\log q}, \cdots, \gamma^n \leftarrow \{0,1\}^{\log q}, \gamma^{n+1} \leftarrow \{0,1\}^{2s}$
2. *Output*

$$Bits(\beta^1 - \langle \mathsf{g}^{\mathsf{R}}, \gamma^1||\gamma^{n+1} \rangle)||\gamma^1|| \cdots ||$$
$$Bits(\beta^n - \langle \mathsf{g}^{\mathsf{R}}, \gamma^n||\gamma^{n+1} \rangle)||\gamma^n||\gamma^{n+1}$$

Lemma 2. *Given uniform* $\boldsymbol{\gamma} = \gamma^1|| \cdots ||\gamma^{n+1} \leftarrow \{0,1\}^{n\log q+2s}$ *and* $\mathsf{g}^{\mathsf{R}} \leftarrow \mathbb{Z}_q^{\log q+2s}$, $h_{\mathsf{g}^{\mathsf{R}}}(\boldsymbol{\gamma}) := \langle \mathsf{g}^{\mathsf{R}}, \gamma^1||\gamma^{n+1} \rangle|| \cdots ||\langle \mathsf{g}^{\mathsf{R}}, \gamma^n||\gamma^{n+1} \rangle$ *is statistically close to uniform with a statistical distance of at most* 2^{-s}.

Proof. Let $\varepsilon = 2^{-s}$, then we have $n \log q = n \log q + 2s - 2\log(\frac{1}{\varepsilon})$. We can define a set of functions $\mathcal{H}$, such that for each $h \in \mathcal{H}$, it is the form of $\mathbb{Z}_q^{\log q+2s} \times \{0,1\}^{n\log q+2s} \to \{0,1\}^{\log q}$. Each function, parameterized by g^R as $h_{\mathsf{g}^R}(\gamma) := \langle \mathsf{g}^{\mathsf{R}}, \gamma^1 || \gamma^{n+1} \rangle || \cdots || \langle \mathsf{g}^{\mathsf{R}}, \gamma^n || \gamma^{n+1} \rangle$ is a 2-universal hash function [22] with output bit length $n \log q$. For input $\boldsymbol{\gamma} \leftarrow \{0,1\}^{n\log q+2s}$, it has information entropy $H_\infty(\boldsymbol{\gamma}) = n \log q + 2s$. Thus,

$$n \log q = H_\infty(\boldsymbol{\gamma}) - 2\log(\frac{1}{\varepsilon}) \tag{3}$$

Equation 3 satisfies the Leftover Hash Lemma, that the output bit length of the 2-universal hash function equals the entropy of input minus $2\log(\frac{1}{\varepsilon})$. Thus, for any g^{R} uniformly distributed over $\mathbb{Z}_q^{\log q+2s}$ and independent of $\boldsymbol{\gamma}$, we have

$$\sigma[(h_{\mathsf{g}^R}(\boldsymbol{\gamma}), \boldsymbol{\gamma}), (U, \boldsymbol{\gamma})] \leq \varepsilon$$

where U is uniform distributed over $\{0,1\}^{n\log q}$ and independent of g^R. Thus, the statistical distance of $h_{\mathsf{g}^R}(\boldsymbol{\gamma})$ and U is at most ε, which equals to 2^{-s}. □

5.2 Imperfect Multiplicative to Additive Shares

The imperfect multiplicative to additive (MtA) shares protocol transforms multiplicative shares to additive shares. It is imperfect because a malicious sender can execute attacks that lead to incorrect additive secret shares, depending on the receiver's input.

We use oblivious transfer based constructions to achieve this MtA. For the semi-honest version, given value $a \in \mathbb{Z}_q$ on the sender side and $b \in \mathbb{Z}_q$ on the receiver side, the sender execute $\log q$ iterations of $\mathcal{F}_{\mathsf{COT}}$ with a as input in each ith iteration, while the receiver inputs $\boldsymbol{b}_i$, representing the ith bit of the binary representation of b. The procedure and its correctness are detailed below:

1. For $i \in [\log q]$, the receiver inputs $\boldsymbol{b}_i$ to $\mathcal{F}_{\mathsf{COT}}$, while the sender inputs a. $\mathcal{F}_{\mathsf{COT}}$ sends $\boldsymbol{q}_i$ to receiver and $\boldsymbol{p}_i$ to sender, such that $\boldsymbol{q}_i = a \cdot \boldsymbol{b}_i + \boldsymbol{p}_i$.

2. Define $d = \sum_{i\in[\log q]} 2^{i-1}\boldsymbol{q}_i$, $c = \sum_{i\in[\log q]} 2^{i-1}\boldsymbol{p}_i$. Then, we have

$$\begin{aligned} d - c &= \sum_{i\in[\log q]} 2^{i-1}\boldsymbol{q}_i - \sum_{i\in[\log q]} 2^{i-1}\boldsymbol{p}_i \\ &= a \sum_{i\in[\log q]} 2^{i-1}\boldsymbol{b}_i \\ &= a \cdot b \end{aligned}$$

However, a malicious sender could potentially execute attacks to the semi-honest protocol above. Specifically, it samples an error vector $\boldsymbol{e} \in \mathbb{Z}_q^n$ and inputs $a + \boldsymbol{e}_i$ to $\mathcal{F}_{\mathsf{COT}}$ in its ith iteration. It results $d - c = a \cdot b + \sum_{i\in[\log q]} 2^{i-1}\boldsymbol{e}_i\boldsymbol{b}_i$. Given $\boldsymbol{e}_i$, the correctness of MtA transformation depends on the receiver's input b. Specifically, the transformation is correct when $\sum_{i\in[\log q]} 2^{i-1}\boldsymbol{e}_i\boldsymbol{b}_i = 0$. Prior works incorporate consistency checks and encoding to resist such malicious behaviors [23]. The consistency check, for input a in different iterations, leaks information. Encoding is involved to further protect privacy. In our construction, MtA is used in OVUF protocol in Section 5.3. Since the verifiability of OVUF implicitly gives the same property as a consistency check, we only incorporate the encoding algorithm in [23] to give an imperfect MtA protocol. To enhance efficiency, we give a batch version in Figure 6. In this scenario, two parties hold collections of n elements, denoted as $\boldsymbol{a} \in \mathbb{Z}_q^n$ and $\boldsymbol{b} \in \mathbb{Z}_q^n$, respectively. There is a receiver that employs $BatchEncode(\mathsf{g}^{\mathsf{R}}, \boldsymbol{b})$

algorithm to encode each element of its input into a batched binary representation. $\mathsf{g}^{\mathsf{R}} \in \mathbb{Z}_q^{\log q+2s}$ is randomly chosen by the receiver and sent to the sender after executing $\mathcal{F}_{\mathsf{COT}}$. To run $\mathcal{F}_{\mathsf{COT}}$ s correctly in each iteration, the sender inputs $\boldsymbol{a}_i$ and the receiver inputs corresponding $\boldsymbol{b}_i$ in its batch encoded bit representation form.

We show correctness of Figure 6 in its single encoded version below:

1. Define $\boldsymbol{w} = \mathit{Encode}(\mathsf{g}^{\mathsf{R}}, b) \in \{0,1\}^{2\log q+2s}$, which is the encoding of b. For $i \in [t+2s]$, $t = 2\log q$, the receiver inputs $\boldsymbol{w}_i$ to $\mathcal{F}_{\mathsf{COT}}$, while the sender inputs a. $\mathcal{F}_{\mathsf{COT}}$ sends $\boldsymbol{q}_i$ to receiver and $\boldsymbol{p}_i$ to sender, such that $\boldsymbol{q}_i = \boldsymbol{w}_i \cdot a + \boldsymbol{p}_i$.

2. For $\mathsf{g} = \mathsf{g}^{\mathsf{G}} || \mathsf{g}^{\mathsf{R}}$, define $d = \sum_{i\in[t]} \mathsf{g}_i \boldsymbol{q}_i + \sum_{i\in[2s]} \mathsf{g}_{t+i} \boldsymbol{q}_{t+i}$ and $c = \sum_{i\in[t]} \mathsf{g}_i \boldsymbol{p}_i + \sum_{i\in[2s]} \mathsf{g}_{t+i} \boldsymbol{p}_{t+i}$. We have
$$\begin{aligned} d - c &= \sum_{i\in[t]} \mathsf{g}_i \boldsymbol{q}_i + \sum_{i\in[2s]} \mathsf{g}_{t+i} \boldsymbol{q}_{t+i} - \sum_{i\in[t]} \mathsf{g}_i \boldsymbol{p}_i - \sum_{i\in[2s]} \mathsf{g}_{t+i} \boldsymbol{p}_{t+i} \\ &= \sum_{i\in[t]} \mathsf{g}_i (\boldsymbol{q}_i - \boldsymbol{p}_i) + \sum_{i\in[2s]} \mathsf{g}_{t+i} (\boldsymbol{q}_{t+i} - \boldsymbol{p}_{t+i}) \\ &= a(\sum_{i\in[t]} \mathsf{g}_i \boldsymbol{w}_i + \sum_{i\in[2s]} \mathsf{g}_{t+i} \boldsymbol{w}_{t+i}) \\ &= a \cdot b \end{aligned}$$

For a malicious sender executing the attacks described above, the relation will be resulted as $d - c = a \cdot b + \sum_{i\in[t]} \mathsf{g}_i \boldsymbol{e}_i \boldsymbol{w}_i + \sum_{i\in[2s]} \mathsf{g}_{t+i} \boldsymbol{e}_{t+i} \boldsymbol{w}_{t+i}$ with respect to the value of $\boldsymbol{w} = \mathit{Encode}(\mathsf{g}^{\mathsf{R}}, b) \in \mathbb{Z}_q^{t+2s}$. For $\boldsymbol{w} = \mathit{BatchEncode}(\mathsf{g}^{\mathsf{R}}, \boldsymbol{b}) \in \mathbb{Z}_q^{nt+2s}$, malicious behavior of sender will result in $\boldsymbol{d}_i - \boldsymbol{c}_i = \boldsymbol{a}_i \cdot \boldsymbol{b}_i + \boldsymbol{f}_i$, where

$$\boldsymbol{f}_i := \sum_{j\in[t]} \mathsf{g}_j \boldsymbol{w}_{(i-1)t+j} \boldsymbol{e}_{(i-1)t+j} + \sum_{k\in[2s]} \mathsf{g}_{t+k} \boldsymbol{w}_{nt+k} \boldsymbol{e}_{nt+(k-1)n+i} \tag{4}$$

We will show how to catch this incorrectness in Section 5.3 below with respect to the detailed OVUF protocol.

5.3 Oblivious VUF from Imperfect MtA

Given the imperfect balanced multiplicative to additive shares protocol in Section 5.2, we instantiate a fully malicious secure OVUF as follows.

1. Given input value sk on the server side and input vector $\boldsymbol{y} \in \mathbb{Z}_q^n$ on the client side, both parties uniformly choose random vectors $\boldsymbol{\phi} \in \mathbb{Z}_q^n$ and $\boldsymbol{\zeta} \in \mathbb{Z}_q^n$ respectively.

2. The inputs and random vectors are specifically ordered as $(\boldsymbol{\phi}_i, sk) \in \mathbb{Z}_q^2, i \in [n]$ and $(\boldsymbol{y}_i, \boldsymbol{\zeta}_i) \in \mathbb{Z}_q^2, i \in [n]$, which serves as input vector for Π_{MtA} in its ith iteration. By running Π_{MtA} on both sides for n times, both parties obtain additive secret shares of $\boldsymbol{\phi}_i \cdot \boldsymbol{y}_i$ and $sk \cdot \boldsymbol{\zeta}_i$.

3. Then, the server raise g to its secret share of $sk \cdot \boldsymbol{\zeta}_i$ for $i \in [n]$ and apply hash function on them. Server sends the hash result to the client to let it check whether the server used correct sk for each iteration.

4. Then, both parties are able to locally add $\boldsymbol{\phi}_i \cdot sk$ or $\boldsymbol{\zeta}_i \cdot \boldsymbol{y}_i$ with the secret shares of $\boldsymbol{\phi}_i \cdot \boldsymbol{y}_i + sk \cdot \boldsymbol{\zeta}_i$, respectively. The results are regarded as secret shares of $\boldsymbol{v}_i = (\boldsymbol{\phi}_i + \boldsymbol{\zeta}_i)(sk + \boldsymbol{y}_i)$. Both parties exchanges the results to recover $\boldsymbol{v}_i$.

Protocol Π_{MtA}

Inputs: P_0 holds $\boldsymbol{a} \in \mathbb{Z}_q^n$. P_1 holds $\boldsymbol{b} \in \mathbb{Z}_q^n$.
Protocol:

1. P_1 samples $\mathsf{g}^{\mathsf{R}} \leftarrow \mathbb{Z}_q^{\log q+2s}$. P_1 encodes $\boldsymbol{b}$ by computing $\boldsymbol{w} := BatchEncode(\mathsf{g}^{\mathsf{R}}, \boldsymbol{b}) \in \{0,1\}^{nt+2s}$, where $t = 2\log q$.
2. For $i \in [n], j \in [t]$, P_1 inputs $\boldsymbol{w}_{(i-1)t+j}$ to $\mathcal{F}_{\mathsf{COT}}$. P_0 inputs $\boldsymbol{a}_i \in \mathbb{F}_q$ to $\mathcal{F}_{\mathsf{COT}}$. P_0 receives $\boldsymbol{p}_{i,j} \in \mathbb{F}_q$ from $\mathcal{F}_{\mathsf{COT}}$. P_1 receives $\boldsymbol{q}_{i,j} \in \mathbb{F}_q$ from $\mathcal{F}_{\mathsf{COT}}$.
3. For $k \in [2s]$, P_1 inputs $\boldsymbol{w}_{nt+k}$ to $\mathcal{F}_{\mathsf{COT}}$. P_0 inputs $\boldsymbol{a} \in \mathbb{F}_q^n$ to $\mathcal{F}_{\mathsf{COT}}$. P_0 receives $\{\boldsymbol{p}'_{1,k}, \cdots, \boldsymbol{p}'_{n,k}\} \in \mathbb{F}_q^n$ from $\mathcal{F}_{\mathsf{COT}}$. P_1 receives $\{\boldsymbol{q}'_{1,k}, \cdots, \boldsymbol{q}'_{n,k}\} \in \mathbb{F}_q^n$ from $\mathcal{F}_{\mathsf{COT}}$.
4. P_1 sends g^{R} to P_0.
5. For $j \in [t]$, $k \in [2s]$, $i \in [n]$, P_0 computes

$$\boldsymbol{c}_i = \sum_{j\in[t]} \mathsf{g}_j \cdot \boldsymbol{p}_{i,j} + \sum_{k\in[2s]} \mathsf{g}_{t+k} \cdot \boldsymbol{p}'_{i,k}$$

   P_1 computes

$$\boldsymbol{d}_i = \sum_{j\in[t]} \mathsf{g}_j \cdot \boldsymbol{q}_{i,j} + \sum_{k\in[2s]} \mathsf{g}_{t+k} \cdot \boldsymbol{q}'_{i,k}$$

   such that $\boldsymbol{d}_i - \boldsymbol{c}_i = \boldsymbol{a}_i \cdot \boldsymbol{b}_i$.

Figure 6: The MtA protocol in $\mathcal{F}_{\mathsf{COT}}$-hybrid.

5. Both parties are able to compute $g^{\phi_i/\boldsymbol{v}_i}$ and $g^{\zeta_i/\boldsymbol{v}_i}$, respectively. Given $g^{\phi_i/\boldsymbol{v}_i}$, the client can computes $F_{sk}(\boldsymbol{y}_i) = g^{\phi_i/\boldsymbol{v}_i} \cdot g^{\zeta_i/\boldsymbol{v}_i}$ and verify correctness of the protocol using the fetched pk from the setup phase.

The detailed scheme is shown in Figure 7. Its correctness can be directly verified. For security, we assume the client acts as a receiver in the execution of $\mathcal{F}_{\mathsf{COT}}$ in sub-protocol Π_{MtA}, while the server acts as a sender. A malicious client might send the wrong $\boldsymbol{y}_i$ or $\boldsymbol{u}_i$ to the server. Incorrect $\boldsymbol{y}_i$ can be extracted by $\mathcal{S}$ given g^{R} from the client. Incorrect $\boldsymbol{u}_i$ leads to abort with all but negligible probability, which can be simulated by $\mathcal{S}$ constructing message $\boldsymbol{h}_i$ to manipulate abort probability. A malicious server could execute selective failure attack in Π_{OVUF} and bias the secret shares of $\boldsymbol{v}_i$ to be $\boldsymbol{u}_i + \boldsymbol{m}_i = \mathsf{diff}_{\mathsf{i}} + (\boldsymbol{\phi}_i + \boldsymbol{\zeta}_i)(sk + \boldsymbol{y}_i) = \mathsf{diff}_{\mathsf{i}} + \boldsymbol{v}_i$. $\mathsf{diff}_{\mathsf{i}}$ resulted from the incorrectness stated in Section 5.2 that $\mathsf{diff}_{\mathsf{i}} = \boldsymbol{f}_1^i + \boldsymbol{f}_2^i$. $\boldsymbol{f}_1^i$ resulted from incorrect $\boldsymbol{\phi}_i \cdot \boldsymbol{y}_i$ and $\boldsymbol{f}_2^i$ resulted from incorrect $sk \cdot \boldsymbol{\zeta}_i$. In the server's perspective, g^{R} is received after the selective failure attack has been executed. For any element $\mathsf{g}_{\mathsf{i}}^{\mathsf{R}}$ uniformly distributed over $\mathbb{Z}_q$, $\mathsf{diff}_{\mathsf{i}}$ is uniformly distributed over $\mathbb{Z}_q$. If the sender sends $\boldsymbol{m}_i$ and $\boldsymbol{h}_i$ honestly, the verification of $F_{sk}(\boldsymbol{y}_i)$ passes if and only if $\mathsf{diff}_{\mathsf{i}} = 0$, which is with negligible probability. If not, the verification of $F_{sk}(\boldsymbol{y}_i)$ passes if and only if diff equals a specific number that results in correct $F_{sk}(\boldsymbol{y}_i)$, which is negligible either. Thus, the server's malicious behavior can be simulated by $\mathcal{S}$ with all but negligible abort probability. The detailed proof of the security of the proposed Π_{OVUF} with sub-protocol Π_{MtA} in the hybrid of ($\mathcal{F}_{\mathsf{PKI}}$, $\mathcal{F}_{\mathsf{COT}}$) is shown in Theorem 2.

Theorem 2. *Protocol Π_{OVUF} with sub-protocol Π_{MtA} shown in Figure 7 UC-realizes $\mathcal{F}_{\mathsf{OVUF}}$ in $(\mathcal{F}_{\mathsf{PKI}}, \mathcal{F}_{\mathsf{COT}})$-hybrid model.*

Protocol Π_{OVUF}

Inputs: Client R holds vector $\boldsymbol{y} \in \mathbb{Z}_q^n$.
Setup:

1. If $\mathcal{F}_{\mathsf{PKI}}$ is not initialized, server S sends (init) to $\mathcal{F}_{\mathsf{PKI}}$, and obtains pair (sk, pk) that $pk = g^{sk}$.
2. Client R sends (fetch) to $\mathcal{F}_{\mathsf{PKI}}$. If $\mathcal{F}_{\mathsf{PKI}}$ is initialized, it sends pk to R; Otherwise, it does nothing.

Protocol:

1. Server S chooses $\boldsymbol{\phi} \leftarrow \mathbb{Z}_q^n$; client R chooses $\boldsymbol{\zeta} \leftarrow \mathbb{Z}_q^n$.
2. For $i \in [n]$, S holds vector $\boldsymbol{a}^i = (\boldsymbol{\phi}_i, sk) \in \mathbb{Z}_q^2$, R holds vector $\boldsymbol{b}^i = (\boldsymbol{y}_i, \boldsymbol{\zeta}_i) \in \mathbb{Z}_q^2$. Both parties run Π_{MtA} with the stated input vector above. Then, S receives $\boldsymbol{c}^i \in \mathbb{Z}_q^2$, R receives $\boldsymbol{d}^i \in \mathbb{Z}_q^2$, such that $\boldsymbol{d}_1^i - \boldsymbol{c}_1^i = \boldsymbol{\phi}_i \cdot \boldsymbol{y}_i$, $\boldsymbol{d}_2^i - \boldsymbol{c}_2^i = sk \cdot \boldsymbol{\zeta}_i$
3. S computes $V_S = H(g^{\boldsymbol{c}_2^1}, \cdots, g^{\boldsymbol{c}_2^n})$, and sends V_S to R. R computes $V_R = H(g^{\boldsymbol{d}_2^1}/pk^{\boldsymbol{\zeta}_1}, \cdots, g^{\boldsymbol{d}_2^n}/pk^{\boldsymbol{\zeta}_n})$. R checks whether $V_R = V_S$ and aborts if they are not equal.
4. S sends $\boldsymbol{m}$ to R, where $\boldsymbol{m}_i = \boldsymbol{\phi}_i \cdot sk - \boldsymbol{c}_1^i - \boldsymbol{c}_2^i$. R sends $\boldsymbol{u}$ to S, where $\boldsymbol{u}_i = \boldsymbol{y}_i \cdot \boldsymbol{\zeta}_i + \boldsymbol{d}_1^i + \boldsymbol{d}_2^i$. Both S and R computes $\boldsymbol{v}_i = \boldsymbol{u}_i + \boldsymbol{m}_i$.
5. For each $i \in [n]$, if $\boldsymbol{v}_i = 0$, S samples $\boldsymbol{h}_i \leftarrow \mathbb{G}$. If $\boldsymbol{v}_i \neq 0$, S computes $\boldsymbol{h}_i = g^{\boldsymbol{\phi}_i/\boldsymbol{v}_i}$. Then S sends $\boldsymbol{h}$ to R. For each $\boldsymbol{v}_i = 0, i \in [n]$, R sets $F_{sk}(\boldsymbol{y}_i) = \perp$. Otherwise, R computes $g^{\boldsymbol{\zeta}_i/\boldsymbol{v}_i}$ and $F_{sk}(\boldsymbol{y}_i) = \boldsymbol{h}_i \cdot g^{\boldsymbol{\zeta}_i/\boldsymbol{v}_i}$.
6. R checks $e(g^{\boldsymbol{y}_i} \cdot pk, F_{sk}(\boldsymbol{y}_i)) = e(g, g)$ for each $i \in [n]$. Otherwise it aborts.

Figure 7: OVUF protocol in ($\mathcal{F}_{\mathsf{COT}}$, $\mathcal{F}_{\mathsf{PKI}}$)-hybrid model with sub-protocol Π_{MtA}.

Proof. Let $\mathcal{A}$ be a PPT adversary that allows to corrupt the server or the client. We construct a PPT simulator $\mathcal{S}$ with access to functionality $\mathcal{F}_{\mathsf{OVUF}}$, which simulates the adversary's view. We consider the following two cases: malicious client and malicious server. The client acts as the receiver of $\mathcal{F}_{\mathsf{COT}}$ in sub-protocol Π_{MtA}, while the server acts as the sender. We will prove that the joint distribution over the output of $\mathcal{A}$ and the honest party in the real world is indistinguishable from the joint distribution over the outputs of $\mathcal{S}$ and the honest party in the ideal world execution.

Corrupted client. Let $\mathcal{S}$ access to $\mathcal{F}_{\mathsf{OVUF}}$ as an honest client and interact with $\mathcal{A}$ as an honest server. $\mathcal{S}$ passes all communication between $\mathcal{A}$ and environment $\mathcal{Z}$.

0. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{PKI}}$, once it receives fetch from $\mathcal{A}$. $\mathcal{S}$ samples $pk^* \leftarrow \mathbb{G}$ and sends it to $\mathcal{A}$.

1-2. For $i \in [n]$, $\mathcal{S}$ simulates the ith iteration of sub-protocol Π_{MtA} below.

(1)-(3) $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{COT}}$ and receives $\boldsymbol{w} \in \mathbb{Z}_q^{2t+2s}$ from $\mathcal{A}$. $\mathcal{S}$ samples $\boldsymbol{q} \leftarrow \mathbb{Z}_q^{2t}$, $\boldsymbol{q}' \leftarrow \mathbb{Z}_q^{4s}$ and sends them to $\mathcal{A}$.

(4) $\mathcal{S}$ receives g^R from $\mathcal{A}$. $\mathcal{S}$ computes $\boldsymbol{y}_i$ and $\boldsymbol{\zeta}_i$ as follows:

$$\boldsymbol{y}_i = \boldsymbol{b}_1^i = \sum_{j \in [t]} \mathsf{g}_j \boldsymbol{w}_j + \sum_{j \in [t+1, t+2s]} \mathsf{g}_j \boldsymbol{w}_{t+j}$$

$$\boldsymbol{\zeta}_i = \boldsymbol{b}_2^i = \sum_{j \in [t]} \mathsf{g}_j \boldsymbol{w}_{t+j} + \sum_{j \in [t+1, t+2s]} \mathsf{g}_j \boldsymbol{w}_{t+j}$$

(5) $\mathcal{S}$ computes $\boldsymbol{d}^i$ as an honest P_1 does in step 5 in Π_{MtA}.

3. $\mathcal{S}$ samples V_S^* and sends it to $\mathcal{A}$. $\mathcal{S}$ emulates H and receives query q from $\mathcal{A}$. If $q = (g^{\boldsymbol{d}_2^1}/pk^{\boldsymbol{\zeta}_1}, \cdots, g^{\boldsymbol{d}_2^n}/pk^{\boldsymbol{\zeta}_n})$, $\mathcal{S}$ sends V_S^* to $\mathcal{A}$. Otherwise, $\mathcal{S}$ sends a random value to $\mathcal{A}$. $\mathcal{S}$ aborts if $\mathcal{A}$ aborts.

4. $\mathcal{S}$ sends $(\mathsf{compute}, \boldsymbol{y})$ to $\mathcal{F}_{\mathsf{OVUF}}$ and receives set O from $\mathcal{F}_{\mathsf{OVUF}}$. For all the index $i \in O$, $\mathcal{S}$ sends $\boldsymbol{m}_i^* = -(\boldsymbol{y}_i \cdot \boldsymbol{\zeta}_i + \boldsymbol{d}_1^i + \boldsymbol{d}_2^i)$ to $\mathcal{A}$. Otherwise, $\mathcal{S}$ sends $\boldsymbol{m}_i^* \leftarrow \mathbb{Z}_q$ to $\mathcal{A}$. $\mathcal{S}$ receives $\boldsymbol{u}_i$ and computes $\boldsymbol{v}_i = \boldsymbol{m}_i^* + \boldsymbol{u}_i$ for each $i \in [n]$.

5. $\mathcal{S}$ waits to receive $F_{sk}(\boldsymbol{y}_i)$ for each $i \notin O$. For each $i \in [n]$, $\mathcal{S}$ checks whether $\boldsymbol{u}_i = \boldsymbol{y}_i \cdot \boldsymbol{\zeta}_i + \boldsymbol{d}_1^i + \boldsymbol{d}_2^i$. If it is and $\boldsymbol{v}_i \neq 0$, $\mathcal{S}$ simulates $\boldsymbol{h}_i^* = \frac{F_{sk}(\boldsymbol{y}_i)}{g^{\frac{\boldsymbol{\zeta}_i}{\boldsymbol{m}_i^* + \boldsymbol{u}_i}}}$, and sends it to $\mathcal{A}$. Otherwise, $\mathcal{S}$ simulates $\boldsymbol{h}_i \leftarrow \mathbb{G}$ and sends it to $\mathcal{A}$.

6. $\mathcal{S}$ aborts if $\mathcal{A}$ aborts and outputs what $\mathcal{A}$ outputs.

We are going to show the simulated execution is indistinguishable from the real protocol execution.

Hybrid $\mathcal{H}_0$. Same as real-world execution in ($\mathcal{F}_{\mathsf{COT}}$, $\mathcal{F}_{\mathsf{PKI}}$)-hybrid.

Hybrid $\mathcal{H}_1$. This hybrid is identical to $\mathcal{H}_0$ except $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{PKI}}$, $\mathcal{F}_{\mathsf{COT}}$, and simulates the messages to $\mathcal{A}$ as follows:

For step 0, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{PKI}}$, samples $pk^* \leftarrow \mathbb{G}$ and sends it to $\mathcal{A}$. In hybrid $\mathcal{H}_0$, $\mathcal{F}_{\mathsf{PKI}}$ samples $sk \leftarrow \mathbb{Z}_q$ and computes $pk = g^{sk}$, which is uniformly distributed in $\mathbb{G}$. Thus, the pk generated by $\mathcal{S}$ is indistinguishable from the one in hybrid H_0.

For step 1-2, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{COT}}$, receives $\boldsymbol{w} \in \mathbb{Z}_q^{2t+2s}$ from $\mathcal{A}$. $\mathcal{S}$ samples $\boldsymbol{q} \leftarrow \mathbb{Z}_q^{2t}$, $\boldsymbol{q}' \leftarrow \mathbb{Z}_q^{4s}$ to $\mathcal{A}$, and receives $\mathsf{g}^R \in \mathbb{Z}_q^{\log q + 2s}$ from $\mathcal{A}$. In Hybrid $\mathcal{H}_0$, $\boldsymbol{q}$ and $\boldsymbol{q}'$ are uniformly distributed according to $\mathcal{F}_{\mathsf{COT}}$. Thus, the $\boldsymbol{q}$, $\boldsymbol{q}'$ sampled by $\mathcal{S}$ is indistinguishable from the one in Hybrid $\mathcal{H}_0$.

For step 3, $\mathcal{S}$ samples V_S^* to $\mathcal{A}$. Then, $\mathcal{S}$ emulates random oracle and returns V_S^* to $\mathcal{A}$ for query $q = (g^{\boldsymbol{d}_2^1}/pk^{\boldsymbol{\zeta}_1}, \cdots, g^{\boldsymbol{d}_2^n}/pk^{\boldsymbol{\zeta}_n})$. In hybrid $\mathcal{H}_0$, an honest server uses correct sk in Π_{MtA} and computes $V_S = H(g^{\boldsymbol{c}_2^1}, \cdots, g^{\boldsymbol{c}_2^n})$. Since $\boldsymbol{c}_2^i = \boldsymbol{d}_2^i - sk \cdot \boldsymbol{\zeta}_i$ holds for an honest server, where $\boldsymbol{d}_2^i$ and $\boldsymbol{\zeta}_i$ can be recovered by $\mathcal{S}$, the simulated V_S^* is indistinguishable from the honest V_S in Hybrid $\mathcal{H}_0$.

For step 4, $\mathcal{S}$ sends $(\mathsf{compute}, \boldsymbol{y})$ to $\mathcal{F}_{\mathsf{OVUF}}$ and waits for set O. For each $i \in O$, $\mathcal{S}$ sends $\boldsymbol{m}_i^* = -(\boldsymbol{y}_i \cdot \boldsymbol{\zeta}_i + \boldsymbol{d}_1^i + \boldsymbol{d}_2^i)$ to $\mathcal{A}$. Otherwise, $\mathcal{S}$ sends random $\boldsymbol{m}_i^* \leftarrow \mathbb{Z}_q$ to $\mathcal{A}$. In hybrid $\mathcal{H}_0$, an honest server computes $\boldsymbol{m}_i = \boldsymbol{\phi}_i \cdot sk + \boldsymbol{c}_1 + \boldsymbol{c}_2$ and sends it to $\mathcal{A}$. If $sk + \boldsymbol{y}_i = 0$, $\boldsymbol{m}_i$ satisfies the distribution that $\boldsymbol{m}_i + \boldsymbol{y}_i \cdot \boldsymbol{\zeta}_i + \boldsymbol{d}_1^i + \boldsymbol{d}_2^i = \boldsymbol{v}_i = (\boldsymbol{\phi}_i + \boldsymbol{\zeta}_i)(sk + \boldsymbol{y}_i) = 0$. The simulated $\boldsymbol{m}_i^*$ for $i \in O$ satisfies this distribution as well, which is indistinguishable from hybrid H_0. If $sk + \boldsymbol{y}_i \neq 0$, $\boldsymbol{m}_i + \boldsymbol{y}_i \cdot \boldsymbol{\zeta}_i + \boldsymbol{d}_1^i + \boldsymbol{d}_2^i = \boldsymbol{v}_i = (\boldsymbol{\phi}_i + \boldsymbol{\zeta}_i)(sk + \boldsymbol{y}_i)$. Because $\boldsymbol{\phi}_i$, is randomly sampled, $\mathcal{A}$ has no idea about the distribution of $\boldsymbol{m}_i$. Also, $\boldsymbol{c}^i$ is randomly uniform in $\mathbb{Z}_q^2$ to $\mathcal{A}$, we have $\boldsymbol{m}_i$ randomly uniform in $\mathbb{Z}_q$ to $\mathcal{A}$. The simulated $\boldsymbol{m}_i^*$ is randomly uniform in $\mathbb{Z}_q$ as well, which is indistinguishable from hybrid $\mathcal{H}_0$. Thus, the view simulated by $\mathcal{S}$ is identical to hybrid $\mathcal{H}_0$.

For step 5, $\mathcal{S}$ waits to receive $F_{sk}(\boldsymbol{y}_i)$. $\mathcal{S}$ checks whether the received $\boldsymbol{u}_i$ is correct. If it is and $\boldsymbol{v}_i \neq 0$, $\mathcal{S}$ sends $\boldsymbol{h}_i^* = \frac{F_{sk}(\boldsymbol{y}_i)}{g^{\frac{\boldsymbol{\zeta}_i}{\boldsymbol{m}_i^* + \boldsymbol{u}_i}}}$ to $\mathcal{A}$, where $\boldsymbol{m}_i^*$ is the value sampled in step 3. Otherwise, $\mathcal{S}$ samples $\boldsymbol{h}_i^* \leftarrow \mathbb{G}$ and sends it to $\mathcal{A}$ instead. In hybrid $\mathcal{H}_0$, an honest server sends $\boldsymbol{h}_i = g^{\frac{\boldsymbol{\phi}_i}{\boldsymbol{m}_i + \boldsymbol{u}_i}}$ to $\mathcal{A}$, where $\boldsymbol{m}_i$ and $\boldsymbol{h}_i$ are computed honestly. If $\boldsymbol{u}_i$ is correct, then $\boldsymbol{h}_i \cdot g^{\frac{\boldsymbol{\zeta}_i}{\boldsymbol{m}_i + \boldsymbol{u}_i}} = F_{sk}(\boldsymbol{y}_i)$. In Hybrid $\mathcal{H}_1$, $\mathcal{S}$ receives $F_{sk}(\boldsymbol{y}_i)$ from $\mathcal{F}_{\mathsf{OVUF}}$ and simulates $\boldsymbol{h}_i^* = \frac{F_{sk}(\boldsymbol{y}_i)}{g^{\frac{\boldsymbol{\zeta}_i}{\boldsymbol{m}_i^* + \boldsymbol{u}_i}}}$, such that $\boldsymbol{h}_i^* \cdot g^{\frac{\boldsymbol{\zeta}_i}{\boldsymbol{m}_i^* + \boldsymbol{u}_i}} = F_{sk}(\boldsymbol{y}_i)$. The view of $\boldsymbol{h}_i^*$ is identical to $\boldsymbol{h}_i$ in Hybrid $\mathcal{H}_0$.If $\boldsymbol{u}_i$ is not correct in hybrid $\mathcal{H}_0$, then $\boldsymbol{m}_i + \boldsymbol{u}_i \neq (\boldsymbol{\phi}_i + \boldsymbol{\zeta}_i)(sk + \boldsymbol{y}_i)$ and $\boldsymbol{h}_i \cdot g^{\frac{\boldsymbol{\zeta}_i}{\boldsymbol{m}_i + \boldsymbol{u}_i}} \neq F_{sk}(\boldsymbol{y}_i)$. In Hybrid $\mathcal{H}_1$, $\mathcal{S}$ simulates $\boldsymbol{h}_i^* \leftarrow \mathbb{G}$, we have $\boldsymbol{h}_i^* \cdot g^{\frac{\boldsymbol{\zeta}_i}{\boldsymbol{m}_i^* + \boldsymbol{u}_i}} = F_{sk}(\boldsymbol{y}_i)$

w.p. $2^{-\log q}$, which is indistinguishable from hybrid $\mathcal{H}_0$. If $\boldsymbol{u}_i$ is correct but $\boldsymbol{v}_i = 0$ in hybrid $\mathcal{H}_0$, $\mathcal{S}$ samples $\boldsymbol{h}_i^* \leftarrow \mathbb{G}$. In hybrid $\mathcal{H}_0$, $\boldsymbol{h}_i$ is sampled from $\mathbb{G}$ as well. Thus, the view simulated by $\mathcal{S}$ is identical to hybrid $\mathcal{H}_0$.

Corrupted server. Let $\mathcal{S}$ access to the $\mathcal{F}_{\mathsf{OVUF}}$ as an honest server and interact with $\mathcal{A}$ as an honest client. $\mathcal{S}$ passes all communication between $\mathcal{A}$ and environment $\mathcal{Z}$.

0. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{PKI}}$, once it receives the first init request from $\mathcal{A}$, $\mathcal{S}$ samples (sk, pk), where $pk = g^{sk}$ to $\mathcal{A}$. For subsequent init request, $\mathcal{S}$ ignores them.

1-2. For each $i \in [n]$, $\mathcal{S}$ simulates ith iteration of Π_{MtA} as follows:

   (1)-(3) $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{COT}}$ and receives a vector $\boldsymbol{\tau} \in \mathbb{Z}_q^{2(t+2s)}$ from $\mathcal{A}$. $\mathcal{S}$ samples $\boldsymbol{p} \leftarrow \mathbb{Z}_q^{2t}$, $\boldsymbol{p}' \leftarrow \mathbb{Z}_q^{4s}$ and sends them to $\mathcal{A}$. $\mathcal{S}$ checks whether the received $\boldsymbol{\tau} \in \mathbb{Z}_q^{2(t+2s)}$ satisfies a pattern that for $k \in [2]$, all the bits $\boldsymbol{\tau}_j$, $j \in [(k-1)t+1, kt] \cup j = 2t + k + (l-1)2$, $l \in [2s]$ are the same. For $k = 1$, if $\boldsymbol{\tau}_j$ are the same, $\mathcal{S}$ extracts $\boldsymbol{\phi}_i = \boldsymbol{\tau}_j$. For $k = 2$, if $\boldsymbol{\tau}_j$ are the same, $\mathcal{S}$ extracts $sk' = \boldsymbol{\tau}_j$.

   (4) $\mathcal{S}$ samples $\mathsf{g}^R \leftarrow \mathbb{Z}_q^{\log q + 2s}$ and sends g^R to $\mathcal{A}$.

   (5) $\mathcal{S}$ computes $\boldsymbol{c}^i$ as an honest P_0 does in step 5 in Π_{MtA}.

3. $\mathcal{S}$ emulates random oracle H and receives query q from $\mathcal{A}$. $\mathcal{S}$ samples V_S to $\mathcal{A}$ and records (q, V_S). Once $\mathcal{S}$ receives V_S from $\mathcal{A}$, $\mathcal{S}$ first checks whether sk's have been extracted in last step and each $sk' = sk$. Then, $\mathcal{S}$ checks whether the corresponded $q = (g^{\boldsymbol{c}_2^1}, \cdots, g^{\boldsymbol{c}_2^i})$. If it is, $\mathcal{S}$ continue, Otherwise, $\mathcal{S}$ aborts. For other cases that any $sk' \neq sk$ or sk' can not be extracted, $\mathcal{S}$ aborts directly.

4. $\mathcal{S}$ receives $\boldsymbol{m}$ from $\mathcal{A}$. $\mathcal{S}$ sends ($\mathsf{compute}$) to $\mathcal{F}_{\mathsf{OVUF}}$ and receives set O from $\mathcal{F}_{\mathsf{OVUF}}$. For each $i \in [n]$, if the received $\boldsymbol{\tau} \in \mathbb{Z}_q^{2(t+2s)}$ satisfies the pattern in step 1-2 and $i \in O$, $\mathcal{S}$ sends value $\boldsymbol{u}_i^* = -\boldsymbol{\phi}_i \cdot sk + \boldsymbol{c}_1^i + \boldsymbol{c}_2^i$ to $\mathcal{A}$. Otherwise, $\mathcal{S}$ sends $\boldsymbol{u}_i^* \leftarrow \mathbb{Z}_q$ to $\mathcal{A}$.

5. $\mathcal{S}$ waits to receive $\boldsymbol{h}$.

6. For the ith iteration, if $\boldsymbol{\tau}$ satisfies the pattern stated above and $i \notin O$, $\mathcal{S}$ checks whether $\boldsymbol{h}_i = g^{\frac{\boldsymbol{\phi}_i}{\boldsymbol{m}_i + \boldsymbol{u}_i^*}}$, $\boldsymbol{m}_i = \boldsymbol{\phi}_i \cdot sk - \boldsymbol{c}_1^i - \boldsymbol{c}_2^i$ and sends $\mathsf{continue}$ to $\mathcal{F}_{\mathsf{OVUF}}$. If $\boldsymbol{\tau}$ satisfies the pattern stated above and $i \in O$, $\mathcal{S}$ checks whether $\boldsymbol{m}_i = \boldsymbol{\phi}_i \cdot sk - \boldsymbol{c}_1^i - \boldsymbol{c}_2^i$ and sends $\mathsf{continue}$ to $\mathcal{F}_{\mathsf{OVUF}}$. Otherwise, $\mathcal{S}$ sends abort to $\mathcal{F}_{\mathsf{OVUF}}$.

We are going to show the simulated execution is indistinguishable from the real protocol execution.

Hybrid $\mathcal{H}_0$. Same as real-world execution in ($\mathcal{F}_{\mathsf{PKI}}$,$\mathcal{F}_{\mathsf{COT}}$)-hybrid model.

Hybrid $\mathcal{H}_1$. This hybrid is identical to $\mathcal{H}_0$ except $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{PKI}}$, $\mathcal{F}_{\mathsf{COT}}$ and generates the messages to $\mathcal{A}$ as follows:

For $\mathsf{Step\ 0}$, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{PKI}}$ and samples (sk, pk) to $\mathcal{A}$ once it receives the first init request from $\mathcal{A}$, which is indistinguishable from Hybrid $\mathcal{H}_0$.

For $\mathsf{Step\ 1\text{-}2}$, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{COT}}$ and waits to receive $\boldsymbol{\tau}$. Then, $\mathcal{S}$ sends $\boldsymbol{p} \leftarrow \mathbb{Z}_q^{2t}$ and $\boldsymbol{p}' \leftarrow \mathbb{Z}_q^{4s}$ to $\mathcal{A}$. $\mathcal{S}$ samples $\mathsf{g}^{\mathsf{R}} \leftarrow \mathbb{Z}_q^{\log q + 2s}$ and sends it to $\mathcal{A}$. For an honest client in hybrid $\mathcal{H}_0$, it samples $\boldsymbol{p} \leftarrow \mathbb{Z}_q^{2t}$, $\boldsymbol{p}' \leftarrow \mathbb{Z}_q^{4s}$, $\mathsf{g}^{\mathsf{R}} \leftarrow \mathbb{Z}_q^{\log q + 2s}$ as well, which is indistinguishable from this hybrid.

For $\mathsf{Step\ 4}$, $\mathcal{S}$ receives $\boldsymbol{m}$. $\mathcal{S}$ sends ($\mathsf{compute}$) to $\mathcal{F}_{\mathsf{OVUF}}$ and waits for O. For each iteration $i \in [n]$, if the received $\boldsymbol{\tau}$ satisfies the pattern stated above and $i \in O$, $\mathcal{S}$ sends $\boldsymbol{u}_i^* = -\boldsymbol{\phi}_i \cdot sk + \boldsymbol{c}_1^i + \boldsymbol{c}_2^i$

to $\mathcal{A}$. Otherwise, $\mathcal{S}$ sends $\boldsymbol{u}_i^* \leftarrow \mathbb{Z}_q$ to $\mathcal{A}$. In the real-world execution, for ith iteration, if $\mathcal{A}$ sends $\boldsymbol{\tau}$ correctly and $i \in O$, an honest client computes $\boldsymbol{u}_i$ such that $\boldsymbol{u}_i + \boldsymbol{\phi}_i \cdot sk - \boldsymbol{c}_1^i - \boldsymbol{c}_2^i = 0$. The sampled $\boldsymbol{u}_i^*$ satisfy this distribution as well. If $i \notin O$, $\boldsymbol{u}_i + \boldsymbol{\phi}_i \cdot sk - \boldsymbol{c}_1^i - \boldsymbol{c}_2^i = \boldsymbol{v}_i = (\boldsymbol{\phi}_i + \boldsymbol{\zeta}_i)(sk + \boldsymbol{y}_i)$, which is uniformly distributed over $\mathbb{Z}_q$. Thus, $\boldsymbol{u}_i$ is uniformly distributed, same as the sampled one.

If there exists an error $\boldsymbol{e}$ sampled by a malicious server, an honest client computes $\boldsymbol{u}_i$ such that $\boldsymbol{u}_i + \boldsymbol{\phi}_i \cdot sk - \boldsymbol{c}_1^i - \boldsymbol{c}_2^i = \boldsymbol{v}_i = (\boldsymbol{\phi}_i + \boldsymbol{\zeta}_i)(sk + \boldsymbol{y}_i) + \mathsf{diff_i}$. $\mathsf{diff_i} = \boldsymbol{f}_1^i + \boldsymbol{f}_2^i$, where $\boldsymbol{f}_1^i$ resulted from incorrect $\boldsymbol{\phi}_i \cdot \boldsymbol{y}_i$ and $\boldsymbol{f}_2^i$ resulted from incorrect $sk \cdot \boldsymbol{\zeta}_i$ as stated in Equation 4. Since $\boldsymbol{e}$ is defined by $\mathcal{A}$ before knowing g^{R}, we have g^{R} is randomly uniform over $\mathbb{Z}_q^{\log q + 2s}$ to $\mathcal{A}$ at this time. Thus, for any given $\boldsymbol{w}$ and $\boldsymbol{e}$, $\boldsymbol{f}_i$ is uniformly distributed over $\mathbb{Z}_q$. Thus, $\mathsf{diff_i}$ is uniform distributed over $\mathbb{Z}_q$. If $sk + \boldsymbol{y}_i = 0$, an honest client computes $\boldsymbol{u}_i$ such that $\boldsymbol{u}_i + \boldsymbol{\phi}_i \cdot sk - \boldsymbol{c}_1^i - \boldsymbol{c}_2^i = 0 + \mathsf{diff_i}$. Since $\mathsf{diff_i}$ is uniform distributed over $\mathbb{Z}_q$, for any $i \in O$ but $\boldsymbol{\tau}$ not satisfy the stated pattern, the simulated $\boldsymbol{u}_i^* \leftarrow \mathbb{Z}_q$ is identical to the distribution of $\boldsymbol{u}_i$ in Hybrid $\mathcal{H}_0$. If $sk + \boldsymbol{y}_i \neq 0$,an honest client computes $\boldsymbol{u}_i$ such that $\boldsymbol{u}_i + \boldsymbol{\phi}_i \cdot sk - \boldsymbol{c}_1^i - \boldsymbol{c}_2^i = (\boldsymbol{\phi}_i + \boldsymbol{\zeta}_i)(sk + \boldsymbol{y}_i) + \mathsf{diff_i}$ in Hybrid $\mathcal{H}_0$. Since $\boldsymbol{b}_2^i \leftarrow \mathbb{Z}_q$ is randomly uniform to $\mathcal{A}$, we have $\boldsymbol{u}_i$ is uniformly distributed over $\mathbb{Z}_q$, which is identically distributed as the simulated $\boldsymbol{u}_i^*$.

Hybrid $\mathcal{H}_2$. This hybrid is identical to $\mathcal{H}_1$ except $\mathcal{S}$ aborts at Step 3 in the following conditions: 1) the extracted $sk' = sk$, the q corresponding to the received V_S not equal to $(g^{\boldsymbol{c}_1}, \cdots, g^{\boldsymbol{c}_n})$; 2) the extracted $sk' \neq sk$ 3) there is an error $\boldsymbol{e}$ added to $\boldsymbol{\tau}$ on the bits corresponding to sk, $\mathcal{S}$ couldn't extract sk' from $\boldsymbol{\tau}$. $\mathcal{S}$ also aborts at Step 6 in the following conditions: 1) $\boldsymbol{\tau}$ does not satisfy the specific pattern; 2) $i \in O$, $\boldsymbol{\tau}$ satisfy the specific pattern, $\boldsymbol{m}_i \neq \boldsymbol{\phi}_i \cdot sk - \boldsymbol{c}_1^i - \boldsymbol{c}_2^i$; 3) $i \notin O$, $\boldsymbol{\tau}$ satisfy the specific pattern, $\boldsymbol{h}_i^* \neq g^{\frac{\boldsymbol{\phi}_i}{\boldsymbol{m}_i + \boldsymbol{u}_i^*}}$ or $\boldsymbol{m}_i \neq \boldsymbol{\phi}_i \cdot sk - \boldsymbol{c}_1^i - \boldsymbol{c}_2^i$.

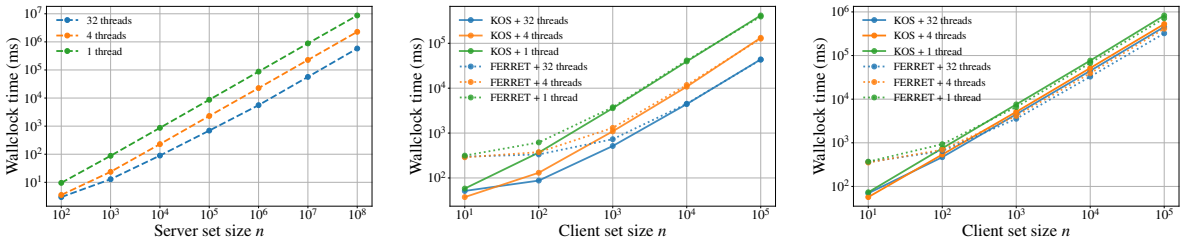
For Step 3 in hybrid $\mathcal{H}_1$, an honest client aborts if the received $V_S \neq V_R$. The client computes $V_R = H(g^{\boldsymbol{d}_2^1}/pk^{\boldsymbol{\zeta}_1}, \cdots, g^{\boldsymbol{d}_2^n}/pk^{\boldsymbol{\zeta}_n})$. For each $g^{\boldsymbol{d}_2^i}/pk^{\boldsymbol{\zeta}_i}$, it equals to $g^{\boldsymbol{d}_2^i - sk \cdot \boldsymbol{\zeta}_i}$. When adversary use correct sk but manipulate V_S from inconsistent $\boldsymbol{z}_i$, an honest client aborts in hybrid $\mathcal{H}_1$ (condition (1)). Adversary $\mathcal{A}$ might use inconsistent sk to Π_{MtA} and manipulate V_S to $\mathcal{A}$. If $\mathcal{A}$ use $sk' \neq sk$ in Π_{MtA}, both parties holds equation $\boldsymbol{c}_2^i = \boldsymbol{d}_2^i - sk' \cdot \boldsymbol{\zeta}_i$. Thus, with V_S computed from $\boldsymbol{c}_2^i$, $V_S \neq V_R$. Moreover, because of the uniformity of $\boldsymbol{d}_2^i$ and $\boldsymbol{\zeta}_i$, $\mathcal{A}$ is not able to construct $\boldsymbol{c}_2^{i'}$ that $\boldsymbol{c}_2^{i'} = \boldsymbol{d}_2^i - sk \cdot \boldsymbol{\zeta}_i$ either. Thus, the client aborts with all but negligible probability with condition (2) in hybrid $\mathcal{H}_1$. Adversary $\mathcal{A}$ might add error $\boldsymbol{e}$ to $\boldsymbol{\tau}$ on bits related to sk in Π_{MtA} and manipulate V_S to $\mathcal{A}$. In this case, both parties holds equation $\boldsymbol{c}_2^i = \boldsymbol{d}_2^i - sk' \cdot \boldsymbol{\zeta}_i + \boldsymbol{f}_2$, where $\boldsymbol{f}_2$ is computed according to Equation. 4. As we analyzed above, $\boldsymbol{f}_2$ is uniformly distributed over $\mathbb{Z}_q$ for any given $\boldsymbol{w}$ and $\boldsymbol{e}$. Thus, with V_S computed from $\boldsymbol{c}_2^i$, $V_S \neq V_R$. Similarly, $\mathcal{A}$ is unable to construct $\boldsymbol{c}_2^{i'} = \boldsymbol{d}_2^i - sk \cdot \boldsymbol{\zeta}_i$ either. Thus, the client aborts with all but negligible probability with condition (2) in hybrid $\mathcal{H}_1$.

For Step 6 in hybrid $\mathcal{H}_1$, an honest client aborts when $F_{sk}(\boldsymbol{y}_i)$ does not satisfy $e(g^{\boldsymbol{y}_i} \cdot pk, F_{sk}(\boldsymbol{y}_i)) = e(g, g)$, where $F_{sk}(\boldsymbol{y}_i) = \boldsymbol{h}_i \cdot g^{\frac{\boldsymbol{\zeta}_i}{\boldsymbol{m}_i + \boldsymbol{u}_i}}$. For the ith iteration, if $\boldsymbol{\tau}$ doesn't satisfy the specific pattern, $\mathsf{diff_i} \neq 0$ with all but negligible probability. Thus, $\boldsymbol{m}_i + \boldsymbol{u}_i \neq (sk + \boldsymbol{y}_i)(\boldsymbol{\phi}_i + \boldsymbol{\zeta}_i)$ with all but negligible probability. $\boldsymbol{h}_i \cdot g^{\frac{\boldsymbol{\zeta}_i}{\boldsymbol{m}_i + \boldsymbol{u}_i}} \neq F_{sk}(\boldsymbol{y}_i)$ with all but negligible probability. When $\boldsymbol{\tau}$ satisfies the specific pattern and $i \in O$, an honest client aborts if $\boldsymbol{m}_i + \boldsymbol{u}_i \neq 0$ and results in a $F_{sk}(\boldsymbol{y}_i)$. When $\boldsymbol{\tau}$ satisfy the specific pattern and $i \notin O$, an honest client aborts if the computed $F_{sk}(\boldsymbol{y}_i)$ does not satisfy the verification procedure. Given either wrong $\boldsymbol{m}_i$ or wrong $\boldsymbol{h}_i$, it will result in wrong $F_{sk}(\boldsymbol{y}_i)$. Therefore, this hybrid is identically distributed as the previous one.

The above hybrid argument completes this proof. □

5.4 Complexity Analysis

For each input element $y_i \in Y$, Π_{OVUF} requires $4 \log q + 4s$ COT and one g^{R}. Thus, this protocol requires $(4 \log q + 4s)n$ COT and n g^{R} in total. To improve its complexity, we propose an improved



(a) Local encoding (b) Interaction phase (LAN) (c) Interaction phase (WAN)

Figure 8: **Performance of our protocol.** We show the performance of both phases. The one-time offline local encoding time for the server set is depicted in (a). The interactive online encoding time for the client set is shown in (b) under LAN network and (c) under WAN network. Both (b) and (c) utilize different OT methods (KOS/FERRET) and number of threads (1/4/32) for comparison.

OVUF in Section A that reduces the number of g^{R}s to two and achieves better RAM usage. The key idea is to batch operations with correlated randomness together but refer to Section A for complete description of the protocol and the proof.

6 Distributing Server Encodings

Distributing the server-set encodings is the main cost of the protocol, but also the most flexible part of the protocol. Here we discuss several solutions that could be used in practice.

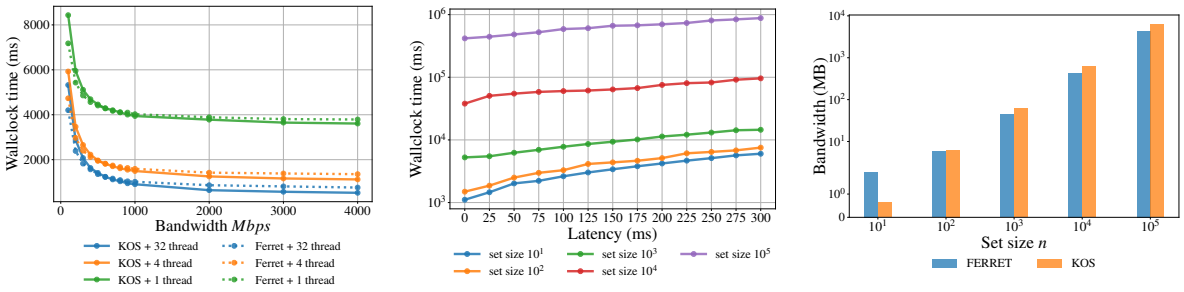
Network caching. Network caching technologies like content distribution network (CDN) are good at distributing content cheaply and quickly. This is the standard technique to distribute common website and streaming services. Our service encoding can take advantage of CDN networks since the server encoding is identical for all clients. Note that prior works on malicious PSI cannot take advantage of CDN since the communication with each client is different.

Verifiable private information retrieval. One can also use verifiable PIR [29, 18] to allow the clients getting only a subset of encodings relevant to their own PSI. Unlike normal PIR, verifiable PIR publishes a digest of the data, which ensures that anyone with the digest can verify that the PIR results are consistent with a global database, something needed to prevent attacks from a corrupted server. However, state-of-the-art verifiable PIR has a digest size of around 600MB for a database of 800MB [18] and thus the current savings are small. With more advances in their efficiency, we believe this solution could be highly valuable.

Other solutions. There are other potential solutions with some trade offs between security and efficiency. First of all, one could directly fetch the needed encodings through a TOR network to hide their identity, which requires assumptions of trusting TOR. Bucketization is another solution that provides better efficiency with reduced privacy. In detail, one can use a hash function to partition all encodings into buckets and reveal which buckets the clients are looking. Indeed, this solution has been used by Google and Cloudflare for credential checking, but there are also demonstration of attacks for various bucketization techniques [40].

7 Performance Evaluation

We implement our protocols using EMP [56] for COT and RELIC [3] for pairings. We benchmark the performance of our protocol when $\mathcal{F}_{\mathsf{COT}}$ is instantiated using KOS [37] and Ferret [58].



(a) Bandwidth vs. time (b) Latency vs. time (c) Bandwidth consumption

Figure 9: **Our performance under different network settings.** We show our performance of time consumption as bandwidth varies in (a) and as latency varies in (b). (a) uses client set size 10^3 to compare performance under different OT methods (KOS/FERRET) and different thread numbers (1/4/32). (b) takes OT method KOS to compare performance as set sizes vary. Figure (c) shows our protocol's bandwidth consumption as the set size varies.

7.1 Benchmark Setup

We instantiate everything ensuring a computational security parameter $\kappa = 128$ and a statistical security parameter $s = 40$. To this end, we use BLS12-381 for all type-III pairing operations. We show the performance in two different network settings: a LAN network with 5Gbps bandwidth and a WAN network with 120 Mbps bandwidth. All experiments are performed on AWS EC2 instances of `6a.8xlarge` type with 32vCPU and 128 GB memory.

7.2 Efficiency of Server Local Encoding

First, we benchmark the performance of the server encoding process. Note that this computation only needs to be executed once given a set of elements. Recall that this step mainly computes the VUF on the input elements. Following conventions from prior works, we hash the output to 64-bit strings, which helps in reducing the encoding size. For example, the encoding file for a set of 10^8 elements is of size 800 MB.

We prepare a list of 256-bit values in a file as the server's set. The benchmark results include the time to: 1) read all elements from the file (w/ disk access), 2) compute the VUF value of each element and then hash it into a 64-bit string, and 3) write the resulting hashes into another file (w/disk access). In Figure 8a, we show the performance of our server computation with different set sizes and threads. From the figure, we can see that the performance of the server's local encoding is linear to the set size. We observe a 3.8× improvement when increasing the threads from 1 to 4 and 15× from 1 to 32 threads. We didn't make the file I/O multi-threaded which we believe could be the bottleneck when we use 32 threads.

7.3 Efficiency of Online Computation

Now we show the performance of the interactive process between a server with a VUF secret key, and a client with a private set. As the output, the client will get VUF evaluation on its own set, which can be further used to lookup the server encoding.

Wallclock time. In Figure 8b and Figure 8c, we show the wallclock of the protocol for different client set sizes. Similarly, the time reported includes the client: 1) reading its own elements from a file, 2) running OVUF with a server to compute $F_{sk}(x_i)$; 3) computing the hash to derive 64-bit strings that can be used for local matching.

With 1 thread, the average cost for the client to process each element is $8.29ms$ in the WAN setting and $4.17ms$ in the LAN setting. With 32 threads, the average cost is $4.53ms$ in the WAN

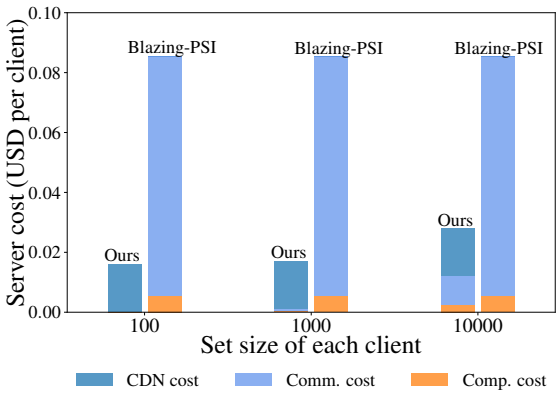


Figure 10: **Server cost comparison with Blazing-PSI [49]**. All experiments are run on AWS instance. Costs are estimated based on AWS instance pricing and network pricing.

Protocol	Security	Server comp. (s)	Offline comm. (MB)	Online time (s)	Online comm. (MB)
[38] (w/ LowMC)	Semi-honest	1869	2144	0.93	24.01
[38] (w/ ECC-NR-PSI)	Semi-honest	52332	2144	1.34	6.06
[14] (T=32)	Semi-honest	4628	0	12.1	18.57
[16] (T=24)	Semi-honest	3680	0	7.80	6.08
[50] (T=32)	Semi-honest	182	2415	0.16	0.07
Ours (w/ KOS) (T=32)	Malicious	1556.7	2147	0.44	63.23

Table 1: **Performance of unbalanced PSI with server set $|X| = 2^{28}$ and client set $|Y| = 2^{10}$.** T represents the number of threads. For the rows in context does not specify T, T equals to 1 by default.

setting and $0.43ms$ in the LAN setting. Noted that Ferret computes COT in large batches, it is not competitive when the set is small, where the protocol cannot consume all COTs. When the set size is large, our protocol in the LAN setting using KOS or Ferret does not show much difference as they have similar computational costs. In the WAN setting, we can observe a slight improvement with Ferret because it consumes less bandwidth. However, the improvement is not huge because the communication caused by our protocol, not counting the cost of COT, is already significant.

Performance dependency on network. We show the efficiency of our protocol under different network condition in Figure 9a and Figure 9b. According to Figure 9a, the efficiency of a client with a set size of 10^3 in a WAN environment increases as the bandwidth increases. However, once the bandwidth reaches $1Gbps$, the efficiency does not improve significantly with further increases in bandwidth. This indicates that our protocol performs best with bandwidth larger than $1Gbps$. In TCP networks, there is a dependency between latency and bandwidth limitations, wherein an increase in latency leads to a decrease in available bandwidth. Figure 9b illustrates that the total protocol wallclock time increases as bandwidth decreases due to added latency. For larger sets that use up more bandwidth, the rise in wallclock time is more significant than for smaller sets experiencing the same increase in latency.

Bandwidth consumption. Regarding bandwidth consumption in Figure 9c, we observed that if the set size is less than 10^2, the protocol using KOS OT performs better in terms of bandwidth usage compared to that using FERRET OT. However, this situation changes once the set size exceeds 10^2. For a set size of 10^5, the KOS OT protocol requires $61.7KB$ to process one element, while the FERRET OT protocol needs $43.0KB$ to encode one element. This is the same reason as we stated

in **Wallclock time**, that Ferret computes COT in large batches but consumes less bandwidth for each COT compared with KOS. For small set size, Ferret is more bandwidth-intensive as it generates more COT than necessary. However, for large set size, Ferret is more efficient as the generated COTs can be utilized and each one consumes less bandwidth than KOS.

7.4 Comparison with Other Protocols

Our protocol works in a special setting where a server with one set repeatedly runs PSI with many clients with small sets. We noticed that existing prior works do not perform well if used in our setting directly; this is not surprising as they are not designed for this setting. Below, we show some comparisons to state-of-the-art protocols in classical PSI settings.

Comparing with state-of-the-art PSI. The first possible solution is to use the best fully malicious secure PSI protocol [49], and have the server run this protocol with each client. However, there exists a security issue that the server might differentiate its set among different clients. Additionally, the performance is poor: each execution of PSI with a different client requires the server to transmit a different encoding of its set over the internet, which incurs great costs. In Figure 10, we compare the cost by the server per client between our protocol and the Blazing-fast [49], which is so far the fastest and improved upon VOLE-based PSI [51]. We assume the server set has 10^8 elements, and the client set ranges from 10^2 to 10^4 elements. We use the real execution time and the instance's unit price (0.1728USD/Hour for `6a.xlarge`) to compute computational cost. We also estimate the communication cost by multiplying the data size that the server transfers out by the communication unit price (0.05USD/GB). Notice that for our scheme, since the server's encoding is reusable, we use AWS CloudFront (CDN) to manage it, thereby reducing this part of the communication cost to a lower unit price (0.02USD/GB). The computation of this reusable server set encoding is a one-time and offline process, making the cost per client negligible when amortized. For our scheme, the total cost is 3x lower for a client set 10000 and 5x lower for client sets 100 and 1000. With smaller set sizes, the cost is primarily dominated by the CDN cost, which is a fixed value of 0.016 USD per client. If we switch to managing the server's encoding through a peer-to-peer network to eliminate the CDN cost, our scheme achieves an 8x reduction in communication cost and a 2x reduction in computation cost compared to Blazing-PSI for a client size of 10000. In this case, the cost of our scheme scales linearly with the client set size and performs better with smaller client sizes.

Comparing with PSI featuring reusable server encoding. Some unbalanced PSI could be better suited to our setting which allows pushing some work to the offline stage as well. In Table 1, we show how our protocol performs with related protocols:

- OPRF-based solutions by [38] allows the server to reuse its computation and encoding that is linear to X across multiple clients. We include two solutions, one based on LowMC PRF and one based on Naor–Reingold PRF. We also update their hash output to achieve a similar level of false positive rate. Our protocol runs at a similar time to OPRF-based protocols with about three times more communication; however, that allows us to achieve full malicious security.

- FHE-based solution [14, 16] does not require sending large encoding but requires more computation. The computation could be made reusable across multiple clients by performing OPRF on top of the value, but existing FHE-PSI implementations or benchmarks do not include these extra steps. We can see that our solution is much faster in terms of online time with higher communication cost. All FHE-based solutions only implement their semi-honest version and could not be made fully malicious secure; however, we do believe that by incorporating our OVUF-based solution, it is possible to achieve full malicious security as well, which we leave as future work.

- Finally, we also compare with a DH-based solution by Resende and Aranha [50]. The solution is semi-honest, but the original proposal by Jarecki and Liu [35] also includes malicious counterparts, which require further use of zero-knowledge proofs to show correct encoding. This approach essentially follows the VOPRF method, where all efficient solutions do not allow extracting client's input in the proof. As such, their solution requires much less communication.

References

[1] Navid Alamati, Pedro Branco, Nico Döttling, Sanjam Garg, Mohammad Hajiabadi, and Sihang Pu. Laconic private set intersection and applications. In Kobbi Nissim and Brent Waters, editors, *TCC 2021, Part III*, volume 13044 of *LNCS*, pages 94–125, Raleigh, NC, USA, November 8–11, 2021. Springer, Heidelberg, Germany.

[2] Martin R. Albrecht, Alex Davidson, Amit Deo, and Nigel P. Smart. Round-optimal verifiable oblivious pseudorandom functions from ideal lattices. In Juan Garay, editor, *PKC 2021, Part II*, volume 12711 of *LNCS*, pages 261–289, Virtual Event, May 10–13, 2021. Springer, Heidelberg, Germany.

[3] D. F. Aranha, C. P. L. Gouvêa, T. Markmann, R. S. Wahby, and K. Liao. RELIC is an Efficient LIbrary for Cryptography. https://github.com/relic-toolkit/relic.

[4] Diego F. Aranha, Chuanwei Lin, Claudio Orlandi, and Mark Simkin. Laconic private set-intersection from pairings. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 111–124, Los Angeles, CA, USA, November 7–11, 2022. ACM Press.

[5] Giuseppe Ateniese, Emiliano De Cristofaro, and Gene Tsudik. (If) size matters: Size-hiding private set intersection. In Dario Catalano, Nelly Fazio, Rosario Gennaro, and Antonio Nicolosi, editors, *PKC 2011*, volume 6571 of *LNCS*, pages 156–173, Taormina, Italy, March 6–9, 2011. Springer, Heidelberg, Germany.

[6] Pierre Baldi, Roberta Baronio, Emiliano De Cristofaro, Paolo Gasti, and Gene Tsudik. Countering GATTACA: efficient and secure testing of fully-sequenced human genomes. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *ACM CCS 2011*, pages 691–702, Chicago, Illinois, USA, October 17–21, 2011. ACM Press.

[7] Andrea Basso. A post-quantum round-optimal oblivious PRF from isogenies. Cryptology ePrint Archive, Report 2023/225, 2023. https://eprint.iacr.org/2023/225.

[8] Shany Ben-David, Yael Tauman Kalai, and Omer Paneth. Verifiable private information retrieval. In Eike Kiltz and Vinod Vaikuntanathan, editors, *TCC 2022, Part III*, volume 13749 of *LNCS*, pages 3–32, Chicago, IL, USA, November 7–10, 2022. Springer, Heidelberg, Germany.

[9] Dan Boneh and Xavier Boyen. Short signatures without random oracles. In Christian Cachin and Jan Camenisch, editors, *EUROCRYPT 2004*, volume 3027 of *LNCS*, pages 56–73, Interlaken, Switzerland, May 2–6, 2004. Springer, Heidelberg, Germany.

[10] Dan Boneh, Dmitry Kogan, and Katharine Woo. Oblivious pseudorandom functions from isogenies. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part II*, volume 12492 of *LNCS*, pages 520–550, Daejeon, South Korea, December 7–11, 2020. Springer, Heidelberg, Germany.

[11] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145, Las Vegas, NV, USA, October 14–17, 2001. IEEE Computer Society Press.

[12] Ran Canetti, Rosario Gennaro, Steven Goldfeder, Nikolaos Makriyannis, and Udi Peled. UC non-interactive, proactive, threshold ECDSA with identifiable aborts. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1769–1787, Virtual Event, USA, November 9–13, 2020. ACM Press.

[13] Guilhem Castagnos, Dario Catalano, Fabien Laguillaumie, Federico Savasta, and Ida Tucker. Two-party ECDSA from hash proof systems and efficient instantiations. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 191–221, Santa Barbara, CA, USA, August 18–22, 2019. Springer, Heidelberg, Germany.

[14] Hao Chen, Zhicong Huang, Kim Laine, and Peter Rindal. Labeled PSI from fully homomorphic encryption with malicious security. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 1223–1237, Toronto, ON, Canada, October 15–19, 2018. ACM Press.

[15] Hao Chen, Kim Laine, and Peter Rindal. Fast private set intersection from homomorphic encryption. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1243–1255, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press.

[16] Kelong Cong, Radames Cruz Moreno, Mariana Botelho da Gama, Wei Dai, Ilia Iliashenko, Kim Laine, and Michael Rosenberg. Labeled PSI from homomorphic encryption with reduced computation and communication. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 1135–1150, Virtual Event, Republic of Korea, November 15–19, 2021. ACM Press.

[17] Alex Davidson, Ian Goldberg, Nick Sullivan, George Tankersley, and Filippo Valsorda. Privacy pass: Bypassing internet challenges anonymously. *PoPETs*, 2018(3):164–180, July 2018.

[18] Leo de Castro and Keewoo Lee. Verisimplepir: Verifiability in simplepir at no online cost for honest servers. Cryptology ePrint Archive, Paper 2024/341, 2024. https://eprint.iacr.org/2024/341.

[19] Emiliano De Cristofaro, Jihye Kim, and Gene Tsudik. Linear-complexity private set intersection protocols secure in malicious model. In Masayuki Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 213–231, Singapore, December 5–9, 2010. Springer, Heidelberg, Germany.

[20] Daniel Demmler, Peter Rindal, Mike Rosulek, and Ni Trieu. PIR-PSI: Scaling private contact discovery. *PoPETs*, 2018(4):159–178, October 2018.

[21] Yevgeniy Dodis and Aleksandr Yampolskiy. A verifiable random function with short proofs and keys. In Serge Vaudenay, editor, *PKC 2005*, volume 3386 of *LNCS*, pages 416–431, Les Diablerets, Switzerland, January 23–26, 2005. Springer, Heidelberg, Germany.

[22] Jack Doerner, Yashvanth Kondi, Eysa Lee, and abhi shelat. Secure two-party threshold ECDSA from ECDSA assumptions. In *2018 IEEE Symposium on Security and Privacy*, pages 980–997, San Francisco, CA, USA, May 21–23, 2018. IEEE Computer Society Press.

[23] Jack Doerner, Yashvanth Kondi, Eysa Lee, and abhi shelat. Threshold ECDSA from ECDSA assumptions: The multiparty case. In *2019 IEEE Symposium on Security and Privacy*, pages 1051–1066, San Francisco, CA, USA, May 19–23, 2019. IEEE Computer Society Press.

[24] Adam Everspaugh, Rahul Chatterjee, Samuel Scott, Ari Juels, and Thomas Ristenpart. The pythia PRF service. In Jaeyeon Jung and Thorsten Holz, editors, *USENIX Security 2015*, pages 547–562, Washington, DC, USA, August 12–14, 2015. USENIX Association.

[25] Rosario Gennaro and Steven Goldfeder. Fast multiparty threshold ECDSA with fast trustless setup. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 1179–1194, Toronto, ON, Canada, October 15–19, 2018. ACM Press.

[26] Niv Gilboa. Two party RSA key generation. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 116–129, Santa Barbara, CA, USA, August 15–19, 1999. Springer, Heidelberg, Germany.

[27] Oded Goldreich. *Foundations of Cryptography: Basic Applications*, volume 2. Cambridge University Press, Cambridge, UK, 2004.

[28] Iftach Haitner, Nikolaos Makriyannis, Samuel Ranellucci, and Eliad Tsfadia. Highly efficient OT-based multiplication protocols. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part I*, volume 13275 of *LNCS*, pages 180–209, Trondheim, Norway, May 30 – June 3, 2022. Springer, Heidelberg, Germany.

[29] Alexandra Henzinger, Matthew M. Hong, Henry Corrigan-Gibbs, Sarah Meiklejohn, and Vinod Vaikuntanathan. One server for the price of two: Simple and fast single-server private information retrieval. Cryptology ePrint Archive, Report 2022/949, 2022. https://eprint.iacr.org/2022/949.

[30] Sharon Huang, Subodh Iyengar, Sundar Jeyaraman, Shiv Kushwah, Chen-Kuei Lee, Zutian Luo, Payman Mohassel, Ananth Raghunathan, Shaahid Shaikh, Yen-Chieh Sung, and Albert Zhang. DIT: De-identified Authenticated Telemetry at Scale. Technical report, Facebook Inc., 2021. https://research.fb.com/wp-content/uploads/2021/04/DIT-De-Identified-Authenticated-Telemetry-at-Scale_final.pdf.

[31] Mihaela Ion, Ben Kreuter, Ahmet Erhan Nergiz, Sarvar Patel, Mariana Raykova, Shobhit Saxena, Karn Seth, David Shanahan, and Moti Yung. On deploying secure computing commercially: Private intersection-sum protocols and their business applications. Cryptology ePrint Archive, Report 2019/723, 2019. https://eprint.iacr.org/2019/723.

[32] Stanislaw Jarecki, Aggelos Kiayias, and Hugo Krawczyk. Round-optimal password-protected secret sharing and T-PAKE in the password-only model. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part II*, volume 8874 of *LNCS*, pages 233–253, Kaoshiung, Taiwan, R.O.C., December 7–11, 2014. Springer, Heidelberg, Germany.

[33] Stanislaw Jarecki, Hugo Krawczyk, and Jason K. Resch. Updatable oblivious key management for storage systems. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 379–393, London, UK, November 11–15, 2019. ACM Press.

[34] Stanislaw Jarecki and Xiaomin Liu. Efficient oblivious pseudorandom function with applications to adaptive OT and secure computation of set intersection. In Omer Reingold, editor,

TCC 2009, volume 5444 of *LNCS*, pages 577–594. Springer, Heidelberg, Germany, March 15–17, 2009.

[35] Stanislaw Jarecki and Xiaomin Liu. Fast secure computation of set intersection. In Juan A. Garay and Roberto De Prisco, editors, *SCN 10*, volume 6280 of *LNCS*, pages 418–435, Amalfi, Italy, September 13–15, 2010. Springer, Heidelberg, Germany.

[36] Daniel Kales, Christian Rechberger, Thomas Schneider, Matthias Senker, and Christian Weinert. Mobile private contact discovery at scale. In Nadia Heninger and Patrick Traynor, editors, *USENIX Security 2019*, pages 1447–1464, Santa Clara, CA, USA, August 14–16, 2019. USENIX Association.

[37] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively secure OT extension with optimal overhead. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part I*, volume 9215 of *LNCS*, pages 724–741, Santa Barbara, CA, USA, August 16–20, 2015. Springer, Heidelberg, Germany.

[38] Ágnes Kiss, Jian Liu, Thomas Schneider, N. Asokan, and Benny Pinkas. Private set intersection for unequal set sizes with mobile applications. *PoPETs*, 2017(4):177–197, October 2017.

[39] Ben Kreuter, Tancrède Lepoint, Michele Orrù, and Mariana Raykova. Anonymous tokens with private metadata bit. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part I*, volume 12170 of *LNCS*, pages 308–336, Santa Barbara, CA, USA, August 17–21, 2020. Springer, Heidelberg, Germany.

[40] Lucy Li, Bijeeta Pal, Junade Ali, Nick Sullivan, Rahul Chatterjee, and Thomas Ristenpart. Protocols for checking compromised credentials. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 1387–1403, London, UK, November 11–15, 2019. ACM Press.

[41] Yehuda Lindell and Ariel Nof. Fast secure multiparty ECDSA with practical distributed key generation and applications to cryptocurrency custody. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 1837–1854, Toronto, ON, Canada, October 15–19, 2018. ACM Press.

[42] Peihan Miao, Sarvar Patel, Mariana Raykova, Karn Seth, and Moti Yung. Two-sided malicious security for private intersection-sum with cardinality. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part III*, volume 12172 of *LNCS*, pages 3–33, Santa Barbara, CA, USA, August 17–21, 2020. Springer, Heidelberg, Germany.

[43] Silvio Micali, Michael O. Rabin, and Salil P. Vadhan. Verifiable random functions. In *40th FOCS*, pages 120–130, New York, NY, USA, October 17–19, 1999. IEEE Computer Society Press.

[44] Shishir Nagaraja, Prateek Mittal, Chi-Yao Hong, Matthew Caesar, and Nikita Borisov. BotGrep: Finding P2P bots with structured graph analysis. In *USENIX Security 2010*, pages 95–110, Washington, DC, USA, August 11–13, 2010. USENIX Association.

[45] Moni Naor and Omer Reingold. Number-theoretic constructions of efficient pseudo-random functions. *Journal of the ACM*, 51(2):231–262, March 2004.

[46] Ofri Nevo, Ni Trieu, and Avishay Yanai. Simple, fast malicious multiparty private set intersection. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 1151–1165, Virtual Event, Republic of Korea, November 15–19, 2021. ACM Press.

[47] Bijeeta Pal, Mazharul Islam, Marina Sanusi Bohuk, Nick Sullivan, Luke Valenta, Tara Whalen, Christopher A. Wood, Thomas Ristenpart, and Rahul Chatterjee. Might I get pwned: A second generation compromised credential checking service. In Kevin R. B. Butler and Kurt Thomas, editors, *USENIX Security 2022*, pages 1831–1848, Boston, MA, USA, August 10–12, 2022. USENIX Association.

[48] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. PSI from PaXoS: Fast, malicious private set intersection. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part II*, volume 12106 of *LNCS*, pages 739–767, Zagreb, Croatia, May 10–14, 2020. Springer, Heidelberg, Germany.

[49] Srinivasan Raghuraman and Peter Rindal. Blazing fast PSI from improved OKVS and subfield VOLE. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 2505–2517, Los Angeles, CA, USA, November 7–11, 2022. ACM Press.

[50] Amanda C. Davi Resende and Diego F. Aranha. Faster unbalanced private set intersection. In Sarah Meiklejohn and Kazue Sako, editors, *FC 2018*, volume 10957 of *LNCS*, pages 203–221, Nieuwpoort, Curaçao, February 26 – March 2, 2018. Springer, Heidelberg, Germany.

[51] Peter Rindal and Phillipp Schoppmann. VOLE-PSI: Fast OPRF and circuit-PSI from vector-OLE. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part II*, volume 12697 of *LNCS*, pages 901–930, Zagreb, Croatia, October 17–21, 2021. Springer, Heidelberg, Germany.

[52] Mike Rosulek and Ni Trieu. Compact and malicious private set intersection for small sets. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 1166–1181, Virtual Event, Republic of Korea, November 15–19, 2021. ACM Press.

[53] István András Seres, Máté Horváth, and Péter Burcsi. The legendre pseudorandom function as a multivariate quadratic cryptosystem: Security and applications. Cryptology ePrint Archive, Report 2021/182, 2021. https://eprint.iacr.org/2021/182.

[54] Tjerand Silde and Martin Strand. Anonymous tokens with public metadata and applications to private contact tracing. In Ittay Eyal and Juan A. Garay, editors, *FC 2022*, volume 13411 of *LNCS*, pages 179–199, Grenada, May 2–6, 2022. Springer, Heidelberg, Germany.

[55] Nirvan Tyagi, Sofía Celi, Thomas Ristenpart, Nick Sullivan, Stefano Tessaro, and Christopher A. Wood. A fast and simple partially oblivious PRF, with applications. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part II*, volume 13276 of *LNCS*, pages 674–705, Trondheim, Norway, May 30 – June 3, 2022. Springer, Heidelberg, Germany.

[56] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit, 2016.

[57] Haiyang Xue, Man Ho Au, Xiang Xie, Tsz Hon Yuen, and Handong Cui. Efficient online-friendly two-party ECDSA signature. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 558–573, Virtual Event, Republic of Korea, November 15–19, 2021. ACM Press.

[58] Kang Yang, Chenkai Weng, Xiao Lan, Jiang Zhang, and Xiao Wang. Ferret: Fast extension for correlated OT with small communication. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1607–1626, Virtual Event, USA, November 9–13, 2020. ACM Press.

A Batched OVUF with Improved Efficiency

In this section, we give an optimized maliciously secure oblivious verifiable unpredictable protocol Π_{OVUF2} in terms of efficiency. This protocol combines a new sub-protocol $\Pi_{\mathsf{U-MtA}}$ called unbalanced imperfect multiplicative to additive shares transformation, introduced in Appendix A.1. The details and proof of Π_{OVUF2} are depicted in Appendix A.2 and Appendix A.3.

A.1 Unbalanced Imperfect MtA

This transformation computes the additive secret shares of scaler-vector multiplication, where the scaler and vector held by different parties are regarded as unbalanced input. Its imperfection follows the same idea as Section 5.2 that a malicious sender can execute attacks and result in incorrect additive secret shares depending on the receiver's input.

The most straightforward way to achieve imperfect scaler-vector multiplicative to additive shares is as follows: Given the input vector $\boldsymbol{a} \in \mathbb{Z}_q^n$ on party P_0 and scaler $b \in \mathbb{Z}_q$ on party P_1, let P_1 create a new vector $\boldsymbol{b}$ with each element $\boldsymbol{b}_i = b$. Then, both parties execute Π_{MtA}, using $\boldsymbol{a}$ and $\boldsymbol{b}$ as inputs. However, in our construction in Figure. 11, we designate P_1 as receiver of $\mathcal{F}_{\mathsf{COT}}$ and have P_1 employ $Encode(\mathsf{g}^{\mathsf{R}}, b)$. Sender P_0 inputs vector element $\boldsymbol{a}_i$ and the receiver P_1 inputs encoded bit element of b to run $\mathcal{F}_{\mathsf{COT}}$ to compute the additive secret share of $\boldsymbol{a}_i \cdot b$. This approach consumes $n(2\log q + 2s)$ iterations of $\mathcal{F}_{\mathsf{COT}}$, the same as the straightforward approach stated above. However, it saves eliminate pseudorandom vector γ of length $(n-1)\log q$ and repetitive encoding of b of length $(n-1)\log q$ when implementing the encoding algorithm.

For the incorrectness caused by the sender P_0's malicious behavior as stated in Section 5.2, it follows the same error representation as the single encoded version in Section 5.2, that $\boldsymbol{d}_i - \boldsymbol{c}_i = \boldsymbol{a}_i \cdot b + \boldsymbol{f}_i$. $\boldsymbol{f}_i$ is denote as follows with respect to $\boldsymbol{w} = Encode(\mathsf{g}^{\mathsf{R}}, b) \in \mathbb{Z}_q^{t+2s}$.

$$\boldsymbol{f}_i = \sum_{i \in [t]} \mathsf{g_i} \boldsymbol{e}_i \boldsymbol{w}_i + \sum_{i \in [2s]} \mathsf{g_{t+i}} \boldsymbol{e}_{t+i} \boldsymbol{w}_{t+i} \tag{5}$$

Given $\boldsymbol{e}$, the correctness of MtA transformation depends on $\boldsymbol{w}$ and g^{R}. Specifically, the transformation is correct when $\boldsymbol{f}_i = 0$. Still, this incorrectness will be caught by $\Pi_{\mathsf{U-MtA}}$ in Appendix A.2 with a detailed proof in Appendix A.3.

A.2 OVUF with Improved Efficiency

In Section 5, we introduced the basic version of oblivious verifiable unpredictable protocol. In the context of Π_{OVUF}, when each client holds a set of n elements $\boldsymbol{y}_i, i \in [n]$ and collaborates with a server to compute OVUF, the Π_{OVUF} processes each input element $\boldsymbol{y}_i$ one by one. This sequential processing involves n iterations of Π_{MtA}. For each iteration of Π_{MtA}, it runs with input vector $(\boldsymbol{\phi}_i, sk) \in \mathbb{Z}_q^2$ and $(\boldsymbol{y}_i, \boldsymbol{\zeta}_i) \in \mathbb{Z}_q^2$ to compute additive share of $sk \cdot \boldsymbol{\zeta}_i$ and $\boldsymbol{y}_i \cdot \boldsymbol{\phi}_i$. In this section, we maximize the batch feature of MTA protocols and execute $sk \cdot \boldsymbol{\zeta}_i$ and $\boldsymbol{y}_i \cdot \boldsymbol{\phi}_i$ for each $i \in [n]$ as follows:

1. Execute Π_{MtA} to compute additive shares of $\boldsymbol{y}_i \cdot \boldsymbol{\phi}_i, i \in [n]$ in one iteration.

2. Execute $\Pi_{\mathsf{U-MtA}}$ to efficiently compute additive shares of $\boldsymbol{\zeta}_i \cdot sk, i \in [n]$ in one iteration.

Protocol $\Pi_{\mathsf{U-MtA}}$

Inputs: P_0 holds $\boldsymbol{a} \in \mathbb{Z}_q^n$. P_1 holds $b \in \mathbb{Z}_q$.
Protocol:

1. P_1 samples $\mathsf{g}^{\mathsf{R}} \leftarrow \mathbb{Z}_q^{\log q+2s}$, and encodes b by computing $\boldsymbol{w} := \mathit{Encode}(\mathsf{g}^R, b) \in \{0,1\}^{t+2s}$.
2. P_1 inputs $\boldsymbol{w}_j, j \in [t+2s]$ to $\mathcal{F}_{\mathsf{COT}}$. P_0 inputs $\boldsymbol{a} \in \mathbb{F}_q^n$ to $\mathcal{F}_{\mathsf{COT}}$. P_0 receives $\{\boldsymbol{p}_{1,j}, \cdots, \boldsymbol{p}_{n,j}\} \in \mathbb{F}_q^n$ from $\mathcal{F}_{\mathsf{COT}}$. P_1 receives $\{\boldsymbol{q}_{1,j}, \cdots, \boldsymbol{q}_{n,j}\} \in \mathbb{F}_q^n$ from $\mathcal{F}_{\mathsf{COT}}$.
3. P_1 sends g^{R} to P_0.
4. For $j \in [t+2s], i \in [n]$, P_0 computes

$$\boldsymbol{c}_i = \sum_{j \in [t+2s]} \mathsf{g}_j \cdot \boldsymbol{p}_{i,j}$$

   P_1 computes

$$\boldsymbol{d}_i = \sum_{j \in [t+2s]} \mathsf{g}_j \cdot \boldsymbol{q}_{i,j}$$

   such that $\boldsymbol{d}_i - \boldsymbol{c}_i = \boldsymbol{a}_i \cdot b$.

Figure 11: The U-MtA protocol in $\mathcal{F}_{\mathsf{COT}}$-hybrid.

Protocol Π_{OVUF2}

Inputs: Client R holds vector $\boldsymbol{y} \in \mathbb{Z}_q^n$.
Setup:

1. If $\mathcal{F}_{\mathsf{PKI}}$ is not initialized, server S sends (init) to $\mathcal{F}_{\mathsf{PKI}}$, and obtains pair (sk, pk) that $pk = g^{sk}$.
2. Client R sends (fetch) to $\mathcal{F}_{\mathsf{PKI}}$. If $\mathcal{F}_{\mathsf{PKI}}$ is initialized, it sends pk to R; Otherwise, it does nothing.

Protocol:

1. Server S chooses $\boldsymbol{\phi} \leftarrow \mathbb{Z}_q^n$; client R chooses $\boldsymbol{\zeta} \leftarrow \mathbb{Z}_q^n$.
2. S and R inputs $\boldsymbol{\phi} \in \mathbb{Z}_q^n$ and $\boldsymbol{y} \in \mathbb{Z}_q^n$ to Π_{MtA}, receives $\boldsymbol{c} \in \mathbb{Z}_q^n$ and $\boldsymbol{d} \in \mathbb{Z}_q^n$ respectively, such that $\boldsymbol{d} - \boldsymbol{c} = \boldsymbol{\phi} \circ \boldsymbol{y}$.
3. S and R inputs sk and $\boldsymbol{\zeta} \in \mathbb{Z}_q^n$ to $\Pi_{\mathsf{U-MtA}}$, receives $\boldsymbol{z} \in \mathbb{Z}_q^n$ and $\boldsymbol{o} \in \mathbb{Z}_q^n$ respectively, such that $\boldsymbol{z} - \boldsymbol{o} = sk \cdot \boldsymbol{\zeta}$.
4. S samples $i \in [n]$, computes $V_S = H(g^{\boldsymbol{z}_i})$, and sends (i, V_S) to R. R computes $V_R = H(pk^{\boldsymbol{\zeta}_i} g^{\boldsymbol{o}_i})$. R checks whether $V_R = V_S$ and aborts if they are not equal.
5. S sends $\boldsymbol{m} = sk \cdot \boldsymbol{\phi} - \boldsymbol{c} + \boldsymbol{z}$ to R. R sends $\boldsymbol{u} = \boldsymbol{y} \circ \boldsymbol{\zeta} + \boldsymbol{d} - \boldsymbol{o}$ to S. Both S and R computes $\boldsymbol{v} = \boldsymbol{u} + \boldsymbol{m}$.
6. For each $i \in [n]$, if $\boldsymbol{v}_i \neq 0$, S computes $\frac{\boldsymbol{\phi}_i}{\boldsymbol{v}_i}$ and sends $\boldsymbol{h}_i = g^{\frac{\boldsymbol{\phi}_i}{\boldsymbol{v}_i}}$ to R. R computes $g^{\frac{\boldsymbol{\zeta}_i}{\boldsymbol{v}_i}}$ and $F_{sk}(\boldsymbol{y}_i) = \boldsymbol{h}_i \cdot g^{\frac{\boldsymbol{\zeta}_i}{\boldsymbol{v}_i}}$. Otherwise, S samples $\boldsymbol{h}_i \leftarrow \mathbb{G}$ and sends $\boldsymbol{h}_i$ to R. R sets $F_{sk}(\boldsymbol{y}_i) = \perp$.
7. R checks $e(g^{\boldsymbol{y}_i} \cdot pk, F_{sk}(\boldsymbol{y}_i)) = e(g, g)$ for each $i \in [n]$. Otherwise it aborts.

Figure 12: The OVUF2 protocol in ($\mathcal{F}_{\mathsf{COT}}$, $\mathcal{F}_{\mathsf{PKI}}$)-hybrid model with sub-protocol Π_{MtA} and $\Pi_{\mathsf{U-MtA}}$.

The other parts of this optimized-oblivious verifiable unpredictable protocol Π_{OVUF2} follow the same idea as Π_{OVUF}. The detailed scheme is shown in Figure. 12. Its correctness can be verified directly. Security-wise, this protocol involves two different MtA transformations. For $\Pi_{\mathsf{U-MtA}}$, the client R is regarded as the sender of $\mathcal{F}_{\mathsf{COT}}$ and the one who executes a selective failure attack to Π_{OVUF2}. For Π_{MtA}, we assume server S as the sender of $\mathcal{F}_{\mathsf{COT}}$ and the one who executes selective failure attacks to Π_{OVUF2} without loss of generality. We prove that Π_{OVUF2} is malicious secure in $(\mathcal{F}_{\mathsf{COT}}, \mathcal{F}_{\mathsf{PKI}})$-hybrid model with sub-protocol Π_{MtA} and $\Pi_{\mathsf{U-MtA}}$. The theorem is stated below with a full proof in Appendix A.3.

Theorem 3. *Protocol Π_{OVUF2} shown in Figure 12 UC-realizes $\mathcal{F}_{\mathsf{OVUF}}$ in the $(\mathcal{F}_{\mathsf{COT}}, \mathcal{F}_{\mathsf{PKI}})$-hybrid model with sub-protocol Π_{MtA} and $\Pi_{\mathsf{U-MtA}}$.*

A.3 Proof of Theorem 3

Proof. Let $\mathcal{A}$ be a PPT adversary that allows to corrupt the server or the client. We construct a PPT simulator $\mathcal{S}$ with access to functionality $\mathcal{F}_{\mathsf{OVUF}}$, which simulates the adversary's view. We consider the following two cases: malicious client and malicious server. We will prove that the joint distribution over the output of $\mathcal{A}$ and the honest party in the real world is indistinguishable from the joint distribution over the outputs of $\mathcal{S}$ and the honest party in the ideal world execution.

Corrupted Client. Let $\mathcal{S}$ access to the $\mathcal{F}_{\mathsf{OVUF}}$ as an honest client and interact with $\mathcal{A}$ as an honest server. $\mathcal{S}$ passes all communication between $\mathcal{A}$ and environment $\mathcal{Z}$.

0. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{PKI}}$. Once it receives fetch from $\mathcal{A}$, it samples $pk^* \leftarrow \mathbb{G}$ to $\mathcal{A}$.

1-2. $\mathcal{S}$ simulates the sub-protocol Π_{MtA} and acts as an honest sender of $\mathcal{F}_{\mathsf{COT}}$ below.

(1)-(3) $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{COT}}$ and receives $\boldsymbol{w} \in \mathbb{Z}_q^{nt+2s}$. $\mathcal{S}$ samples $\boldsymbol{q} \leftarrow \mathbb{Z}_q^{nt}$, $\boldsymbol{q}' \leftarrow \mathbb{Z}_q^{2ns}$ and sends them to $\mathcal{A}$.

(4) $\mathcal{S}$ receives g^R from $\mathcal{A}$. $\mathcal{S}$ computes $\boldsymbol{y}_i$ for each $i \in [n]$ as follows:

$$\boldsymbol{y}_i = \sum_{j\in[t]} \mathsf{g}_j \boldsymbol{w}_{(i-1)t+j} + \sum_{j\in[t+1,t+2s]} \mathsf{g}_j \boldsymbol{w}_{nt+j}$$

(5) $\mathcal{S}$ computes $\boldsymbol{d}_i$ as an honest P_1 does in step 5 in Π_{MtA}.

3. $\mathcal{S}$ simulates the sub-protocol $\Pi_{\mathsf{U-MtA}}$ and acts as an honest receiver of $\mathcal{F}_{\mathsf{COT}}$ below.

(1)-(2) $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{COT}}$. $\mathcal{S}$ receives $\boldsymbol{\tau} \in \mathbb{Z}_q^{nt+2ns}$ and sends $\boldsymbol{p} \leftarrow \mathbb{Z}_q^{nt+2ns}$ to $\mathcal{A}$. $\mathcal{S}$ checks whether the received $\boldsymbol{\tau}$ satisfy the pattern that for $i \in [n]$, all the bits $\boldsymbol{\tau}_j$, $j = (k-1)t+i, k \in [t+2s]$ are the same. Then, $\mathcal{S}$ extracts $\boldsymbol{\zeta}_i = \boldsymbol{\tau}_j$.

(3) $\mathcal{S}$ sends $\mathsf{g}^R \leftarrow \mathbb{Z}_q^{\log q+2s}$ to $\mathcal{A}$.

(4) $\mathcal{S}$ computes $\boldsymbol{o}$ as an honest P_0 does in step 4 in $\Pi_{\mathsf{U-MtA}}$.

4. $\mathcal{S}$ samples $i \in [n]$, $V_S^* \leftarrow \mathbb{G}$ to $\mathcal{A}$. $\mathcal{S}$ emulates H. Once $\boldsymbol{\zeta}_i$ is extracted in step 3 and the received query $q = (pk^{\boldsymbol{\zeta}_i} g^{\boldsymbol{o}_i})$, $\mathcal{S}$ sends V_S^* to $\mathcal{A}$. Otherwise, $\mathcal{S}$ sends a random value to $\mathcal{A}$.

5. $\mathcal{S}$ sends $(\mathsf{compute}, \boldsymbol{y})$ to $\mathcal{F}_{\mathsf{OVUF}}$ and receives set O from $\mathcal{F}_{\mathsf{OVUF}}$. For each $i \in [n]$, if the received $\boldsymbol{\tau}$ satisfy the pattern stated above and $i \in O$, $\mathcal{S}$ sends $\boldsymbol{m}_i^* = -\boldsymbol{y}_i \cdot \boldsymbol{\zeta}_i - \boldsymbol{d}_i + \boldsymbol{o}_i$ to $\mathcal{A}$. Otherwise, $\mathcal{S}$ sends $\boldsymbol{m}_i^* \leftarrow \mathbb{Z}_q$ to $\mathcal{A}$. $\mathcal{S}$ receives $\boldsymbol{u}$ from $\mathcal{A}$.

6. $\mathcal{S}$ waits to receive $F_{sk}(\boldsymbol{y}_i)$ for each $i \notin O$. For each $i \in [n]$, $\mathcal{S}$ checks whether $\boldsymbol{u}_i = \boldsymbol{y}_i \cdot \boldsymbol{\zeta}_i + \boldsymbol{d}_i - \boldsymbol{o}_i$, $\boldsymbol{\tau}$ satisfies the specific pattern and $\boldsymbol{v}_i \neq 0$, $\mathcal{S}$ simulates $\boldsymbol{h}_i^* = \frac{F_{sk}(\boldsymbol{y}_i)}{g^{\frac{\boldsymbol{\zeta}_i}{\boldsymbol{m}_i^* + \boldsymbol{u}_i}}}$ and sends it to $\mathcal{A}$. Otherwise, $\mathcal{S}$ simulates $\boldsymbol{h}_i \leftarrow \mathbb{G}$ and sends $\boldsymbol{h}$ to $\mathcal{A}$.

7. $\mathcal{S}$ aborts if $\mathcal{A}$ aborts and outputs what $\mathcal{A}$ outputs.

We are going to show the simulated execution is indistinguishable from the real protocol execution.

Hybrid $\mathcal{H}_0$. Same as real-world execution in ($\mathcal{F}_{\mathsf{PKI}}$, $\mathcal{F}_{\mathsf{COT}}$)-hybrid.

Hybrid $\mathcal{H}_1$. This hybrid is identical to $\mathcal{H}_0$ except $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{PKI}}$, $\mathcal{F}_{\mathsf{COT}}$, and simulates the messages to $\mathcal{A}$ as follows:

For step 0, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{PKI}}$, samples $pk^* \leftarrow \mathbb{G}$ and sends pk^* to $\mathcal{A}$. In hybrid $\mathcal{H}_0$, $\mathcal{F}_{\mathsf{PKI}}$ samples $sk \leftarrow \mathbb{Z}_q$ and computes $pk = g^{sk}$, which is uniformly distributed over $\mathbb{G}$. Thus, the pk^* sampled by $\mathcal{S}$ is indistinguishable from the one generated in hybrid H_0.

For step 1-2, $\mathcal{S}$ simulates sub-protocol Π_{MtA}. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{COT}}$, sends $\boldsymbol{q} \leftarrow \mathbb{Z}_q^{nt}, \boldsymbol{q}' \leftarrow \mathbb{Z}_q^{2ns}$ to $\mathcal{A}$. In Hybrid $\mathcal{H}_0$, $\boldsymbol{q}$ and $\boldsymbol{q}'$ are uniformly distributed according to $\mathcal{F}_{\mathsf{COT}}$. Thus, the $\boldsymbol{q}$ and $\boldsymbol{q}'$ sampled by $\mathcal{S}$ are indistinguishable from the those in Hybrid $\mathcal{H}_0$.

For step 3, $\mathcal{S}$ simulates sub-protocol $\Pi_{\mathsf{U-MtA}}$. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{COT}}$, sends $\boldsymbol{p} \leftarrow \mathbb{Z}_q^{nt+2ns}$ to $\mathcal{A}$. $\mathcal{S}$ also sends $\mathsf{g}^{\mathsf{R}} \leftarrow \mathbb{Z}_q^{\log q + 2s}$ to $\mathcal{A}$. In Hybrid $\mathcal{H}_0$, $\boldsymbol{p}$ is uniformly distributed over $\mathbb{Z}_q^{nt+2ns}$ according to $\mathcal{F}_{\mathsf{COT}}$. g^{R} is sampled by the client and uniformly distributed to $\mathcal{A}$. Thus, the sampled $\boldsymbol{p}$ and g^{R} are indistinguishable from Hybrid $\mathcal{H}_0$.

For step 4, $\mathcal{S}$ samples (i, V_S^*) to $\mathcal{A}$. $\mathcal{S}$ also programs random oracle H and sends V_S^* to $\mathcal{A}$ if it receives query $q = pk^{\boldsymbol{\zeta}_i} g^{\boldsymbol{o}_i}$ with $\boldsymbol{\zeta}_i$ extracted. Otherwise, $\mathcal{S}$ samples a random value to $\mathcal{A}$.

In hybrid $\mathcal{H}_0$, an honest server uses correct sk in $\Pi_{\mathsf{U-MtA}}$ and computes $V_S = H(g^{\boldsymbol{z}_i})$ for any $i \in [n]$. Since $\boldsymbol{z}_i = \boldsymbol{o}_i + sk \cdot \boldsymbol{\zeta}_i$ holds for an honest server and honest client in $\Pi_{\mathsf{U-MtA}}$, the simulated V_S^* is indistinguishable from the real V_S in Hybrid $\mathcal{H}_0$. If $\mathcal{A}$ adds error $\boldsymbol{e} \in \mathbb{Z}_q^{nt+2ns}$ in the execution of $\Pi_{\mathsf{U-MtA}}$, $\boldsymbol{z}_i = \boldsymbol{o}_i + sk \cdot \boldsymbol{\zeta}_i + \boldsymbol{f}_i$ holds for honest server and the adversary. $\boldsymbol{f}_i$ is computed from Equation 5. Since $\boldsymbol{e}$ is defined by $\mathcal{A}$ before knowing g^{R}, g^{R} is uniformly distributed over $\mathbb{Z}_q^{\log q + 2s}$ to $\mathcal{A}$. For any given $\boldsymbol{w}$ and $\boldsymbol{e}$, $\boldsymbol{f}_i$ is uniformly distributed over $\mathbb{Z}_q$ to $\mathcal{A}$. Thus, $\boldsymbol{z}_i$ is uniformly distributed over $\mathbb{Z}_q$ to $\mathcal{A}$ and the sampled V_S^* is indistinguishable from V_S in hybrid $\mathcal{H}_0$.

For step 5, $\mathcal{S}$ sends $(\mathsf{compute}, \boldsymbol{y})$ to $\mathcal{F}_{\mathsf{OVUF}}$ and waits for set O from $\mathcal{F}_{\mathsf{OVUF}}$. If $i \in O$, the received vector $\boldsymbol{\tau}$ satisfy a specified pattern stated in simulation, $\mathcal{S}$ sends $\boldsymbol{m}_i^* = -(\boldsymbol{y}_i \cdot \boldsymbol{\zeta}_i + \boldsymbol{d}_i - \boldsymbol{o}_i)$ to $\mathcal{A}$. Otherwise, $\mathcal{S}$ sends $\boldsymbol{m}_i^* \leftarrow \mathbb{Z}_q$ to $\mathcal{A}$.

In hybrid $\mathcal{H}_0$, an honest server computes $\boldsymbol{m}_i = sk \cdot \boldsymbol{\phi}_i - \boldsymbol{c}_i + \boldsymbol{z}_i$ and sends it to $\mathcal{A}$. If there exists an error $\boldsymbol{e}$ sampled by a malicious client in sub-protocol $\Pi_{\mathsf{U-MtA}}$, an honest server computes $\boldsymbol{m}_i$ such that $\boldsymbol{m}_i + \boldsymbol{y}_i \cdot \boldsymbol{\zeta}_i + \boldsymbol{d}_i - \boldsymbol{o}_i = \boldsymbol{v}_i = (sk + \boldsymbol{y}_i)(\boldsymbol{\phi}_i + \boldsymbol{\zeta}_i) + \mathsf{diff_i}$. $\mathsf{diff_i}$ resulted from incorrect $sk \cdot \boldsymbol{\zeta}_i$, computed by $\boldsymbol{f}_i$ in Equation 5. As analyzed above, $\boldsymbol{f}_i / \mathsf{diff_i}$ are uniformly distributed over $\mathbb{Z}_q$ to $\mathcal{A}$. Thus, $\boldsymbol{m}_i$ is uniformly distributed over $\mathbb{Z}_q$ to $\mathcal{A}$, which is indistinguishable from the simulated $\boldsymbol{m}_i^*$. If $\boldsymbol{\tau}$ satisfies the specific pattern and $sk + \boldsymbol{y}_i = 0$, $\boldsymbol{m}_i$ satisfy distribution that $\boldsymbol{m}_i + \boldsymbol{y}_i \cdot \boldsymbol{\zeta}_i + \boldsymbol{d}_i - \boldsymbol{o}_i = \boldsymbol{v}_i = (sk + \boldsymbol{y}_i)(\boldsymbol{\phi}_i + \boldsymbol{\zeta}_i) = 0$. The simulated $\boldsymbol{m}_i^*$ under the constraint of $i \in O (sk + \boldsymbol{y}_i = 0)$ satisfies this distribution as well. If $\boldsymbol{\tau}$ satisfies the specific pattern and $sk + \boldsymbol{y}_i \neq 0$, $\boldsymbol{m}_i + \boldsymbol{y}_i \cdot \boldsymbol{b}_i + \boldsymbol{d}_i + \boldsymbol{z}_i = \boldsymbol{v}_i = (sk + \boldsymbol{y}_i)(\boldsymbol{\phi}_i + \boldsymbol{\zeta}_i)$, because of the randomness of $\boldsymbol{\phi}_i$, $\mathcal{A}$ has no idea about the distribution of $\boldsymbol{m}_i$. Also, because $\boldsymbol{c}_i$ is randomly uniform in $\mathbb{Z}_q$ to $\mathcal{A}$, we have $\boldsymbol{m}_i$ randomly uniform in $\mathbb{Z}_q$ to $\mathcal{A}$, which is indistinguishable from the simulated $\boldsymbol{m}_i^*$. Thus, the view simulated by $\mathcal{S}$ is identical to hybrid $\mathcal{H}_0$.

For step 6, $\mathcal{S}$ checks for each $i \in [n]$, whether the received $\boldsymbol{u}_i$ is correct, $\boldsymbol{\tau}$ satisfy the specific pattern, and $\boldsymbol{v}_i \neq 0$. If it is, $\mathcal{S}$ sends $\boldsymbol{h}_i^* = \frac{F_{sk}(\boldsymbol{y}_i)}{g^{\frac{\boldsymbol{\zeta}_i}{\boldsymbol{m}_i^* + \boldsymbol{u}_i}}}$ to $\mathcal{A}$, where $F_{sk}(\boldsymbol{y}_i)$ is the value received from

$\mathcal{F}_{\mathsf{OVUF}}$ and $\boldsymbol{m}_i^*$ is the value sampled in step 4. Otherwise, $\mathcal{S}$ sends $\boldsymbol{h}_i^* \leftarrow \mathbb{G}$ to $\mathcal{A}$ instead. In hybrid $\mathcal{H}_0$, an honest server sends $\boldsymbol{h}_i = g^{\frac{\boldsymbol{\phi}_i}{\boldsymbol{m}_i+\boldsymbol{u}_i}}$ to $\mathcal{A}$, where $\boldsymbol{m}_i$ and $\boldsymbol{h}_i$ are computed honestly. If $\boldsymbol{u}_i$ is correct, $\boldsymbol{\tau}$ satisfies the specific pattern, and $\boldsymbol{v}_i \neq 0$, then $\boldsymbol{h}_i \cdot g^{\frac{\boldsymbol{\zeta}_i}{\boldsymbol{m}_i+\boldsymbol{u}_i}} = F_{sk}(\boldsymbol{y}_i)$. The view of $\boldsymbol{h}_i^*$ is identical to $\boldsymbol{h}_i$ in Hybrid $\mathcal{H}_0$. If $\boldsymbol{u}_i$ is not correct, or $\boldsymbol{\tau}$ doesn't satisfy the specific pattern in hybrid $\mathcal{H}_0$, $\boldsymbol{m}_i + \boldsymbol{u}_i = (sk + \boldsymbol{y}_i)(\boldsymbol{\phi}_i + \boldsymbol{\zeta}_i)$ if and only if $\mathsf{diff_i} = \boldsymbol{y}_i \cdot \boldsymbol{\zeta}_i + \boldsymbol{d}_i - \boldsymbol{o}_i - \boldsymbol{u}_i$. In this way, $\boldsymbol{h}_i \cdot g^{\frac{\boldsymbol{b}_i}{\boldsymbol{m}_i+\boldsymbol{u}_i}} = F_{sk}(\boldsymbol{y}_i)$. However, the probability that $\mathsf{diff_i}$ equals a specific value is negligible. In Hybrid $\mathcal{H}_1$, $\mathcal{S}$ simulates $\boldsymbol{h}_i^* \leftarrow \mathbb{G}$. $\boldsymbol{h}_i^* \cdot g^{\frac{\boldsymbol{\zeta}_i}{\boldsymbol{m}_i^*+\boldsymbol{u}_i}} = F_{sk}(\boldsymbol{y}_i)$ is negligible and indistinguishable from hybrid $\mathcal{H}_0$. If $\boldsymbol{v}_i = 0$, an honest server samples $\boldsymbol{h}_i \leftarrow \mathbb{G}$ in hybrid $\mathcal{H}_0$, which is indistinguishable from the one simulated by $\mathcal{S}$. Thus, the view simulated by $\mathcal{S}$ is identical to hybrid $\mathcal{H}_0$.

Corrupted Server. Let $\mathcal{S}$ access to the $\mathcal{F}_{\mathsf{OVUF}}$ as an honest server and interact with $\mathcal{A}$ as an honest client. $\mathcal{S}$ passes all communication between $\mathcal{A}$ and environment $\mathcal{Z}$.

0. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{PKI}}$. Once it receives first init request from $\mathcal{A}$, it samples (sk, pk), where $pk = g^{sk}$ to $\mathcal{A}$. $\mathcal{S}$ ignores subsequent init request from $\mathcal{A}$.

1-2. $\mathcal{S}$ simulates sub-protocol Π_{MtA} and acts as an honest receiver below.

(1)-(3) $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{COT}}$ and receives a vector $\boldsymbol{\tau} \in \mathbb{Z}_q^{n(t+2s)}$. $\mathcal{S}$ samples $\boldsymbol{p} \leftarrow \mathbb{Z}_q^{nt}$, $\boldsymbol{p'} \leftarrow \mathbb{Z}_q^{2ns}$ and sends them to $\mathcal{A}$. $\mathcal{S}$ checks whether the received $\boldsymbol{\tau} \in \mathbb{Z}_q^{n(t+2s)}$ satisfies a pattern that for $i \in [n]$, all the bits $\boldsymbol{\tau}_j$, $j \in [(i-1)t+1, it] \cup j = nt + i + (l-1)n$, $l \in [2s]$ are the same.

(4) $\mathcal{S}$ samples $\mathsf{g}^R \leftarrow \mathbb{Z}_q^{\log q + 2s}$ and sends it to $\mathcal{A}$.

(5) $\mathcal{S}$ computes $\boldsymbol{c}_i$ as an honest P_0 does in step 5 in Π_{MtA}.

3. $\mathcal{S}$ simulates the sub-protocol $\Pi_{\mathsf{U-MtA}}$ and acts as an honest sender below.

(1)-(2) $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{COT}}$. $\mathcal{S}$ receives $\boldsymbol{w} \in \mathbb{Z}_q^{t+2s}$ and sends $\boldsymbol{q} \leftarrow \mathbb{Z}_q^{nt+2ns}$ to $\mathcal{A}$.

(3) $\mathcal{S}$ receives $\mathsf{g}^R \in \mathbb{Z}_q^{\log q + 2s}$ from $\mathcal{A}$. $\mathcal{S}$ recover sk' as follows:

$$sk' = \sum_{i \in [2\log q + 2s]} \mathsf{g_i} \boldsymbol{w}_i$$

(4) $\mathcal{S}$ computes $\boldsymbol{z}$ as an honest P_1 does in step 4 in $\Pi_{\mathsf{U-MtA}}$.

4. $\mathcal{S}$ emulates random oracle H and receives query q from $\mathcal{A}$. $\mathcal{S}$ samples V_S to A and records (q, V_S). Once $\mathcal{S}$ receives (i, V_S) from $\mathcal{A}$, $\mathcal{S}$ first checks whether $sk' = sk$ and aborts if not. Then, $\mathcal{S}$ checks whether the corresponded $q = (g^{\boldsymbol{z}_i})$. If it is, $\mathcal{S}$ continue; Otherwise, $\mathcal{S}$ aborts.

5. $\mathcal{S}$ receives $\boldsymbol{m}$ from $\mathcal{A}$. $\mathcal{S}$ sends $(\mathsf{compute}, n)$ to $\mathcal{F}_{\mathsf{OVUF}}$ and receives set O from $\mathcal{F}_{\mathsf{OVUF}}$. If $sk = sk'$, for each $i \in [n]$, if the received $\boldsymbol{\tau}$ satisfies the pattern and $i \in O$, $\mathcal{S}$ sends $\boldsymbol{u}_i^* = -\boldsymbol{\phi}_i \cdot sk + \boldsymbol{c}_i - \boldsymbol{z}_i$ to $\mathcal{A}$. Otherwise, $\mathcal{S}$ sends $\boldsymbol{u}_i^* \leftarrow \mathbb{Z}_q$ to $\mathcal{A}$. If $sk \neq sk'$, $\mathcal{S}$ sends $\boldsymbol{u}_i^* \leftarrow \mathbb{Z}_q$ to $\mathcal{A}$ for each $i \in [n]$.

6. $\mathcal{S}$ waits to receive $\boldsymbol{h}$.

7. For each $i \in [n]$, if $sk = sk'$, $\boldsymbol{\tau}$ satisfies the pattern stated above and $i \notin O$, $\mathcal{S}$ checks whether $\boldsymbol{h}_i = g^{\frac{\boldsymbol{\phi}_i}{\boldsymbol{m}_i+\boldsymbol{u}_i^*}}$, $\boldsymbol{m}_i = \boldsymbol{\phi}_i \cdot sk - \boldsymbol{c}_i + \boldsymbol{z}_i$ and sends $\mathsf{continue}$ to $\mathcal{F}_{\mathsf{OVUF}}$. If $sk = sk'$, $\boldsymbol{\tau}$ satisfies the pattern stated above and $i \in O$, $\mathcal{S}$ checks whether $\boldsymbol{m}_i = \boldsymbol{\phi}_i \cdot sk - \boldsymbol{c}_i + \boldsymbol{z}_i$ and sends $\mathsf{continue}$ to $\mathcal{F}_{\mathsf{OVUF}}$. Otherwise, $\mathcal{S}$ sends abort to $\mathcal{F}_{\mathsf{OVUF}}$.

We are going to show the simulated execution is indistinguishable from the real protocol execution.

Hybrid $\mathcal{H}_0$. Same as real-world execution in ($\mathcal{F}_{\mathsf{PKI}}$,$\mathcal{F}_{\mathsf{COT}}$)-hybrid model.

Hybrid $\mathcal{H}_1$. This hybrid is identical to $\mathcal{H}_0$ except $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{PKI}}$, $\mathcal{F}_{\mathsf{COT}}$, and generates the messages to $\mathcal{A}$ as follows:

For Step 0, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{PKI}}$ and samples (sk, pk) to $\mathcal{A}$ once it receives the first init request from $\mathcal{A}$, which is indistinguishable from Hybrid $\mathcal{H}_0$.

For Step 1-2, $\mathcal{S}$ simulates sub-protocol Π_{MtA}. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{COT}}$, sends $\boldsymbol{p} \leftarrow \mathbb{Z}_q^{nt}, \boldsymbol{p}' \leftarrow \mathbb{Z}_q^{2ns}$ to $\mathcal{A}$. $\mathcal{S}$ also samples $\mathsf{g}^{\mathsf{R}} \leftarrow \mathbb{Z}_q^{\log q+2s}$ and sends it to $\mathcal{A}$. In Hybrid $\mathcal{H}_0$, $\boldsymbol{p}, \boldsymbol{p}'$ are uniformly distributed according to $\mathcal{F}_{\mathsf{COT}}$. g^{R} is sampled by client and uniformly distributed over $\mathbb{Z}_q^{\log q+2s}$ to $\mathcal{A}$. Thus, the simulated $\boldsymbol{p}, \boldsymbol{p}', \mathsf{g}^{\mathsf{R}}$ are indistinguishable from Hybrid $\mathcal{H}_0$.

For step 3, $\mathcal{S}$ simulates sub-protocol $\Pi_{\mathsf{U-MtA}}$. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{COT}}$, sends $\boldsymbol{q} \leftarrow \mathbb{Z}_q^{n(t+2s)}$ to $\mathcal{A}$. In Hybrid $\mathcal{H}_0$, $\boldsymbol{q}$ is uniformly distributed over $\mathbb{Z}_q^{n(t+2s)}$ according to $\mathcal{F}_{\mathsf{COT}}$. Thus, the sampled $\boldsymbol{q}$ is indistinguishable from Hybrid $\mathcal{H}_0$.

For Step 5, $\mathcal{S}$ receives $\boldsymbol{m}$ from $\mathcal{A}$, $\mathcal{S}$ sends $(\mathsf{compute}, n)$ to $\mathcal{F}_{\mathsf{OVUF}}$ and waits for set O. If $sk = sk'$, for each $i \in [n]$, if the received vector $\boldsymbol{\tau}$ in step 1-2 satisfies the stated pattern above and $i \in O$, $\mathcal{S}$ sends $\boldsymbol{u}_i^* = -\boldsymbol{\phi}_i \cdot sk + \boldsymbol{c}_i - \boldsymbol{z}_i$ to $\mathcal{A}$. Otherwise, $\mathcal{S}$ sends random $\boldsymbol{u}_i^* \leftarrow \mathbb{Z}_q$ to $\mathcal{A}$. In Hybrid $\mathcal{H}_0$, under the constraint of $sk = sk'$, if there exist error $\boldsymbol{e}$ in step 1-2, an honest client computes $\boldsymbol{u}_i$ such that $\boldsymbol{u}_i + \boldsymbol{a}_i \cdot sk - \boldsymbol{c}_i + \boldsymbol{z}_i = \boldsymbol{v}_i = (sk + \boldsymbol{y}_i)(\boldsymbol{a}_i + \boldsymbol{b}_i) + \mathsf{diff_i}$. Since $\mathsf{diff_i}$ is uniform distributed over $\mathbb{Z}_q$, $\boldsymbol{u}_i$ is uniform distributed over $\mathbb{Z}_q$ to $\mathcal{A}$ as well. If there does not exist $\boldsymbol{e}$ and $sk + \boldsymbol{y}_i \neq 0$, we have $\boldsymbol{u}_i + \boldsymbol{a}_i \cdot sk - \boldsymbol{c}_i - \boldsymbol{o}_i = \boldsymbol{v}_i = (sk + \boldsymbol{y}_i)(\boldsymbol{a}_i + \boldsymbol{b}_i)$. Since $\boldsymbol{b}_i \leftarrow \mathbb{Z}_q$ to $\mathcal{A}$, we have $\boldsymbol{u}_i$ is uniform distributed over $\mathbb{Z}_q$ to $\mathcal{A}$. If there does not exist $\boldsymbol{e}$ and $sk + \boldsymbol{y}_i = 0$, we have $\boldsymbol{u}_i + \boldsymbol{a}_i \cdot sk - \boldsymbol{c}_i - \boldsymbol{o}_i = \boldsymbol{v}_i = (sk + \boldsymbol{y}_i)(\boldsymbol{a}_i + \boldsymbol{b}_i) = 0$. Under the constraint of $sk \neq sk'$, there exists an y' that satisfy $sk' + y' = 0$ with negligible probability. $\boldsymbol{u}_i$ is uniformly distributed. Thus, the simulated $\boldsymbol{u}_i^*$ is indistinguishable from the distribution of $\boldsymbol{u}_i$ in Hybrid $\mathcal{H}_0$.

Hybrid $\mathcal{H}_2$**.** This hybrid is identical to $\mathcal{H}_1$ except $\mathcal{S}$ aborts at step 4 in the following conditions: 1) $sk' = sk$, the q corresponding to the received V_S not equal to $g^{\boldsymbol{z}_i}$ 2) $sk' \neq sk$. $\mathcal{S}$ also aborts at Step 7 in the following conditions: 1) $sk \neq sk'$; 2) vector $\boldsymbol{\tau}$ received in step 1-2 does not satisfy the specific pattern; 3) $i \in O$, $\boldsymbol{\tau}$ satisfy the specific pattern, $\boldsymbol{m}_i \neq \boldsymbol{\phi}_i \cdot sk - \boldsymbol{c}_i + \boldsymbol{z}_i$; 4) $i \notin O$, $\boldsymbol{\tau}$ satisfy the specific pattern, $\boldsymbol{m}_i \neq \boldsymbol{\phi}_i \cdot sk - \boldsymbol{c}_i + \boldsymbol{z}_i$ or $\boldsymbol{h}_i^* \neq g^{\frac{\boldsymbol{\phi}_i}{\boldsymbol{m}_i + \boldsymbol{u}_i^*}}$.

For step 4 in hybrid $\mathcal{H}_1$, an honest client aborts if the received $V_S \neq V_R$. The client computes $V_R = H(pk^{\boldsymbol{\zeta}_i} g^{\boldsymbol{o}_i})$. For each $pk^{\boldsymbol{\zeta}_i} g^{\boldsymbol{o}_i}$, it equals to $g^{sk \cdot \boldsymbol{\zeta}_i + \boldsymbol{o}_i}$. When adversary use correct sk but manipulate V_S from inconsistent $\boldsymbol{z}_i$, an honest client aborts in hybrid $\mathcal{H}_1$ (condition (1)). Adversary $\mathcal{A}$ might use inconsistent sk' to Π_{MtA} and manipulate V_S to $\mathcal{A}$. If $\mathcal{A}$ use $sk' \neq sk$ in Π_{MtA}, both parties holds equation $\boldsymbol{c}_2^i = \boldsymbol{d}_2^i - sk' \cdot \boldsymbol{\zeta}_i$. Thus, with V_S computed from $\boldsymbol{c}_2^i$, $V_S \neq V_R$. Moreover, because of the uniformity of $\boldsymbol{d}_2^i$ and $\boldsymbol{\zeta}_i$, $\mathcal{A}$ is not able to construct $\boldsymbol{c}_2^{i'}$ that $\boldsymbol{c}_2^{i'} = \boldsymbol{d}_2^i - sk \cdot \boldsymbol{\zeta}_i$ either. Thus, the client aborts with all but negligible probability with condition (2) in hybrid $\mathcal{H}_1$.

For step 7 in hybrid $\mathcal{H}_1$, an honest client aborts when $F_{sk}(\boldsymbol{y}_i)$ does not satisfy $e(g^{\boldsymbol{y}_i} \cdot pk, F_{sk}(\boldsymbol{y}_i)) = e(g, g)$, where $F_{sk}(\boldsymbol{y}_i) = \boldsymbol{h}_i \cdot g^{\frac{\boldsymbol{\zeta}_i}{\boldsymbol{m}_i + \boldsymbol{u}_i}}$. When $sk \neq sk'$, the probability that $\boldsymbol{m}_i + \boldsymbol{u}_i = (sk + \boldsymbol{y}_i)(\boldsymbol{\phi}_i + \boldsymbol{\zeta}_i)$ with negligible probability. Thus, the protocol aborts with all but negligible probability. For the ith iteration, if $\boldsymbol{\tau}$ doesn't satisfy the specific pattern, $\mathsf{diff_i} \neq 0$ with all but negligible probability. Thus, $\boldsymbol{m}_i + \boldsymbol{u}_i \neq (sk + \boldsymbol{y}_i)(\boldsymbol{\phi}_i + \boldsymbol{\zeta}_i)$ with all but negligible probability. $\boldsymbol{h}_i \cdot g^{\frac{\boldsymbol{\zeta}_i}{\boldsymbol{m}_i + \boldsymbol{u}_i}} \neq F_{sk}(\boldsymbol{y}_i)$ with all but negligible probability. The honest client aborts with all but negligible probability. When $\boldsymbol{\tau}$ satisfy the specific pattern and $i \in O$, an honest client aborts if $\boldsymbol{m}_i + \boldsymbol{u}_i \neq 0$, which results in a $F_{sk}(\boldsymbol{y}_i)$ rather than $\perp$. When $\boldsymbol{\tau}$ satisfy the specific pattern and $i \notin O$, given either wrong $\boldsymbol{m}_i$ or wrong $\boldsymbol{h}_i$, it will result in wrong $F_{sk}(\boldsymbol{y}_i)$. The honest client aborts as the computed $F_{sk}(\boldsymbol{y}_i)$ can not satisfy the verification procedure.

Therefore, this hybrid is identically distributed as the previous one.
The above hybrid argument completes this proof. □