Actively Secure 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 runs a PSI protocol with multiple clients, each with its own smaller set. In this setting, existing protocols fall short: they either achieve only semi-honest security, or else require the server to run the protocol from scratch for each execution.

We design an efficient protocol for 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 independently execute a PSI protocol with the server, with complexity linear in the size of each client's set. To learn the intersection, a client can download the server's encoding, which can be accelerated via content-distribution or peer-to-peer networks since the same encoding is used by all clients; alternatively, clients can fetch only the relevant parts of the encoding using verifiable private information retrieval. A key ingredient of our protocol is an efficient instantiation of an oblivious verifiable unpredictable function, which may be of independent interest.

Our implementation shows that our protocol is highly efficient. For a server holding 10^8 elements and each client holding 10^3 elements, the size of the server's encoding is 800MB; an execution of the protocol uses 60MB of communication, runs in under 5s in a WAN network with 120 Mbps bandwidth, and costs only 0.017 USD when utilizing network-caching infrastructures, a 5× saving compared to a state-of-the-art PSI protocol.

1 Introduction

Protocols for private set intersection (PSI) allow two parties to compute the intersection of their private sets without revealing anything else. PSI has found many applications, including genome testing [BBD+11], botnet detection [NMH+10], online advertising [IKN+20], compromised credential checking [PIB+22], contact discovery [KRS+19], etc. In many applications, one of the parties (a *server*) holds a large, fairly static set and repeatedly executes a PSI protocol with several other parties (*clients*) holding much smaller sets. This is the case, e.g., for a "password checkup" service in which the server holds a large set of compromised credentials while each client holds its own credentials and wants to find out if any have been compromised. This is also the case for contact discovery, where the server holds a large database of contact information for multiple users while clients each have their own list of contacts and want to learn which among them are in the database.

One might naively think that any (actively secure) PSI protocol could be used in the above setting. There are at least two drawbacks to doing so. First, there is a security concern: independently invoking a secure PSI protocol multiple times does not ensure that a malicious server uses

*Northwestern University, {yunqing.sun,wangxiao}@northwestern.edu
†Google and University of Maryland, jkatz2@gmail.com
‡Google, mariana@cs.columbia.edu, schoppmann@google.com

the same set in all executions. Second, even if all parties are (semi-)honest, it can be prohibitively expensive to require the server to repeatedly process its (large) input every time it runs the protocol with a new client; more preferable are solutions that allow the server to do work proportional to the size of its input *once* in an offline phase, and then repeatedly run an online phase with complexity linear in the size of a client's set.

Existing PSI protocols fall short. While many PSI protocols allow the server to re-use work done in an offline phase [KLS+17, CLR17, CHLR18, CMdG+21], existing solutions with this property do not achieve (full) security against malicious attackers. On the other hand, while several recent works have shown actively secure PSI protocols [NTY21, RT21, PRTY20, DKT10, RS21, RR22], all such protocols require the server to do work linear in the size of its input in each execution.

1.1 Our Contributions

Actively secure PSI in the client-server setting. We design an actively secure PSI protocol that is particularly suitable for the multi-client setting. In our protocol the server encodes its set once during an offline phase, and can then repeatedly execute the online phase of the protocol with multiple clients. Thus, the server's initial encoding is *reusable*, so only needs to be computed once. Moreover, clients do not need to know the server's encoding when running the online phase, but can retrieve it *asynchronously* even after the online phase is complete. This offers flexibility, potentially allowing the encoding to be distributed via content-distribution or peer-to-peer networks. (See further discussion in Section 4.2.) Alternately, the client can reduce bandwidth and retrieve only relevant portions of the encoding (after completing the online phase of the protocol) using verifiable private information retrieval [BDKP22, dCL24].

An efficient oblivious VRF. Similar to prior work, our PSI protocol relies on a subprotocol for oblivious pseudorandom function (OPRF) evaluation. To achieve security against a malicious server (as explained further in Section 2.1) we strengthen this to oblivious evaluation of a *verifiable* pseudorandom function (OVRF). (For technical reasons, it is actually more convenient for us to work with verifiable unpredictable functions, or VUFs.) While OPRFs are well-studied (see [CHL22] for a systematic summary) we are not aware of any prior work constructing (efficient) OVUFs/OVRFs. Some prior works consider verifiable OPRFs; however, efficient constructions [DGS+18, ECS+15, JKR19, KLOR20, SS22, TCR+22] are not extractable (thus not applicable here), while extractable constructions are all far from being practical [Bas24, BKW20, ADDS21, SHB23]. We also show an efficient OVUF/OVRF protocol based on the (non-oblivious) VRF of Dodis and Yampolskiy [DY05]. Our protocol relies on techniques for converting secret shares from multiplicative to additive form (aka MtA conversion), something considered by several prior works in other contexts [DKLs18, XAX+21, CCL+19]. To further improve efficiency, we rely on an "imperfect" MtA protocol that allows a cheating server to cause a client to output an incorrect result. We show that this suffices in our setting.

Practical efficiency. We implemented our PSI protocol using state-of-the-art building blocks, and our experimental results show that our protocol is highly efficient. For example, at the 128-bit computational / 40-bit statistical security level, for a server holding 10^8 elements and a client holding 10^3 elements, the size of the server's encoding is 800MB; an execution of the protocol uses 60MB of communication, runs in under 5s in a WAN network with 120 Mbps bandwidth, and costs only 0.017 USD when utilizing network caching infrastructures, a 5× saving compared to a state-of-the-art malicious PSI protocol without consistency guarantee.

1.2 Outline of the Paper

In Section 2 we give a technical overview of our PSI protocol, as well as the OVUF/OVRF sub-protocol we propose. After giving preliminary definitions in Section 3, we describe our PSI protocol in detail, based on any OVUF, in Section 4. In Section 5, we present our efficient OVUF/OVRF protocol. We conclude with an experimental evaluation, and comparison to prior work, in Section 6.

2 Technical Overview

In this section we give a more-detailed overview of our PSI protocol. Our protocol can be based on any sub-protocol for oblivious evaluation of a verifiable unpredictable function (OVUF); we provide an overview of an efficient construction of the latter as well.

2.1 Actively Secure PSI from OVUFs

As in prior work on PSI [DKT10, PRTY20, RS21], our protocol relies on the following idea: The server begins by generating a private key sk. Then, in an offline phase, the server with set $X = \{x_i\}$ computes a deterministic encoding $EX = \{ex_i = \mathsf{En}(\mathsf{sk}; x_i)\}$ of its set. To compute the intersection with a set $Y = \{y_i\}$ held by some client, the server runs an interactive protocol with the client (with complexity linear in $|Y|$) that allows the client to learn $EY = \{\mathsf{En}(\mathsf{sk}; y_i)\}$. Once the client learns EX it can compute $EX \cap EY$, from which it can deduce the elements in the intersection.

In prior work, the encoding of an element was done by setting $\mathsf{En}(\mathsf{sk}; x_i) = F_{\mathsf{sk}}(x)$ for a pseudorandom function F. The server can locally compute this encoding, while the client can compute this encoding by interacting with the server in an OPRF sub-protocol. This suffices to achieve semi-honest security: informally, the client learns nothing beyond the intersection (and the size of the server's set) since each encoding outside the intersection is random; the server learns nothing about the client's set due to the obliviousness of the OPRF protocol.

Unfortunately, the above does not appear to allow for proving security against a malicious server. In that setting, it must be possible to extract the server's (effective) input from its published encoding EX. There is no obvious way to do this in the above protocol. In particular, for a single value ex published by the server, a simulator has no way to even tell whether ex is a (correct) encoding of some element or a garbage value that will not match anything.

Malicious security via verifiability. We address this issue by using a *verifiable* random function (VRF) [MRV99]. A VRF is associated with both a private key sk and a public key pk; informally, $F_{\mathsf{sk}}(\cdot)$ should look random even given pk, but a VRF has the extra property that $x' = F_{\mathsf{sk}}(x)$ can be verified as correct given x and pk by running a verification procedure $\mathsf{Verify}(\mathsf{pk}, x, x')$. We modify the protocol given above by having the server publish pk, and setting $\mathsf{En}(\mathsf{sk}; x_i) = H(x, F_{\mathsf{sk}}(x))$ for H a hash function modeled as a random oracle. (This means we now need a sub-protocol for oblivious evaluation of a VRF, which we present in the following section.)

To see how this allows for extraction of the server's input, consider again a single encoded value ex published by the server. The simulator can look for a corresponding H-query $H(x, x')$ with output ex; if a unique such query exists then ex can only possibly correspond to x. (If there is no H-query with output ex, then the simulator knows that ex does not correspond to any element.) Crucially, the simulator can then check whether ex is indeed a (correct) encoding of x by checking whether $\mathsf{Verify}(\mathsf{pk}, x, x') = 1$.

We remark that using a VRF also allows clients to verify that their encoding EY is computed correctly during the online phase of the protocol, something that is also critical for security against a malicious server.

Finally, we observe that using a VRF is overkill, and it suffices to rely on a verifiable *unpredictable* function (VUF); we thus construct an oblivious VUF (OVUF) sub-protocol in the next section.[1]

2.2 Constructing an OVUF

Our starting point is the VUF of Dodis and Yampolskiy [DY05] based on a bilinear map $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$.[2] Let g be a generator of $\mathbb{G}$. In this VUF, the server's public key is $\mathsf{pk} = g^{\mathsf{sk}} \in \mathbb{G}$, and evaluation is defined as $F_{\mathsf{sk}}(x) = g^{1/(\mathsf{sk}+x)}$. Verification is done by checking if $e(F_{\mathsf{sk}}(x), \mathsf{pk} \cdot g^x) = e(g, g)$.

We now sketch a protocol for oblivious evaluation of this function, run between a server S holding sk and a client C holding input y. A a high level, our protocol works as follows:

1. The parties choose random values ϕ_1 and ϕ_2, respectively, viewed as an additive sharing of a random value.

2. The parties run a multiplicative-to-additive share-conversion protocol, where S uses sk and C uses ϕ_2; as a result, S and C obtain A_1 and A_2, respectively, such that $A_2 + A_1 = \phi_2 \cdot \mathsf{sk}$.

3. Similarly, the parties obtain B_1 and B_2 such that $B_2 + B_1 = \phi_1 \cdot y$.

4. S sends $\phi_1 \cdot \mathsf{sk} + A_1 + B_1$ and C sends $\phi_2 \cdot y + A_2 + B_2$. The parties then add these values to obtain $v = (\mathsf{sk} + y)(\phi_1 + \phi_2)$.

5. S sends $g^{\phi_1/v}$ to C, who computes $g^{\phi_1/v} \cdot g^{\phi_2/v} = g^{1/(\mathsf{sk}+y)}$.

Note that C can verify the final result using the verifiability property of the VUF and the server's public key.

The bottleneck in the above is the subroutine for multiplicative-to-additive (MtA) share conversion. Actively secure MtA protocols have been a key building block in the context of threshold ECDSA, and there have been proposals for constructing them using oblivious transfer (OT) [DKLs19, HMRT22], Paillier encryption [CGG+20, GG18, LN18], and Castagnos-Laguillaumie encryption [CCL+19]; see Xue et al. [XAX+21] for a more detailed survey. In this paper, we focus on constructions from OT as they are the most computationally efficient.

We adapt the MtA approach used by Doerner et al. [DKLs19] that can be viewed as a malicious version of an idea by Gilboa [Gil99]. For two parties with a and b as input, the high-level idea is to use OT to generate additive secret sharings of $a \cdot b_i$, where b_i is the ith bit of b. The two parties can then compute an additive secret sharing of $a \cdot b$ as a linear combination of the shares of the $\{ab_i\}$. To achieve security against malicious behavior, Doerner et al. made two changes: (1) each OT will select two sets of values, where the second set of values is used solely for checking correctness of the output; (2) to prevent selective-failure attacks, they encode the bit b_i as a longer string of choice bits instead of using just b_i itself. These changes lead to an overhead of 4–5× in communication as compared to the underlying semi-honest protocol.

We observe that since the final output in our application can be verified anyway, we can save half the communication by not doing checking in the MtA protocol itself. Removing the check in the MtA protocol complicates the proof of security. In particular, we are unable to define an appropriate functionality that our "imperfect" MtA sub-protocol realizes, and so instead we prove security of the entire OVUF directly.

[1] It is easy to turn a VUF F into a VRF F' in the random-oracle model by defining $F'_{\mathsf{sk}}(x) = H(F_{\mathsf{sk}}(x))$. Nevertheless, relying on a VUF provides a cleaner abstraction for our protocol. OVUFs may also be easier to construct than OVRFs.

[2] For efficiency, we use Type-III pairings where the groups in the domain are different; here, we describe things in the Type-I setting for simplicity.

It is still possible for a malicious server to cheat by using an incorrect value of sk in the protocol. This can even lead to a concrete attack: to determine whether the client's input is some value y, a malicious server can use $\mathsf{sk}^* = -y$ in step 2 of the protocol and then see whether $v = 0$ in step 4. To ensure that this does not happen, we add an extra verification step after the second step. Essentially, we want to verify that the server holds A_1 such that $A_1 + A_2 = \phi_2 \cdot \mathsf{sk}$, where A_2, ϕ_2 are known to the client and $\mathsf{sk} = \log_g \mathsf{pk}$. To do this, we have the server send g^{A_1}, and the client checks if this is equal to $\mathsf{pk}^{\phi_2} \cdot g^{-A_2}$. (When the server is honest, this does not reveal anything to the client that it did not already know; on the other hand, if the client is honest then A_2 is uniform and so a cheating server will be caught with overwhelming probability.) Using hashing, this check can be batched when evaluating the VUF at multiple points; thus, the check incurs negligible (amortized) communication and only a few exponentiations.

3 Preliminaries

We use κ as a computational security parameter and s as a statistical security parameter. We use $H_\infty(\gamma)$ to denote the min-entropy of a random variable γ; and use log to denote logarithms base 2. We let $[n] = \{1, \cdots, n\}$. Bold lowercase letters like $\boldsymbol{a}$ represent row vectors, where $\boldsymbol{a}_i$ denotes the ith component of $\boldsymbol{a}$. We also write $\boldsymbol{a} \circ \boldsymbol{b}$ for the Hadamard product of two vectors. For $b \in \mathbb{Z}_q$, we use $Bits(b)$ to denote the bit decomposition of b. We write $a \leftarrow S$ to indicate that a is sampled uniformly from set S.

3.1 Verifiable Unpredictable Functions

A verifiable random function (VRF) is a keyed function whose output is verifiable given a public key and an associated proof; informally, the output should be indistinguishable from random without the proof. A verifiable unpredictable function (VUF) is a weaker primitive, where all that is required is for the output to be unpredictable. Note, however, that in contrast to a VRF, the output of a VUF can be verified without any additional proof. We only rely on VUFs in our work.

Definition 1. *A VUF consists of algorithms* $(\mathsf{Gen}, F, \mathsf{Vrfy})$ *where*

- Gen *takes as input* 1^κ*, and outputs a key pair* $(\mathsf{sk}, \mathsf{pk})$*.*
- F *takes as input a secret key* sk *and an element* x *and outputs* y*.*
- Vrfy *takes as input a public key* pk *and elements* x, y *and outputs a bit.*

It is required that for all $(\mathsf{sk}, \mathsf{pk})$ *output by* Gen *and all* x *in the domain of* F*, we have* $\mathsf{Vrfy}(\mathsf{pk}, x, F_{\mathsf{sk}}(x)) = 1$*.*

Moreover, the following security properties hold:

Uniqueness: *There do not exist* $(\mathsf{pk}, x, y_1, y_2)$ *with* $y_1 \neq y_2$ *and*

$$\mathsf{Vrfy}(\mathsf{pk}, x, y_1) = 1 = \mathsf{Vrfy}(pk, x, y_2).$$

Unpredictability: *For any efficient algorithm* $\mathcal{A}$*, the following is negligible:*

$$Pr\left[(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^\kappa); (x, y) \leftarrow \mathcal{A}^{F_{\mathsf{sk}}(\cdot)}(\mathsf{pk}) : y = F_{\mathsf{sk}}(x)\right],$$

where $\mathcal{A}$ *does not query its oracle on* x*.*

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

There is a server S and clients $C_1, \ldots$.

Initialization: Upon receiving (init, X) from S, store X, send $|X|$ to $\mathcal{A}$, and ignore subsequent initialization requests.

Computation: Upon receiving (PSI, Y) from C_j, send $|Y|$ to $\mathcal{A}$ and S. If S sends ok and X is stored, send $X \cap Y$ to C_j. Otherwise, send $\perp$ to C_j.

Figure 1: Functionality for private set intersection.

For technical reasons, we also require that it is possible to identify whether a key pair $(\mathsf{sk}, \mathsf{pk})$ is valid or not; for a valid key pair $\mathsf{Vrfy}(\mathsf{pk}, x, F_{\mathsf{sk}}(x)) = 1$ for all x, while for an invalid key pair $\mathsf{Vrfy}(\mathsf{pk}, x, F_{\mathsf{sk}}(x)) = 0$ for all x.

We recall the VUF proposed by Dodis and Yampolskiy [DY05], based on prior work of Boneh and Boyen [BB04]. Let $\mathbb{G}$, $\mathbb{G}_T$ be cyclic groups of prime order q, with $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$ an efficiently computable pairing.[3] The Dodis-Yampolskiy VUF is defined as follows:

1. Gen: Choose $\mathsf{sk} \leftarrow \mathbb{Z}_q$ and output $(\mathsf{sk}, \mathsf{pk} = g^{\mathsf{sk}})$.
2. $F_{\mathsf{sk}}(x) = g^{1/(\mathsf{sk}+x)}$. (We define $F_{\mathsf{sk}}(-\mathsf{sk}) = 1$.)
3. $\mathsf{Vrfy}(\mathsf{pk}, x, y)$ outputs 1 iff $e(g^x \cdot \mathsf{pk}, y) = e(g, g)$, or $\mathsf{pk} = g^{-x}$ and $y = 1$.

The security of this construction for small domain relies on variations of (bilinear) Diffie-Hellman assumptions. Subsequent work [CHK+06] shows its security for general input domains in the generic group model.

3.2 Ideal Functionalities

We prove security of our protocols in the UC framework [Can01], assuming static corruptions. Below we describe the PSI functionality as well as other functionalities we use. We omit session IDs for readability. Some of our protocols rely on a programmable random oracle, which can be formalized as a functionality within the generalized UC framework [CDG+18]; we do not do this explicitly here. However, we note that our PSI functionality is explicitly defined for a single server interacting with multiple clients. We assume authenticated channels, but do not require private channels.

Private set intersection. Private set intersection (PSI) allows two parties to jointly compute the intersection of their private sets without revealing any additional information (except the sizes of their sets). In Figure 1, we describe the ideal functionality corresponding to PSI, which allows a server to compute intersections with multiple clients. The functionality ensures that the server uses the same set with every client.

Oblivious verifiable unpredictable function. One natural way to formalize an oblivious VRF is via a functionality that internally generates a random function F; when queried by a client with input (eval, x), the functionality returns $F(x)$ to the client if the server approves. Moreover, the functionality should allow any party to query (verify, x, y) to learn whether $y = F(x)$ (without notifying the server or requiring its approval). There are at least two problem with such an approach.

[3] Our implementation uses a Type-III pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ for efficiency, but for simplicity we describe our protocols using Type-I pairings.

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

There is a server S and clients $C_1, \ldots$ Let $(\mathsf{Gen}, F, \mathsf{Vrfy})$ be a VUF.

Initialization: Upon receiving $(\mathsf{init}, \mathsf{pk})$ from S, store pk, send pk to $\mathcal{A}$, and ignore subsequent initialization requests.

The queries below are ignored if pk is not stored.

Key query: Upon receiving fetch from C_j, send pk to C_j.

Evaluation: Upon receiving $(\mathsf{eval}, (y_1, \ldots, y_n))$ from C_j, send n to S and $\mathcal{A}$. When S responds with sk, check the validity of $(\mathsf{sk}, \mathsf{pk})$. If $(\mathsf{sk}, \mathsf{pk})$ is invalid, send $\perp$ to C_j. Otherwise, send $(F_{\mathsf{sk}}(y_1), \ldots, F_{\mathsf{sk}}(y_n))$ to C_j.

Figure 2: Functionality of OVUF.

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

Upon receiving $\boldsymbol{\tau} \in \mathbb{Z}_q^n$ from S and $\boldsymbol{w} \in \{0,1\}^n$ from C_j, for $i \in [n]$ choose $\boldsymbol{p}_i \leftarrow \mathbb{Z}_q$ and set $\boldsymbol{q}_i = \boldsymbol{w}_i \cdot \boldsymbol{\tau}_i - \boldsymbol{p}_i$. Send $\boldsymbol{p}$ to S and $\boldsymbol{q}$ to C_j.

Figure 3: Functionality of correlated OT.

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

There is a server S and clients $C_1, \ldots$.

Send: Upon receiving msg from S, send msg to $\mathcal{A}$ and store msg. Ignore subsequent messages from S.

Fetch: Upon receiving fetch from C_j, if msg is stored then send it to C_j.

Figure 4: Functionality of bulletin board.

First, it would need to be modified to handle a malicious server who may not choose a uniform key. While such a modification is possible, it complicates things. Second, it seems difficult to model an *unpredictable* (rather than *random*) function using this type of approach.

We therefore choose to model the OVUF functionality as a secure evaluation of a concrete VUF, as shown in Figure 2. We allow the client to request oblivious evaluation at multiple points ("batch evaluation"), as this can allow for better efficiency.

Correlated oblivious transfer. Correlated oblivious transfer (COT) is a variant of oblivious transfer. See Figure 3.

Bulletin board. We use a bulletin board functionality that allows the server to post messages that can be read by all clients. This functionality is used for distribution of the server's public key as well as the server's encoding. See Figure 4. In practice, the server's public key would be distributed through standard PKI mechanisms, and we envision that the server's encoding would be distributed through content-distribution networks.

Protocol Π_{PSI}

The server S holds X and each client C_j holds Y_j.
$H : \mathbb{Z}_q \times \mathbb{G} \to \{0,1\}^\sigma$ is a hash function.

Initialization:

1. S runs $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}$ and sends $(\mathsf{init}, \mathsf{pk})$ to $\mathcal{F}_{\mathsf{OVUF}}$.
2. For each $x_i \in X$, the server computes $ex_i = H(x_i, F_{\mathsf{sk}}(x_i))$. It then sends $EX = \{ex_i\}$ to $\mathcal{F}_{\mathsf{BB}}$.

Compute intersection:

1. C_j sends $(\mathsf{eval}, y_1, \cdots, y_n)$ to $\mathcal{F}_{\mathsf{OVUF}}$. Upon receiving n from $\mathcal{F}_{\mathsf{OVUF}}$, S sends sk to $\mathcal{F}_{\mathsf{OVUF}}$. Then $\mathcal{F}_{\mathsf{OVUF}}$ sends $(F_{\mathsf{sk}}(y_1), \cdots, F_{\mathsf{sk}}(y_n))$ to C_j. (If $\mathcal{F}_{\mathsf{OVUF}}$ sends $\perp$, then C_j aborts.)
2. C_j computes $ey_i = H(y_i, F_{\mathsf{sk}}(y_i))$ and lets $EY = \{ey_i\}$.
3. C_j sends fetch to $\mathcal{F}_{\mathsf{BB}}$, and receives EX in return. It then outputs $\{y_i : ey_i \in EX \cap EY\}$.

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

4 OVUF-based PSI

4.1 The PSI Protocol

We have already given an overview of our approach in Section 2.1. The detailed PSI protocol Π_{PSI} is shown in Figure 5.

Theorem 1. *Assume the VUF used by $\mathcal{F}_{\mathsf{OVUF}}$ is secure. If H is modeled as a random oracle, then Π_{PSI} UC-realizes $\mathcal{F}_{\mathsf{PSI}}$ in the $\{\mathcal{F}_{\mathsf{OVUF}}, \mathcal{F}_{\mathsf{BB}}\}$-hybrid model.*

Proof. Let $\mathcal{A}$ be a PPT adversary that may corrupt the server and any number of clients. We construct a simulator Sim with access to functionality $\mathcal{F}_{\mathsf{PSI}}$ that runs $\mathcal{A}$ as a subroutine. Note that there is nothing to simulate if a corrupted server interacts with a corrupted client. When an honest server interacts with a corrupted client, the only communication observed by the adversary is pk and EX; thus, that case is covered in the same way as in the case of an honest server interacting with a corrupted client.

Corrupted server with some honest clients. Sim runs $\mathcal{A}$, simulating H by returning random responses to $\mathcal{A}$'s queries. Then:

1. Let $(\mathsf{init}, \mathsf{pk})$ be the message $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{OVUF}}$, and let $EX = \{ex_i\}$ be the message $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{BB}}$.
2. Initialize $X = \emptyset$. Then for each $ex \in EX$, do:
   (a) If $\mathcal{A}$ did not make any H-query with output ex, do nothing.
   (b) If $\mathcal{A}$ made an H-query with output ex, let $H(x, x')$ be the first such query. Add x to X iff $\mathsf{Vrfy}(\mathsf{pk}, x, x') = 1$.

   Send X to $\mathcal{F}_{\mathsf{PSI}}$ on behalf of S.
3. Upon receiving n from $\mathcal{F}_{\mathsf{PSI}}$, send n to $\mathcal{A}$ on behalf of $\mathcal{F}_{\mathsf{OVUF}}$. If $\mathcal{A}$ does not respond, or responds with sk for which $(\mathsf{sk}, \mathsf{pk})$ is invalid, send abort to $\mathcal{F}_{\mathsf{PSI}}$. Otherwise, send ok to $\mathcal{F}_{\mathsf{PSI}}$.

It is not hard to see that the simulation is statistically close to an execution of Π_{PSI} in the $\{\mathcal{F}_{\mathsf{OVUF}}, \mathcal{F}_{\mathsf{BB}}\}$-hybrid world. The uniqueness of the VUF ensure that the simulator does not include wrong elements: if the adversary could find values (x, x') that pass the verification but not a VUF input-output pair, the extraction would include such an element incorrectly.

Corrupted clients with an honest server. Sim runs $\mathcal{A}$, simulating H by returning random responses to $\mathcal{A}$'s queries. Then:

1. Run $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}$. If a corrupted client queries fetch to $\mathcal{F}_{\mathsf{OVUF}}$, send pk in response.
2. Sim receives n from $\mathcal{F}_{\mathsf{PSI}}$. It chooses $ex_1, \ldots, ex_n \leftarrow \{0,1\}^\sigma$ and sets $EX = \{ex_i\}$. If any client (corrupted or not) queries $\mathcal{F}_{\mathsf{BB}}$, it sends EX in response. Sim also maintains a table T indexed by $[n]$, initially empty.
3. Whenever $\mathcal{A}$ sends $(\mathsf{eval}, y_1, \ldots, y_m)$ to $\mathcal{F}_{\mathsf{OVUF}}$ on behalf of a corrupted client, do
   (a) Send $Y = \{y_i\}$ to $\mathcal{F}_{\mathsf{PSI}}$, and receive in return a set $Z \subseteq Y$. Send $(F_{\mathsf{sk}}(y_1), \ldots, F_{\mathsf{sk}}(y_m))$ to $\mathcal{A}$.
   (b) For each $z \in Z$ do: If there is an i with $T[i] = z$, do nothing. Otherwise, choose a uniform empty entry $T[i]$, set $T[i] = z$, and program H so that $H(z, F_{\mathsf{sk}}(z)) = ex_i$.

It is again not hard to see that the simulation is statistically close to an execution of Π_{PSI} in the $\{\mathcal{F}_{\mathsf{OVUF}}, \mathcal{F}_{\mathsf{BB}}\}$-hybrid world. In particular, the simulation relies on the fact that corrupted clients do not query $(z, F_{\mathsf{sk}}(z))$ to RO ahead of time, which reduces to the unpredictability of the underlying VUF. □

4.2 Distributing the Server Encoding

Here we discuss several solutions that could be used in practice to distribute the server encoding.

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 [HHC+23, dCL24] 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 [dCL24] 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 [LPA+19].

5 An Oblivious VUF

In this section, we present an OVUF protocol for the Dodis-Yampolskiy VUF, with security against malicious adversaries. Our protocol works in the $(\mathcal{F}_{\mathsf{BB}}, \mathcal{F}_{\mathsf{COT}})$-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 B.

5.1 Encoding for Coalesced Multiplication

We provide a brief recap of the randomized encoding scheme described by Doerner et al. [DKLs19]. 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 $\mathsf{g}^{\mathsf{G}} \in \mathbb{Z}_q^{\log q}$, $\mathsf{g}_i^{\mathsf{G}} = 2^{i-1}$, and $\mathsf{g}^{\mathsf{R}} \in \mathbb{Z}_q^{\log q + 2s}$.

Algorithm 1. *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. We defer the proof to Appendix A.1. □

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

Algorithm 2. *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 within statistical distance* s^{-s} *of uniform.*

Proof. We defer the proof to Appendix A.2. □

5.2 Imperfect MtA Protocol

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. This protocol is specially designed for our efficient DY-based oblivious VUF protocol because it does not directly instantiate the MtA functionality due to the lack of correctness guarantee. Therefore, we do not model it as a functionality. The correctness will be checked for free as part of the OVUF protocol.

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 $\boldsymbol{q}_i + \boldsymbol{p}_i = a \cdot \boldsymbol{b}_i$.

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

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

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 [DKLs19]. 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 [DKLs19] 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:

1. Define $\boldsymbol{w} = Encode(\mathsf{g}^{\mathsf{R}}, b) \in \{0,1\}^{2\log q+2s}$, 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 $\boldsymbol{q}_i + \boldsymbol{p}_i = \boldsymbol{w}_i \cdot a$.

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 $\boldsymbol{w} = Encode(\mathsf{g}^{\mathsf{R}}, b) \in$

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 $\boldsymbol{w} := \mathit{BatchEncode}(\mathsf{g}^{\mathsf{R}}, \boldsymbol{b}) \in \{0,1\}^{nt+2s}$, 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 $\boldsymbol{c}_i = \sum_{j\in[t]} \mathsf{g}_j \cdot \boldsymbol{p}_{i,j} + \sum_{k\in[2s]} \mathsf{g}_{t+k} \cdot \boldsymbol{p}'_{i,k}$ and 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 the $\mathcal{F}_{\mathsf{COT}}$-hybrid model.

$\mathbb{Z}_q^{t+2s}$. 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

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

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

5.3 An Oblivious VUF from Imperfect MtA

Based on the imperfect MtA protocol from Section 5.2, we construct an OVUF protocol as follows.

1. The server first uses pk to initiate the bulletin board $\mathcal{F}_{\mathsf{BB}}$. Then, upon receiving pk from $\mathcal{F}_{\mathsf{BB}}$, the client checks whether $g^{-y_i} = \mathsf{pk}$ for each y_i in its input set $(y_1, \cdots, y_n)$. If so, the client replaces these y_i values with random values to avoid corner cases in the protocol.
2. Given input value sk on the server side and input vector $(y_1, \cdots, y_n) \in \mathbb{Z}_q^n$ on the client side, both parties uniformly choose random vectors $\boldsymbol{\phi}_1 \in \mathbb{Z}_q^n$ and $\boldsymbol{\phi}_2 \in \mathbb{Z}_q^n$ respectively.
3. The inputs and random vectors are specifically ordered as $(\mathsf{sk}, \boldsymbol{\phi}_1^i) \in \mathbb{Z}_q^2, i \in [n]$ and $(\boldsymbol{\phi}_2^i, y_i) \in \mathbb{Z}_q^2, i \in [n]$, 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 A_1^i and B_1^i of $\mathsf{sk} \cdot \boldsymbol{\phi}_2^i$ and additive secret shares A_2^i and B_2^i of $\boldsymbol{\phi}_1^i \cdot y_i$.
4. Then, the server raise g to A_1^i for each $i \in [n]$, where A_1^i is the secret share of $\mathsf{sk} \cdot \boldsymbol{\phi}_2^i$, and apply a hash function to these values. The server sends the hash result to the client, allowing it to check whether sk used by the server in each iteration is consistent with the pk initialized on $\mathcal{F}_{\mathsf{BB}}$.
5. Then, both parties are able to locally compute $\mathsf{sk} \cdot \boldsymbol{\phi}_1^i + A_1^i + B_1^i$ and $\boldsymbol{\phi}_2^i \cdot y_i + A_2^i + B_2^i$, respectively. The results are regarded as secret shares of $\boldsymbol{v}_i = (\boldsymbol{\phi}_1^i + \boldsymbol{\phi}_2^i)(\mathsf{sk} + y_i)$. Both parties exchanges the results to recover $\boldsymbol{v}_i$.

Protocol Π_{OVUF}

Inputs and parameters: Hash function H modeled as RO. Client C_j holds $(y_1, \ldots, y_n) \in \mathbb{Z}_q^n$.
Initialization: S chooses $\mathsf{sk} \in \mathbb{Z}_q$, sets $\mathsf{pk} = g^{\mathsf{sk}}$, and sends pk to $\mathcal{F}_{\mathsf{BB}}$.
Key query: Client C_j sends fetch to $\mathcal{F}_{\mathsf{BB}}$ and receives pk.
Evaluation:

1. C_j checks if $g^{-y_i} = \mathsf{pk}$ for each $i \in [n]$. If it is, C_j inserts i to set I and sample uniform $y_i \leftarrow \mathbb{Z}_q$.
2. Server S chooses $\boldsymbol{\phi}_1 \leftarrow \mathbb{Z}_q^n$; client C_j chooses $\boldsymbol{\phi}_2 \leftarrow \mathbb{Z}_q^n$.
3. For $i \in [n]$, S holds vector $(\mathsf{sk}, \boldsymbol{\phi}_1^i) \in \mathbb{Z}_q^2$, C_j holds vector $(\boldsymbol{\phi}_2^i, y_i) \in \mathbb{Z}_q^2$. Both parties run Π_{MtA} with the stated input vector above. Then, S receives $(A_1^i, B_1^i) \in \mathbb{Z}_q^2$, C_j receives $(A_2^i, B_2^i) \in \mathbb{Z}_q^2$, such that $A_2^i + A_1^i = \boldsymbol{\phi}_2^i \cdot \mathsf{sk}$, $B_2^i + B_1^i = \boldsymbol{\phi}_1^i \cdot y_i$.
4. S computes $V_S = H(g^{A_1^1}, \cdots, g^{A_1^n})$, and sends V_S to C_j. C_j computes $V_R = H(\mathsf{pk}^{\boldsymbol{\phi}_2^1}/g^{A_2^1}, \cdots, \mathsf{pk}^{\boldsymbol{\phi}_2^n}/g^{A_2^n})$. C_j checks whether $V_R = V_S$ and aborts if they are not equal.
5. S sends $\boldsymbol{m}$ to C_j, where $\boldsymbol{m}_i = \boldsymbol{\phi}_1^i \cdot \mathsf{sk} + A_1^i + B_1^i$. C_j computes $\boldsymbol{u}_i = \boldsymbol{\phi}_2^i \cdot y_i + A_2^i + B_2^i$ and sends it to S. Both parties compute $\boldsymbol{v} = \boldsymbol{m} + \boldsymbol{u}$.
6. For each $i \in [n]$, S computes $\boldsymbol{h}_i = g^{\boldsymbol{\phi}_1^i/\boldsymbol{v}_i}$. Then S sends $\boldsymbol{h}$ to C_j. C_j sets $F_{\mathsf{sk}}(y_i) = 1, i \in I$. For each $i \in [n] \setminus I$, C_j computes $F_{\mathsf{sk}}(y_i) = \boldsymbol{h}_i \cdot g^{\boldsymbol{\phi}_2^i/\boldsymbol{v}_i}$.
7. C_j outputs $(F_{\mathsf{sk}}(y_1), \cdots, F_{\mathsf{sk}}(y_n))$ if $e(g^{y_i} \cdot \mathsf{pk}, F_{\mathsf{sk}}(y_i)) = e(g, g)$ for each $i \in [n] \setminus I$. Otherwise it aborts.

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

6. Both parties are able to compute $g^{\boldsymbol{\phi}_1^i/\boldsymbol{v}_i}$ and $g^{\boldsymbol{\phi}_2^i/\boldsymbol{v}_i}$, respectively. Given $g^{\boldsymbol{\phi}_1^i/\boldsymbol{v}_i}$, the client computes $F_{\mathsf{sk}}(y_i) = g^{\boldsymbol{\phi}_1^i/\boldsymbol{v}_i} \cdot g^{\boldsymbol{\phi}_2^i/\boldsymbol{v}_i}$ and verifies correctness of the protocol using the fetched pk.

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 y_i or $\boldsymbol{u}_i$ to the server. Incorrect y_i can be extracted by Sim given g^{R} from the client. Incorrect $\boldsymbol{u}_i$ leads to abort with all but negligible probability, which can be simulated by Sim 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 $\boldsymbol{u}_i + \boldsymbol{m}_i = \mathsf{diff}_{\mathsf{i}} + (\boldsymbol{\phi}_1^i + \boldsymbol{\phi}_2^i)(\mathsf{sk} + y_i) = \mathsf{diff}_{\mathsf{i}} + \boldsymbol{v}_i$. $\mathsf{diff}_{\mathsf{i}}$ 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 $\boldsymbol{\phi}_1^i \cdot y_i$ and $\boldsymbol{f}_2^i$ resulted from incorrect $\mathsf{sk} \cdot \boldsymbol{\phi}_2^i$. 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 server sends $\boldsymbol{m}_i$ and $\boldsymbol{h}_i$ honestly, the verification of $F_{\mathsf{sk}}(y_i)$ passes if and only if $\mathsf{diff}_{\mathsf{i}} = 0$, which is with negligible probability. If not, the verification of $F_{\mathsf{sk}}(y_i)$ passes if and only if diff equals a specific number that results in correct $F_{\mathsf{sk}}(y_i)$, which is negligible either. Thus, the server's malicious behavior can be simulated by Sim 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 $(\mathcal{F}_{\mathsf{BB}}, \mathcal{F}_{\mathsf{COT}})$ is shown in Theorem 2.

Theorem 2. *If H is modeled as a random oracle, then protocol Π_{OVUF} with sub-protocol Π_{MtA} shown in Figure 7 UC-realizes $\mathcal{F}_{\mathsf{OVUF}}$ in $(\mathcal{F}_{\mathsf{BB}}, \mathcal{F}_{\mathsf{COT}})$-hybrid model.*

Proof. Let $\mathcal{A}$ be a PPT adversary that allows to corrupt the server or the client. We construct a PPT simulator Sim 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 Sim and the honest party in the ideal world execution.

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

0. Sim emulates $\mathcal{F}_{\mathsf{BB}}$, once it receives fetch from $\mathcal{A}$. Sim sends fetch to $\mathcal{F}_{\mathsf{OVUF}}$ and receives pk. Sim sends pk to $\mathcal{A}$.

2-3. For $i \in [n]$, Sim simulates the ith iteration of sub-protocol Π_{MtA} below.

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

(4) Sim receives g^R from $\mathcal{A}$. Sim computes $\boldsymbol{\phi}_2^i$ and y_i as follows:

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

(5) Sim computes A_2^i, B_2^i as an honest P_1 does in step 5 in Π_{MtA}.

4. Sim samples V_S^* and sends it to $\mathcal{A}$. Sim emulates H and receives query q from $\mathcal{A}$. If $q = (\mathsf{pk}^{\phi_2^1}/g^{A_2^1}, \cdots, \mathsf{pk}^{\phi_2^n}/g^{A_2^n})$, Sim sends $V_R^* = V_S^*$ to $\mathcal{A}$. Otherwise, Sim uniformly samples V_R^* and sends it to $\mathcal{A}$. Sim aborts if $\mathcal{A}$ aborts.

5. Sim sends $\boldsymbol{m}_i^* \leftarrow \mathbb{Z}_q$ to $\mathcal{A}$. Sim receives $\boldsymbol{u}$ from $\mathcal{A}$. Sim computes $\boldsymbol{v}$.

6. Sim sends $(\mathsf{eval}, (y_1, \cdots, y_n))$ to $\mathcal{F}_{\mathsf{OVUF}}$ and waits to receive $(F_{\mathsf{sk}}(y_1), \cdots, F_{\mathsf{sk}}(y_n))$. For each $i \in [n]$, Sim checks whether $\boldsymbol{u}_i = y_i \cdot \boldsymbol{\phi}_2^i + A_2^i + B_2^i$. If it is, Sim simulates $\boldsymbol{h}_i^* = \frac{F_{\mathsf{sk}}(y_i)}{g^{\boldsymbol{\phi}_2^i/\boldsymbol{v}_i}}$, and sends it to $\mathcal{A}$. Otherwise, Sim simulates $\boldsymbol{h}_i^* \leftarrow \mathbb{G}$ and sends it to $\mathcal{A}$.

7. Sim 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 the $(\mathcal{F}_{\mathsf{COT}}, \mathcal{F}_{\mathsf{BB}})$-hybrid model.

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

For $\mathsf{step\ 0}$, Sim emulates $\mathcal{F}_{\mathsf{BB}}$. Upon receiving fetch from $\mathcal{A}$, Sim sends fetch to $\mathcal{F}_{\mathsf{OVUF}}$ and receives pk. Sim sends pk to $\mathcal{A}$. In hybrid $\mathcal{H}_0$, $\mathcal{F}_{\mathsf{BB}}$ was initialized by an honest server with pk and sends it to $\mathcal{A}$ upon receiving fetch. Thus, the pk sends by Sim is same as the one in hybrid $\mathcal{H}_0$.

For $\mathsf{step\ 2\text{-}3}$, Sim emulates $\mathcal{F}_{\mathsf{COT}}$, receives $\boldsymbol{w} \in \mathbb{Z}_q^{2t+2s}$ from $\mathcal{A}$. Sim 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 Sim is indistinguishable from the one in Hybrid $\mathcal{H}_0$.

For $\mathsf{step\ 4}$, Sim samples V_S^* to $\mathcal{A}$. Then, Sim emulates random oracle and returns $V_R^* = V_S^*$ to $\mathcal{A}$ for query $q = (\mathsf{pk}^{\phi_2^1}/g^{A_2^1}, \cdots, \mathsf{pk}^{\phi_2^n}/g^{A_2^n})$, where A_2^i and $\boldsymbol{\phi}_2^i$ are recovered by Sim. For other queries, Sim samples V_R^* to $\mathcal{A}$. In hybrid $\mathcal{H}_0$, an honest server uses sk that corresponding to pk in Π_{MtA} and computes $V_S = H(g^{A_1^1}, \cdots, g^{A_1^n})$. Since $A_1^i + A_2^i = \mathsf{sk} \cdot \boldsymbol{\phi}_2^i$ holds for an honest server, the received

V_S equals to V_R if V_R is computed from $(\mathsf{pk}^{\phi_2^1}/g^{A_2^1}, \cdots, \mathsf{pk}^{\phi_2^n}/g^{A_2^n})$ honestly. Thus, the simulated V_R^* equals to V_S^* if $\mathcal{A}$ query random oracle honestly, which is indistinguishable from Hybrid $\mathcal{H}_0$.

For step 5, Sim sends random $\boldsymbol{m}_i^* \leftarrow \mathbb{Z}_q$ to $\mathcal{A}$. In hybrid $\mathcal{H}_0$, an honest server computes $\boldsymbol{m}_i = \boldsymbol{\phi}_1^i \cdot \mathsf{sk} + A_1^i + B_1^i$ and sends it to $\mathcal{A}$. $\boldsymbol{m}_i$ satisfies the distribution that $\boldsymbol{m}_i + y_i \cdot \boldsymbol{\phi}_2^i + A_2^i + B_2^i = \boldsymbol{v}_i = (\boldsymbol{\phi}_1^i + \boldsymbol{\phi}_2^i)(\mathsf{sk} + y_i)$. Since $\boldsymbol{\phi}_1^i$ is randomly sampled by an honest server, $\mathcal{A}$ has no idea about the distribution of $\boldsymbol{m}_i$. 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 Sim is identical to hybrid $\mathcal{H}_0$.

For step 6, Sim sends $(\mathsf{eval}, (y_1, \cdots, y_n))$ to $\mathcal{F}_{\mathsf{OVUF}}$ and waits for $(F_{\mathsf{sk}}(y_1), \cdots, F_{\mathsf{sk}}(y_n))$. Sim checks whether the received $\boldsymbol{u}_i$ is computed from $y_i \cdot \boldsymbol{\phi}_2^i + A_2^i + B_2^i$ correctly. If it is, Sim sends $\boldsymbol{h}_i^* = \frac{F_{\mathsf{sk}}(y_i)}{g^{\boldsymbol{\phi}_2^i/(\boldsymbol{m}_i^* + \boldsymbol{u}_i)}}$ to $\mathcal{A}$, such that $\boldsymbol{h}_i^* \cdot g^{\boldsymbol{\phi}_2^i/(\boldsymbol{m}_i^* + \boldsymbol{u}_i)} = F_{\mathsf{sk}}(y_i)$. Otherwise, Sim samples $\boldsymbol{h}_i^* \leftarrow \mathbb{G}$ and sends it to $\mathcal{A}$. In hybrid $\mathcal{H}_0$, an honest server sends $\boldsymbol{h}_i = g^{\boldsymbol{\phi}_1^i/(\boldsymbol{m}_i + \boldsymbol{u}_i)}$ to $\mathcal{A}$. If $\boldsymbol{u}_i$ is computed honestly from $y_i \cdot \boldsymbol{\phi}_2^i + A_2^i + B_2^i$, then $\boldsymbol{h}_i \cdot g^{\boldsymbol{\phi}_2^i/(\boldsymbol{m}_i + \boldsymbol{u}_i)} = F_{\mathsf{sk}}(y_i)$. Thus, the view of $\boldsymbol{h}_i^*$ in hybrid $\mathcal{H}_0$ is identical to $\boldsymbol{h}_i$ in Hybrid $\mathcal{H}_0$.If $\boldsymbol{u}_i$ is not computed honestly , then $\boldsymbol{m}_i + \boldsymbol{u}_i \neq (\boldsymbol{\phi}_1^i + \boldsymbol{\phi}_2^i)(\mathsf{sk} + y_i)$ and thus $\boldsymbol{h}_i \cdot g^{\boldsymbol{\phi}_2^i/(\boldsymbol{m}_i + \boldsymbol{u}_i)} \neq F_{\mathsf{sk}}(y_i)$. In hybrid $\mathcal{H}_1$, Sim simulates $\boldsymbol{h}_i^* \leftarrow \mathbb{G}$, we have $\boldsymbol{h}_i^* \cdot g^{\boldsymbol{\phi}_2^i/(\boldsymbol{m}_i^* + \boldsymbol{u}_i)} = F_{\mathsf{sk}}(y_i)$ w.p. $2^{-\log q}$, which is indistinguishable from hybrid $\mathcal{H}_0$. Thus, the view simulated by Sim is identical to hybrid $\mathcal{H}_0$.

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

0. Sim emulates $\mathcal{F}_{\mathsf{BB}}$, once it receives the pk from $\mathcal{A}$, Sim stores pk and ignores subsequent messages from $\mathcal{A}$. Sim sends $(\mathsf{init}, \mathsf{pk})$ to $\mathcal{F}_{\mathsf{OVUF}}$.

2-3. Upon receiving n from $\mathcal{F}_{\mathsf{OVUF}}$, Sim simulates iterations of Π_{MtA} for each $i \in [n]$. The ith iteration of Π_{MtA} is simulated as follows:

(1)-(3) Sim emulates $\mathcal{F}_{\mathsf{COT}}$ and receives a vector $\boldsymbol{\tau} \in \mathbb{Z}_q^{2(t+2s)}$ from $\mathcal{A}$. Sim samples $\boldsymbol{p} \leftarrow \mathbb{Z}_q^{2t}$, $\boldsymbol{p'} \leftarrow \mathbb{Z}_q^{4s}$ and sends them to $\mathcal{A}$. Sim 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, Sim extracts $\mathsf{sk'} = \boldsymbol{\tau}_j$. For $k = 2$, if $\boldsymbol{\tau}_j$ are the same, Sim extracts $\boldsymbol{\phi}_1^i = \boldsymbol{\tau}_j$.

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

(5) Sim computes A_1^i, B_1^i as an honest P_0 does in step 5 in Π_{MtA}.

4. Sim emulates random oracle H and receives query q from $\mathcal{A}$. Sim samples V_S to $\mathcal{A}$ and records (q, V_S). Once Sim receives V_S from $\mathcal{A}$, Sim first checks whether $\mathsf{sk'}$s have been extracted and are the same in last step. Sim also checks whether $g^{\mathsf{sk'}} = \mathsf{pk}$. Then, Sim checks if the corresponded $q = (g^{A_1^1}, \cdots, g^{A_1^n})$. If all these requirements are satisfied, Sim continue; Otherwise, Sim aborts.

5. Sim receives $\boldsymbol{m}$ from $\mathcal{A}$. Sim samples $\boldsymbol{u}^* \leftarrow \mathbb{Z}_q^n$ to $\mathcal{A}$. Sim computes $\boldsymbol{v}^* = \boldsymbol{m} + \boldsymbol{u}^*$.

6. Sim waits to receive $\boldsymbol{h}$.

7. For each ith iteration, if $\boldsymbol{\phi}_1^i$ is extracted, Sim checks whether $\boldsymbol{h}_i = g^{\boldsymbol{\phi}_1^i/\boldsymbol{v}_i^*}$, $\boldsymbol{m}_i = \boldsymbol{\phi}_1^i \cdot \mathsf{sk'} + A_1^i + B_1^i$ and sends $\mathsf{sk'}$ to $\mathcal{F}_{\mathsf{OVUF}}$. Otherwise, Sim aborts.

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 ($\mathcal{F}_{\mathsf{BB}}$,$\mathcal{F}_{\mathsf{COT}}$)-hybrid model.

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

For Step 2-3, Sim emulates $\mathcal{F}_{\mathsf{COT}}$ and waits to receive $\boldsymbol{\tau}$. Then, Sim sends $\boldsymbol{p} \leftarrow \mathbb{Z}_q^{2t}$ and $\boldsymbol{p}' \leftarrow \mathbb{Z}_q^{4s}$ to $\mathcal{A}$. Sim 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 Step 5, Sim receives $\boldsymbol{m}$ and samples $\boldsymbol{u}_i^* \leftarrow \mathbb{Z}_q$ to $\mathcal{A}$. In the hybrid $\mathcal{H}_0$, for ith iteration, if $\mathcal{A}$ sends $\boldsymbol{\tau}$ correctly , $\boldsymbol{u}_i + \boldsymbol{\phi}_1^i \cdot \mathsf{sk} + A_1^i + B_1^i = \boldsymbol{v}_i = (\boldsymbol{\phi}_1^i + \boldsymbol{\phi}_2^i)(\mathsf{sk} + y_i)$, which is uniformly distributed over $\mathbb{Z}_q$. Thus, $\boldsymbol{u}_i$ is uniformly distributed, same as the sampled one in this hybrid. If there exists an error $\boldsymbol{e}$ sampled by $\mathcal{A}$, an honest client computes $\boldsymbol{u}_i$ such that $\boldsymbol{u}_i + \boldsymbol{\phi}_1^i \cdot \mathsf{sk} + A_1^i + B_1^i = \boldsymbol{v}_i = (\boldsymbol{\phi}_1^i + \boldsymbol{\phi}_2^i)(\mathsf{sk} + y_i) + \mathsf{diff}_{\mathsf{i}}$. $\mathsf{diff}_{\mathsf{i}} = \boldsymbol{f}_1^i + \boldsymbol{f}_2^i$, where $\boldsymbol{f}_1^i$ resulted from incorrect $\boldsymbol{\phi}_1^i \cdot y_i$ and $\boldsymbol{f}_2^i$ resulted from incorrect $\mathsf{sk} \cdot \boldsymbol{\phi}_2^i$ as stated in Equation 1. Since $\boldsymbol{e}$ is defined by $\mathcal{A}$ before knowing g^{R}, g^{R} appears uniformly random 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 $\mathbb{Z}_q$. Therefore, $\mathsf{diff}_{\mathsf{i}}$ is uniformly distributed over $\mathbb{Z}_q$. $\boldsymbol{u}_i$ is uniformly distributed over $\mathbb{Z}_q$, which is identically distributed as the simulated $\boldsymbol{u}_i^*$ in this hybrid.

Hybrid $\mathcal{H}_2$**.** This hybrid is identical to $\mathcal{H}_1$ except Sim aborts at Step 4 in the following conditions: 1) any sk' in Π_{MtA} iterations is not extractable; 2) the extracted sk's are not the same or any $g^{\mathsf{sk}'} \neq \mathsf{pk}$; 3) the q corresponding to the received V_S is not equal to $(g^{A_1^1}, \ldots, g^{A_1^n})$. Sim aborts at Step 7 in the following conditions: 1) any $\boldsymbol{\phi}_1^i$ in Π_{MtA} iterations is not extractable; 2) $\boldsymbol{h}_i^* \neq g^{\boldsymbol{\phi}_1^i/(\boldsymbol{m}_i + \boldsymbol{u}_i^*)}$ or $\boldsymbol{m}_i \neq \boldsymbol{\phi}_1^i \cdot \mathsf{sk} + A_1^i + B_1^i$.

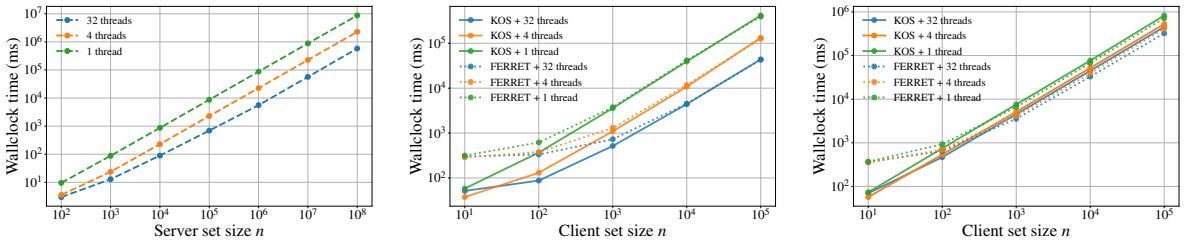
For Step 4 in hybrid $\mathcal{H}_1$, an honest client aborts if the received $V_S \neq V_R$. The client computes $V_R = H(\mathsf{pk}^{\boldsymbol{\phi}_2^1}/g^{A_2^1}, \ldots, \mathsf{pk}^{\boldsymbol{\phi}_2^n}/g^{A_2^n})$. For each $\mathsf{pk}^{\boldsymbol{\phi}_2^i}/g^{A_2^i}$, it equals to $g^{\mathsf{sk} \cdot \boldsymbol{\phi}_2^i - A_2^i}$. 1) Adversary $\mathcal{A}$ might add error $\boldsymbol{e}$ to $\boldsymbol{\tau}$ on bits related to sk in one iteration of Π_{MtA}. In this case, sk' is unextractable. If we set a value as sk', then both parties holds equation $A_1^i + A_2^i = \mathsf{sk}' \cdot \boldsymbol{\phi}_2^i + \boldsymbol{f}_2$, where $\boldsymbol{f}_2$ is computed according to Equation. 1. 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 A_1^i, $V_S \neq V_R$. Furthermore, even $\mathcal{A}$ tries to manipulate V_S, $\mathcal{A}$ is unable to construct $A_1^{i'} = \mathsf{sk} \cdot \boldsymbol{\phi}_2^i - A_2^i$ as $\boldsymbol{\phi}_2^i$ and A_2^i are uniformly distributed. Thus, the client aborts with all but negligible probability, which is indistinguishable from condition (1) in hybrid $\mathcal{H}_2$. 2) Adversary $\mathcal{A}$ might use inconsistent sk in different Π_{MtA} iterations. If $\mathcal{A}$ use $\mathsf{sk}' \neq \mathsf{sk}$ in Π_{MtA}, both parties holds equation $A_1^i + A_2^i = \mathsf{sk}' \cdot \boldsymbol{\phi}_2^i$. Thus, with V_S computed from A_1^i, $V_S \neq V_R$. $\mathcal{A}$ is not able to construct $A_1^{i'}$ to manipulate V_S either. Thus, the client aborts with all but negligible probability, which is indistinguishable from condition (2) in hybrid $\mathcal{H}_2$. 3) When adversary use correct sk but manipulate V_S from inconsistent q, an honest client aborts which is indistinguishable from condition (3) in hybrid $\mathcal{H}_2$.

For Step 7 in hybrid $\mathcal{H}_1$, an honest client aborts when $F_{\mathsf{sk}}(y_i)$ does not satisfy $e(g^{y_i} \cdot pk, F_{\mathsf{sk}}(y_i)) = e(g, g)$, where $F_{\mathsf{sk}}(y_i) = \boldsymbol{h}_i \cdot g^{\boldsymbol{\phi}_2^i/(\boldsymbol{m}_i + \boldsymbol{u}_i)}$. 1) For the ith iteration of Π_{MtA}, if adversary adds error $\boldsymbol{e}$ to $\boldsymbol{\tau}$ such that ϕ_1^i is not extractable, then $\mathsf{diff}_{\mathsf{i}} \neq 0$ with all but negligible probability. Thus, $\boldsymbol{m}_i + \boldsymbol{u}_i \neq (\mathsf{sk} + y_i)(\boldsymbol{\phi}_1^i + \boldsymbol{\phi}_2^i)$ with all but negligible probability. Consequently, $\boldsymbol{h}_i \cdot g^{\boldsymbol{\phi}_2^i/(\boldsymbol{m}_i + \boldsymbol{u}_i)} \neq F_{\mathsf{sk}}(y_i)$ with all but negligible probability, identical to condition (1) in hybrid $\mathcal{H}_2$. 2) If Π_{MtA} is executed honestly by $\mathcal{A}$, an honest client aborts if $\mathcal{A}$ provides an incorrect $\boldsymbol{m}_i$ or $\boldsymbol{h}_i$, which will result in an incorrect $F_{\mathsf{sk}}(\boldsymbol{y}_i)$ that does not satisfy the verification procedure, identical to condition (2) in hybrid $\mathcal{H}_2$. 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 OVUF in Section B that reduces the number of g^{R}s to two and achieves better RAM usage. The



(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.

key idea is to batch operations with correlated randomness together but refer to Section B for complete description of the protocol and the proof.

6 Performance Evaluation

We implement our protocols using EMP [WMK16] for COT and RELIC [AGM+] for pairings. We benchmark the performance of our protocol when $\mathcal{F}_{\mathsf{COT}}$ is instantiated using KOS [KOS15] and Ferret [YWL+20].

6.1 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.

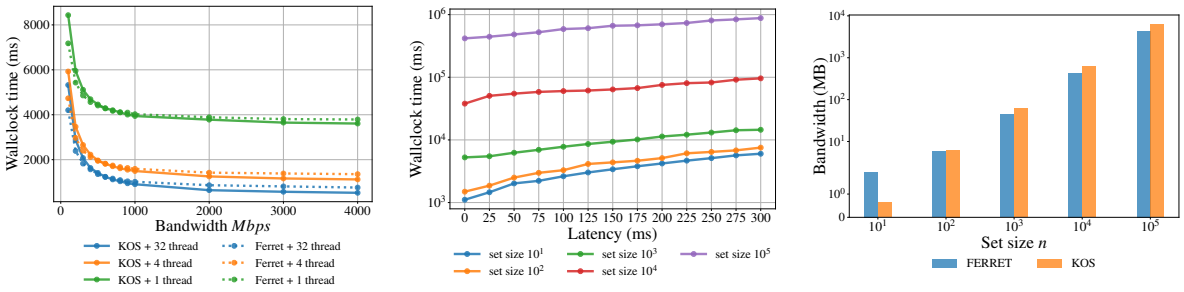
6.2 Efficiency of Server's Encoding

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.

6.3 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.



(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.

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 $F_{\mathsf{sk}}(x_i)$; 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 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 dependence 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.

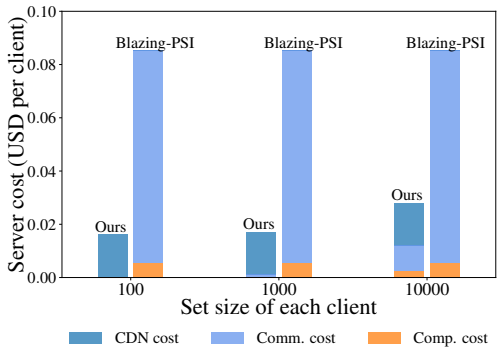


Figure 10: **Server cost comparison with Blazing-PSI [RR22].** All experiments are run on AWS instance. Costs are estimated based on AWS instance pricing and network pricing.

Set size		Security	Protocol	Offline		Online	
$\|X\|$	$\|Y\|$			time (s)	comm. (MB)	time (s)	comm. (MB)
2^{28}	2^{10}	Semi-honest	[KLS+17] (w/ LowMC)	164.82	2144	1.37	23.6
			[KLS+17] (w/ NR)	44681	2144	0.63	6.07
			[CHLR18]	4628	0	12.1	18.4
			[CMdG+21]	4371	0	23.35	12.86
			[RA18]	3684.1	2415	0.16	0.07
		Malicious	Ours (w/ KOS)	**1556.7**	**2147**	**0.44**	**63.23**
2^{28}	2^{7}	Semi-honest	[KLS+17] (w/ LowMC)	164.82	2144	0.41	2.96
			[KLS+17] (w/ NR)	44681	2144	0.13	0.77
			[CHLR18]	4628	0	12.1	18.4
			[CMdG+21]	4350	0	25.65	12.81
			[RA18]	3684.1	2415	0.02	0.008
		Malicious	Ours (w/ KOS)	**1556.7**	**2147**	**0.11**	**7.92**
2^{20}	2^{10}	Semi-honest	[KLS+17] (w/ LowMC)	0.51	8.37	1.37	23.6
			[KLS+17] (w/ NR)	173.41	8.37	0.63	6.07
			[CHLR18]	1.1	0	0.5	5
			[CMdG+21]	13.4	0	1.1	2.04
			[RA18]	18.17	9.43	0.06	0.031
		Malicious	Ours (w/ KOS)	**5.85**	**8.38**	**0.44**	**63.23**
2^{20}	2^{7}	Semi-honest	[KLS+17] (w/ LowMC)	0.51	8.37	0.41	2.96
			[KLS+17] (w/ NR)	173.41	8.37	0.13	0.77
			[CHLR18]	1.2	0	0.2	3.9
			[CMdG+21]	13.5	0	1.1	1.99
			[RA18]	18.17	9.43	0.027	0.004
		Malicious	Ours (w/ KOS)	**5.85**	**8.38**	**0.11**	**7.92**

Table 1: Performance of unbalanced PSI with server set X and client set Y. Our protocol and [CMdG+21] were tested with 32 threads, with [CMdG+21] using 256 GB RAM for a 2^{28} server set. [CHLR18] is obtained based on numbers from their paper, which is based on faster hardware than our testbed. [KLS+17] is tested with 32 threads for the offline phase and 1 thread for the online phase. [RA18] is tested with 1 thread, except the case of $|X| = 2^{28}$ where the online performance are extrapolated.

6.4 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 [RR22], 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 [RR22], which is so far the fastest and improved upon VOLE-based PSI [RS21]. 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 ($0.1728 USD/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 ($0.05 USD/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 ($0.02 USD/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 our protocol performs scalably compared to related protocols across server set sizes $\{2^{20}, 2^{28}\}$ and client set sizes $\{2^7, 2^{10}\}$:

- OPRF-based solutions by [KLS+17] 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 [CHLR18, CMdG+21] 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 when the server set is at a large scale of 2^{28}, albeit with higher communication costs. 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 [RA18]. The solution is semi-honest, but the original proposal by Jarecki and Liu [JL10] 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.

7 Discussion

In this work, we designed an efficient OVUF protocol and used it to construct a malicious protocol that allows a server to compute PSI with multiple clients while ensuring consistency. Interesting future works include extending the model to general-purpose MPC and also reducing the size of server encodings.

Acknowledgements

Work of Xiao Wang is supported by NSF award #2236819 and Google Research Awards.

References

[ADDS21] Martin R. Albrecht, Alex Davidson, Amit Deo, and Nigel P. Smart. Round-optimal verifiable oblivious pseudorandom functions from ideal lattices. In *PKC 2021, Part II*, volume 12711 of *LNCS*, May 10–13, 2021.

[AGM+] 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.

[Bas24] Andrea Basso. A post-quantum round-optimal oblivious PRF from isogenies. In *SAC 2023*, LNCS, August 2024.

[BB04] Dan Boneh and Xavier Boyen. Short signatures without random oracles. In *EUROCRYPT 2004*, volume 3027 of *LNCS*, May 2–6, 2004.

[BBD+11] Pierre Baldi, Roberta Baronio, Emiliano De Cristofaro, Paolo Gasti, and Gene Tsudik. Countering GATTACA: efficient and secure testing of fully-sequenced human genomes. In *ACM CCS 2011*, October 17–21, 2011.

[BDKP22] Shany Ben-David, Yael Tauman Kalai, and Omer Paneth. Verifiable private information retrieval. In *TCC 2022, Part III*, volume 13749 of *LNCS*, November 7–10, 2022.

[BKW20] Dan Boneh, Dmitry Kogan, and Katharine Woo. Oblivious pseudorandom functions from isogenies. In *ASIACRYPT 2020, Part II*, volume 12492 of *LNCS*, December 7–11, 2020.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*. IEEE Computer Society Press, October 14–17, 2001.

[CCL+19] Guilhem Castagnos, Dario Catalano, Fabien Laguillaumie, Federico Savasta, and Ida Tucker. Two-party ECDSA from hash proof systems and efficient instantiations. In *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, August 18–22, 2019.

[CDG+18] Jan Camenisch, Manu Drijvers, Tommaso Gagliardoni, Anja Lehmann, and Gregory Neven. The wonderful world of global random oracles. In *EUROCRYPT 2018, Part I*, volume 10820 of *LNCS*, April 29 – May 3, 2018.

[CGG+20] Ran Canetti, Rosario Gennaro, Steven Goldfeder, Nikolaos Makriyannis, and Udi Peled. UC non-interactive, proactive, threshold ECDSA with identifiable aborts. In *ACM CCS 2020*, November 9–13, 2020.

[CHK+06] Jan Camenisch, Susan Hohenberger, Markulf Kohlweiss, Anna Lysyanskaya, and Mira Meyerovich. How to win the clonewars: Efficient periodic n-times anonymous authentication. In *ACM CCS 2006*, October 30 – November 3, 2006.

[CHL22] Sílvia Casacuberta, Julia Hesse, and Anja Lehmann. Sok: oblivious pseudorandom functions. In *IEEE EuroS&P*, 2022.

[CHLR18] Hao Chen, Zhicong Huang, Kim Laine, and Peter Rindal. Labeled PSI from fully homomorphic encryption with malicious security. In *ACM CCS 2018*, October 15–19, 2018.

[CLR17] Hao Chen, Kim Laine, and Peter Rindal. Fast private set intersection from homomorphic encryption. In *ACM CCS 2017*, October 31 – November 2, 2017.

[CMdG+21] 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 *ACM CCS 2021*, November 15–19, 2021.

[dCL24] Leo de Castro and Keewoo Lee. Verisimplepir: Verifiability in simplepir at no online cost for honest servers. In *USENIX Security 2024*, August 10–12, 2024.

[DGS+18] Alex Davidson, Ian Goldberg, Nick Sullivan, George Tankersley, and Filippo Valsorda. Privacy pass: Bypassing internet challenges anonymously. *PoPETs*, 2018(3), July 2018.

[DKLs18] 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*. IEEE Computer Society Press, May 21–23, 2018.

[DKLs19] 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*. IEEE Computer Society Press, May 19–23, 2019.

[DKT10] Emiliano De Cristofaro, Jihye Kim, and Gene Tsudik. Linear-complexity private set intersection protocols secure in malicious model. In *ASIACRYPT 2010*, volume 6477 of *LNCS*, December 5–9, 2010.

[DY05] Yevgeniy Dodis and Aleksandr Yampolskiy. A verifiable random function with short proofs and keys. In *PKC 2005*, volume 3386 of *LNCS*, January 23–26, 2005.

[ECS+15] Adam Everspaugh, Rahul Chatterjee, Samuel Scott, Ari Juels, and Thomas Ristenpart. The pythia PRF service. In *USENIX Security 2015*, August 12–14, 2015.

[GG18] Rosario Gennaro and Steven Goldfeder. Fast multiparty threshold ECDSA with fast trustless setup. In *ACM CCS 2018*, October 15–19, 2018.

[Gil99] Niv Gilboa. Two party RSA key generation. In *CRYPTO'99*, volume 1666 of *LNCS*, August 15–19, 1999.

[HHC+23] 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. In *USENIX Security 2023*, August 10–12, 2023.

[HMRT22] Iftach Haitner, Nikolaos Makriyannis, Samuel Ranellucci, and Eliad Tsfadia. Highly efficient OT-based multiplication protocols. In *EUROCRYPT 2022, Part I*, volume 13275 of *LNCS*, May 30 – June 3, 2022.

[IKN+20] Mihaela Ion, Ben Kreuter, Ahmet Erhan Nergiz, Sarvar Patel, Shobhit Saxena, Karn Seth, Mariana Raykova, David Shanahan, and Moti Yung. On deploying secure computing: Private intersection-sum-with-cardinality. In *EuroS&P*. IEEE, 2020.

[JKR19] Stanislaw Jarecki, Hugo Krawczyk, and Jason K. Resch. Updatable oblivious key management for storage systems. In *ACM CCS 2019*, November 11–15, 2019.

[JL10] Stanislaw Jarecki and Xiaomin Liu. Fast secure computation of set intersection. In *SCN 10*, volume 6280 of *LNCS*, September 13–15, 2010.

[KLOR20] Ben Kreuter, Tancrède Lepoint, Michele Orrù, and Mariana Raykova. Anonymous tokens with private metadata bit. In *CRYPTO 2020, Part I*, volume 12170 of *LNCS*, August 17–21, 2020.

[KLS+17] Ágnes Kiss, Jian Liu, Thomas Schneider, N. Asokan, and Benny Pinkas. Private set intersection for unequal set sizes with mobile applications. *PoPETs*, 2017(4), October 2017.

[KOS15] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively secure OT extension with optimal overhead. In *CRYPTO 2015, Part I*, volume 9215 of *LNCS*, August 16–20, 2015.

[KRS+19] Daniel Kales, Christian Rechberger, Thomas Schneider, Matthias Senker, and Christian Weinert. Mobile private contact discovery at scale. In *USENIX Security 2019*, August 14–16, 2019.

[LN18] Yehuda Lindell and Ariel Nof. Fast secure multiparty ECDSA with practical distributed key generation and applications to cryptocurrency custody. In *ACM CCS 2018*, October 15–19, 2018.

[LPA+19] Lucy Li, Bijeeta Pal, Junade Ali, Nick Sullivan, Rahul Chatterjee, and Thomas Ristenpart. Protocols for checking compromised credentials. In *ACM CCS 2019*, November 11–15, 2019.

[MRV99] Silvio Micali, Michael O. Rabin, and Salil P. Vadhan. Verifiable random functions. In *40th FOCS*. IEEE Computer Society Press, October 17–19, 1999.

[NMH+10] Shishir Nagaraja, Prateek Mittal, Chi-Yao Hong, Matthew Caesar, and Nikita Borisov. BotGrep: Finding P2P bots with structured graph analysis. In *USENIX Security 2010*, August 11–13, 2010.

[NTY21] Ofri Nevo, Ni Trieu, and Avishay Yanai. Simple, fast malicious multiparty private set intersection. In *ACM CCS 2021*, November 15–19, 2021.

[PIB+22] 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 *USENIX Security 2022*, August 10–12, 2022.

[PRTY20] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. PSI from PaXoS: Fast, malicious private set intersection. In *EUROCRYPT 2020, Part II*, volume 12106 of *LNCS*, May 10–14, 2020.

[RA18] Amanda C. Davi Resende and Diego F. Aranha. Faster unbalanced private set intersection. In *FC 2018*, volume 10957 of *LNCS*, February 26 – March 2, 2018.

[RR22] Srinivasan Raghuraman and Peter Rindal. Blazing fast PSI from improved OKVS and subfield VOLE. In *ACM CCS 2022*, November 7–11, 2022.

[RS21] Peter Rindal and Phillipp Schoppmann. VOLE-PSI: Fast OPRF and circuit-PSI from vector-OLE. In *EUROCRYPT 2021, Part II*, volume 12697 of *LNCS*, October 17–21, 2021.

[RT21] Mike Rosulek and Ni Trieu. Compact and malicious private set intersection for small sets. In *ACM CCS 2021*, November 15–19, 2021.

[SHB23] István András Seres, Máté Horváth, and Péter Burcsi. The legendre pseudorandom function as a multivariate quadratic cryptosystem: Security and applications. *Applicable Algebra in Engineering, Communication and Computing*, 2023.

[SS22] Tjerand Silde and Martin Strand. Anonymous tokens with public metadata and applications to private contact tracing. In *FC 2022*, volume 13411 of *LNCS*, May 2–6, 2022.

[TCR+22] Nirvan Tyagi, Sofía Celi, Thomas Ristenpart, Nick Sullivan, Stefano Tessaro, and Christopher A. Wood. A fast and simple partially oblivious PRF, with applications. In *EUROCRYPT 2022, Part II*, volume 13276 of *LNCS*, May 30 – June 3, 2022.

[WMK16] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit, 2016.

[XAX+21] Haiyang Xue, Man Ho Au, Xiang Xie, Tsz Hon Yuen, and Handong Cui. Efficient online-friendly two-party ECDSA signature. In *ACM CCS 2021*, November 15–19, 2021.

[YWL+20] Kang Yang, Chenkai Weng, Xiao Lan, Jiang Zhang, and Xiao Wang. Ferret: Fast extension for correlated OT with small communication. In *ACM CCS 2020*, November 9–13, 2020.

A Deferred Proofs

A.1 Proof of Lemma 1

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 $h_{\mathsf{g}^{\mathsf{R}}}(\gamma) := \langle \mathsf{g}^{\mathsf{R}}, \gamma \rangle$ is a 2-universal hash function [DKLs18] 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

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

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

A.2 Proof of Lemma 2

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 $h_{\mathsf{g}^{\mathsf{R}}}(\boldsymbol{\gamma}) := \langle \mathsf{g}^{\mathsf{R}}, \gamma^1 || \gamma^{n+1} \rangle || \cdots || \langle \mathsf{g}^{\mathsf{R}}, \gamma^n || \gamma^{n+1} \rangle$ is a 2-universal hash function [DKLs18] 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

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

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

B 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 B.1. The details and proof of Π_{OVUF2} are depicted in Appendix B.2.

B.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 eliminates 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.

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.

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 $\boldsymbol{w} = \mathit{Encode}(\mathsf{g}^{\mathsf{R}}, b) \in \mathbb{Z}_q^{t+2s}$.

$$\boldsymbol{f}_i = \sum_{i \in [t]} \mathsf{g}_i \boldsymbol{e}_i \boldsymbol{w}_i + \sum_{i \in [2s]} \mathsf{g}_{t+i} \boldsymbol{e}_{t+i} \boldsymbol{w}_{t+i} \tag{4}$$

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 B.2 with a detailed proof.

B.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 $(y_1, \ldots, y_n)$ and collaborates with a server to compute OVUF, the Π_{OVUF} processes each input element $y_i, i \in [n]$ one by one. This sequential processing involves n iterations of Π_{MtA}. For each iteration of Π_{MtA}, it runs with input vector $(sk, \boldsymbol{\phi}_1^i) \in \mathbb{Z}_q^2$ and $(\boldsymbol{\phi}_2^i, y_i) \in \mathbb{Z}_q^2$ to compute additive share of $sk \cdot \boldsymbol{\phi}_2^i$ and $\boldsymbol{\phi}_1^i \cdot y_i$. In this section, we maximize the batch feature of MTA protocols and execute $sk \cdot \boldsymbol{\phi}_2^i$ and $\boldsymbol{\phi}_1^i \cdot y_i$ for each $i \in [n]$ as follows:

1. Execute $\Pi_{\mathsf{U-MtA}}$ to efficiently compute additive shares of $sk \cdot \boldsymbol{\phi}_2^i, i \in [n]$.
2. Execute Π_{MtA} to compute additive shares of $\boldsymbol{\phi}_1^i \cdot y_i, i \in [n]$.

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 C_j is regarded as the sender of $\mathcal{F}_{\mathsf{COT}}$ and the one who executes a selective failure attack to

Protocol Π_{OVUF2}

Inputs and parameters: Hash function H modeled as RO. Client C_j holds vector $(y_1, \ldots, y_n) \in \mathbb{Z}_q^n$.
Initialization: S chooses $\mathsf{sk} \in \mathbb{Z}_q$, sets $\mathsf{pk} = g^{\mathsf{sk}}$, and sends pk to $\mathcal{F}_{\mathsf{BB}}$.
Key query: Client C_j sends fetch to $\mathcal{F}_{\mathsf{BB}}$ and receives pk.
Evaluation:

1. C_j checks if $g^{-y_i} = \mathsf{pk}$ for each $i \in [n]$. If it is, C_j inserts i to set I and sets $y_i \leftarrow \mathbb{Z}_q, i \in I$.
2. Server S chooses $\boldsymbol{\phi}_1 \leftarrow \mathbb{Z}_q^n$; client C_j chooses $\boldsymbol{\phi}_2 \leftarrow \mathbb{Z}_q^n$.
3. S and C_j inputs sk and $\boldsymbol{\phi}_2 \in \mathbb{Z}_q^n$ to $\Pi_{\mathsf{U-MtA}}$, receives $A_1 \in \mathbb{Z}_q^n$ and $A_2 \in \mathbb{Z}_q^n$ respectively, such that $A_1 + A_2 = \mathsf{sk} \cdot \boldsymbol{\phi}_2$.
4. S and C_j inputs $\boldsymbol{\phi}_1 \in \mathbb{Z}_q^n$ and $(y_1, \ldots, y_n) \in \mathbb{Z}_q^n$ to Π_{MtA}, receives $B_1 \in \mathbb{Z}_q^n$ and $B_2 \in \mathbb{Z}_q^n$ respectively, such that $B_1^i + B_2^i = \boldsymbol{\phi}_1^i \cdot y_i$ for each $i \in [n]$.
5. S samples $t \in [n]$, computes $V_S = H(g^{A_1^t})$, and sends (t, V_S) to C_j. C_j computes $V_R = H(\mathsf{pk}^{\boldsymbol{\phi}_2^t} / g^{A_2^t})$. C_j checks whether $V_R = V_S$ and aborts if they are not equal.
6. S sends $\boldsymbol{m}$ that $\boldsymbol{m}_i = \mathsf{sk} \cdot \boldsymbol{\phi}_1^i + A_1^i + B_1^i$ to C_j. C_j sends $\boldsymbol{u}$ that $\boldsymbol{u}_i = y_i \cdot \boldsymbol{\phi}_2^i + A_2^i + B_2^i$ to S. Both S and C_j computes $\boldsymbol{v} = \boldsymbol{u} + \boldsymbol{m}$.
7. For each $i \in [n]$, S sends $\boldsymbol{h}_i = g^{\boldsymbol{\phi}_i / \boldsymbol{v}_i}$ to C_j. C_j sets $F_{\mathsf{sk}}(y_i) = 1, i \in I$. For each $i \in [n] \setminus I$, C_j computes $F_{\mathsf{sk}}(y_i) = \boldsymbol{h}_i \cdot g^{\boldsymbol{\phi}_2^i / \boldsymbol{v}_i}$.
8. C_j outputs $(F_{\mathsf{sk}}(y_1), \ldots, F_{\mathsf{sk}}(y_n))$ if $e(g^{y_i} \cdot \mathsf{pk}, F_{\mathsf{sk}}(y_i)) = e(g, g)$ for each $i \in [n] \setminus I$. Otherwise it aborts.

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

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} (with sub-protocols Π_{MtA} and $\Pi_{\mathsf{U-MtA}}$) is secure in the ($\mathcal{F}_{\mathsf{COT}}$, $\mathcal{F}_{\mathsf{BB}}$)-hybrid model.

Theorem 3. *If H is modeled as a random oracle, then protocol Π_{OVUF2} UC-realizes $\mathcal{F}_{\mathsf{OVUF}}$ in the ($\mathcal{F}_{\mathsf{COT}}$, $\mathcal{F}_{\mathsf{BB}}$)-hybrid model with sub-protocol Π_{MtA} and $\Pi_{\mathsf{U-MtA}}$.*

Proof. Let $\mathcal{A}$ be a PPT adversary that allows to corrupt the server or the client. We construct a PPT simulator Sim 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 Sim and the honest party in the ideal world execution.

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

0. Sim emulates $\mathcal{F}_{\mathsf{BB}}$, once it receives fetch from $\mathcal{A}$. Sim sends fetch to $\mathcal{F}_{\mathsf{OVUF}}$ and receives pk. Sim sends pk to $\mathcal{A}$.

2-3. Sim simulates the sub-protocol $\Pi_{\mathsf{U-MtA}}$ and acts as an honest receiver of $\mathcal{F}_{\mathsf{COT}}$ below.

(1)-(2) Sim emulates $\mathcal{F}_{\mathsf{COT}}$. Sim receives $\boldsymbol{\tau} \in \mathbb{Z}_q^{nt+2ns}$ and sends $\boldsymbol{p} \leftarrow \mathbb{Z}_q^{nt+2ns}$ to $\mathcal{A}$. Sim 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, Sim extracts $\boldsymbol{\phi}_2^i = \boldsymbol{\tau}_j$.

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

(4) Sim computes $A_2 \in \mathbb{Z}_q^n$ as an honest P_0 does in step 4 in $\Pi_{\mathsf{U-MtA}}$.

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

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

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

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

(5) Sim computes $B_2 \in \mathbb{Z}_q$ as an honest P_1 does in step 5 in Π_{MtA}.

5. Sim samples $t \in [n]$, $V_S^* \leftarrow \mathbb{G}$ to $\mathcal{A}$. Sim emulates H. Once $\boldsymbol{\phi}_2^t$ is extracted in step 3 and the received query $q = (\mathsf{pk}^{\boldsymbol{\phi}_2^t}/g^{A_2^t})$, Sim sends V_S^* to $\mathcal{A}$. Otherwise, Sim sends a random value to $\mathcal{A}$.

6. Sim sends $\boldsymbol{m}_i^* \leftarrow \mathbb{Z}_q$ to $\mathcal{A}$. Sim receives $\boldsymbol{u}$ from $\mathcal{A}$ and computes $\boldsymbol{v}$.

7. Sim sends $(\mathsf{eval}, (y_1, \ldots, y_n))$ to $\mathcal{F}_{\mathsf{OVUF}}$ and waits to receive $(F_{\mathsf{sk}}(y_1), \ldots, F_{\mathsf{sk}}(y_n))$. For each $i \in [n]$, Sim checks whether $\boldsymbol{\phi}_2^i$ is extracted and $\boldsymbol{u}_i = \boldsymbol{\phi}_2^i \cdot y_i + A_2^i + B_2^i$. If all requirements are satisfied, Sim simulates $\boldsymbol{h}_i^* = \frac{F_{\mathsf{sk}}(y_i)}{g^{\boldsymbol{\phi}_2^i/(\boldsymbol{m}_i^* + \boldsymbol{u}_i)}}$ and sends it to $\mathcal{A}$. Otherwise, Sim simulates $\boldsymbol{h}_i^* \leftarrow \mathbb{G}$ and sends $\boldsymbol{h}^*$ to $\mathcal{A}$.

8. Sim 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 $(\mathcal{F}_{\mathsf{BB}}, \mathcal{F}_{\mathsf{COT}})$-hybrid.

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

For $\mathsf{step\ 0}$, Sim emulates $\mathcal{F}_{\mathsf{BB}}$. Upon receiving fetch from $\mathcal{A}$, Sim sends fetch to $\mathcal{F}_{\mathsf{OVUF}}$ and receives pk. Sim sends pk to $\mathcal{A}$. In hybrid $\mathcal{H}_0$, $\mathcal{F}_{\mathsf{BB}}$ was initialized by an honest server with pk and sends it to $\mathcal{A}$ upon receiving fetch. Thus, the pk sends by Sim is same as the one in hybrid $\mathcal{H}_0$.

For $\mathsf{step\ 2\text{-}3}$, Sim simulates sub-protocol $\Pi_{\mathsf{U-MtA}}$. Sim emulates $\mathcal{F}_{\mathsf{COT}}$, sends $\boldsymbol{p} \leftarrow \mathbb{Z}_q^{nt+2ns}$ to $\mathcal{A}$. Sim 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 $\mathsf{step\ 4}$, Sim simulates sub-protocol Π_{MtA}. Sim 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 Sim are indistinguishable from those in Hybrid $\mathcal{H}_0$.

For $\mathsf{step\ 5}$, Sim samples (t, V_S^*) to $\mathcal{A}$. Sim also programs random oracle H and sends V_S^* to $\mathcal{A}$ if it receives query $q = \mathsf{pk}^{\boldsymbol{\phi}_2^t}/g^{A_2^t}$ with $\boldsymbol{\phi}_2^t$ extracted. Otherwise, Sim samples a random value to $\mathcal{A}$. In hybrid $\mathcal{H}_0$, an honest server uses sk that corresponding to pk in $\Pi_{\mathsf{U-MtA}}$ and computes $V_S = H(g^{A_1^t})$ for a randomly sampled $t \leftarrow [n]$. Since $A_1^t + A_2^t = \mathsf{sk} \cdot \boldsymbol{\phi}_2^t$ holds for an honest server and honest client, the received V_S equals to V_R if V_R is computed from $(\mathsf{pk}^{\boldsymbol{\phi}_2^t}/g^{A_2^t})$ honestly. Thus, the simulated V_R^* equals to V_S^* if $\mathcal{A}$ query random oracle honestly, which is indistinguishable from Hybrid $\mathcal{H}_0$. If $\mathcal{A}$ adds error $\boldsymbol{e} \in \mathbb{Z}_q^{nt+2ns}$ in the execution of $\Pi_{\mathsf{U-MtA}}$, $A_1^t + A_2^t = \mathsf{sk} \cdot \boldsymbol{\phi}_2^t + \boldsymbol{f}_t$ holds for honest server and the adversary. $\boldsymbol{f}_t$ is computed from Equation 4. 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, A_1^t is uniformly distributed over $\mathbb{Z}_q$ to $\mathcal{A}$ and V_S^* is indistinguishable from V_S in hybrid $\mathcal{H}_0$.

For step 6, Sim sends $\boldsymbol{m}_i^* \leftarrow \mathbb{Z}_q$ to $\mathcal{A}$. In hybrid $\mathcal{H}_0$, an honest server computes $\boldsymbol{m}_i = \boldsymbol{\phi}_1^i \cdot \mathsf{sk} + A_1^i + B_1^i$ and sends it to $\mathcal{A}$. If $\mathcal{A}$ acts honestly in previous steps, $\boldsymbol{m}_i$ satisfies the distribution that $\boldsymbol{m}_i + y_i \cdot \boldsymbol{\phi}_2^i + A_2^i + B_2^i = \boldsymbol{v}_i = (\boldsymbol{\phi}_1^i + \boldsymbol{\phi}_2^i)(\mathsf{sk} + y_i)$. Since $\boldsymbol{\phi}_1^i$ is randomly sampled by an honest server, $\mathcal{A}$ has no idea about the distribution of $\boldsymbol{m}_i$. 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 Sim is identical to hybrid $\mathcal{H}_0$. If there exists an error $\boldsymbol{e}$ sampled by $\mathcal{A}$ in sub-protocol $\Pi_{\mathsf{U-MtA}}$, $\boldsymbol{m}_i$ satisfies the distribution that $\boldsymbol{m}_i + y_i \cdot \boldsymbol{\phi}_2^i + A_2^i + B_2^i = \boldsymbol{v}_i = (\boldsymbol{\phi}_1^i + \boldsymbol{\phi}_2^i)(\mathsf{sk} + y_i) + \mathsf{diff_i}$. $\mathsf{diff_i}$ resulted from incorrect $\mathsf{sk} \cdot \boldsymbol{\phi}_2^i$, computed by Equation 4. As analyzed above, $\mathsf{diff_i}$ is 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 $\boldsymbol{m}_i^*$.

Thus, the view simulated by Sim is identical to hybrid $\mathcal{H}_0$.

For step 7, Sim sends $(\mathsf{eval}, (y_1, \cdots, y_n))$ to $\mathcal{F}_{\mathsf{OVUF}}$ and waits for $(F_{\mathsf{sk}}(y_1), \cdots, F_{\mathsf{sk}}(y_n))$. Sim checks whether $\boldsymbol{\phi}_2^i$ is extractable and the received $\boldsymbol{u}_i$ is computed from $y_i \cdot \boldsymbol{\phi}_2^i + A_2^i + B_2^i$ correctly. If it is, Sim sends $\boldsymbol{h}_i^* = \frac{F_{\mathsf{sk}}(y_i)}{g^{\boldsymbol{\phi}_2^i/(\boldsymbol{m}_i^* + \boldsymbol{u}_i)}}$ to $\mathcal{A}$, such that $\boldsymbol{h}_i^* \cdot g^{\boldsymbol{\phi}_2^i/(\boldsymbol{m}_i^* + \boldsymbol{u}_i)} = F_{\mathsf{sk}}(y_i)$. Otherwise, Sim samples $\boldsymbol{h}_i^* \leftarrow \mathbb{G}$ and sends it to $\mathcal{A}$.

In hybrid $\mathcal{H}_0$, an honest server sends $\boldsymbol{h}_i = g^{\boldsymbol{\phi}_1^i/(\boldsymbol{m}_i + \boldsymbol{u}_i)}$ to $\mathcal{A}$. If $\boldsymbol{u}_i$ is computed honestly from $y_i \cdot \boldsymbol{\phi}_2^i + A_2^i + B_2^i$, where $\boldsymbol{\phi}_2^i$ is extracted, then $\boldsymbol{h}_i \cdot g^{\boldsymbol{\phi}_2^i/(\boldsymbol{m}_i + \boldsymbol{u}_i)} = F_{\mathsf{sk}}(y_i)$. Thus, the view of $\boldsymbol{h}_i^*$ in hybrid $\mathcal{H}_0$ is identical to $\boldsymbol{h}_i$ in Hybrid $\mathcal{H}_0$. If $\boldsymbol{u}_i$ is computed honestly, but there exists an error $\boldsymbol{e}$ sampled by $\mathcal{A}$ in sub-protocol $\Pi_{\mathsf{U-MtA}}$ in hybrid $\mathcal{H}_0$, then $\boldsymbol{m}_i + \boldsymbol{u}_i = (\mathsf{sk} + y_i)(\boldsymbol{\phi}_1^i + \boldsymbol{\phi}_2^i) + \mathsf{diff_i}$. Since $\mathsf{diff_i}$ is uniformly distributed over $\mathbb{Z}_q$, $\boldsymbol{h}_i \cdot g^{\boldsymbol{\phi}_2^i/(\boldsymbol{m}_i + \boldsymbol{u}_i)} = F_{\mathsf{sk}}(y_i)$ with negligible probability. In Hybrid $\mathcal{H}_1$, Sim simulates $\boldsymbol{h}_i^* \leftarrow \mathbb{G}$. $\boldsymbol{h}_i^* \cdot g^{\boldsymbol{\phi}_2^i/(\boldsymbol{m}_i^* + \boldsymbol{u}_i)} = F_{\mathsf{sk}}(y_i)$ is negligible and indistinguishable from hybrid $\mathcal{H}_0$. If $\boldsymbol{u}_i$ is not computed honestly, then $\boldsymbol{m}_i + \boldsymbol{u}_i \neq (\boldsymbol{\phi}_1^i + \boldsymbol{\phi}_2^i)(\mathsf{sk} + y_i)$ and thus $\boldsymbol{h}_i \cdot g^{\boldsymbol{\phi}_2^i/(\boldsymbol{m}_i + \boldsymbol{u}_i)} \neq F_{\mathsf{sk}}(y_i)$. In hybrid $\mathcal{H}_1$, Sim simulates $\boldsymbol{h}_i^* \leftarrow \mathbb{G}$, we have $\boldsymbol{h}_i^* \cdot g^{\boldsymbol{\phi}_2^i/(\boldsymbol{m}_i^* + \boldsymbol{u}_i)} = F_{\mathsf{sk}}(y_i)$ w.p. $2^{-\log q}$, which is indistinguishable from hybrid $\mathcal{H}_0$.

Thus, the view simulated by Sim is identical to hybrid $\mathcal{H}_0$.

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

0. Sim emulates $\mathcal{F}_{\mathsf{BB}}$, once it receives the pk from $\mathcal{A}$, Sim stores pk and ignores subsequent messages from $\mathcal{A}$. Sim sends $(\mathsf{init}, \mathsf{pk})$ to $\mathcal{F}_{\mathsf{OVUF}}$.

2-3. Sim simulates the sub-protocol $\Pi_{\mathsf{U-MtA}}$ and acts as an honest sender below.

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

(3) Sim receives $\mathsf{g}^R \in \mathbb{Z}_q^{\log q + 2s}$ from $\mathcal{A}$. Sim recover sk' as: $\mathsf{sk}' = \sum_{i \in [2\log q + 2s]} \mathsf{g_i} \boldsymbol{w}_i$.

(4) Sim computes $A_1 \in \mathbb{Z}_q^n$ as an honest P_1 does in step 4 in $\Pi_{\mathsf{U-MtA}}$.

4. Sim simulates sub-protocol Π_{MtA} and acts as an honest receiver below.

(1)-(3) Sim emulates $\mathcal{F}_{\mathsf{COT}}$ and receives a vector $\boldsymbol{\tau} \in \mathbb{Z}_q^{n(t+2s)}$. Sim samples $\boldsymbol{p} \leftarrow \mathbb{Z}_q^{nt}$, $\boldsymbol{p}' \leftarrow \mathbb{Z}_q^{2ns}$ and sends them to $\mathcal{A}$. Sim 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. Then, it extracts $\boldsymbol{\phi}_1^i = \boldsymbol{\tau}_j$.

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

(5) Sim computes $B_1 \in \mathbb{Z}_q^n$ as an honest P_0 does in step 5 in Π_{MtA}.

5. Sim emulates random oracle H and receives query q from $\mathcal{A}$. Sim samples V_S^* to $\mathcal{A}$ and records (q, V_S^*). Once Sim receives (t, V_S^*) from $\mathcal{A}$, Sim first checks whether $g^{\mathsf{sk}'} = \mathsf{pk}$ and aborts if not.

Then, Sim checks whether the corresponded $q = g^{A_1^t}$. If it is, Sim continue; Otherwise, Sim aborts.

6. Sim receives $\boldsymbol{m}$ from $\mathcal{A}$. Sim sends $\boldsymbol{u}_i^* \leftarrow \mathbb{Z}_q$ to $\mathcal{A}$.

7. Sim waits to receive $\boldsymbol{h}$.

8. For each ith iteration, if $\boldsymbol{\phi}_1^i$ is extracted, Sim checks whether $\boldsymbol{h}_i = g^{\boldsymbol{\phi}_1^i/\boldsymbol{v}_i^*}$, $\boldsymbol{m}_i = \boldsymbol{\phi}_1^i \cdot \mathsf{sk}' + A_1^i + B_1^i$ and sends sk' to $\mathcal{F}_{\mathsf{OVUF}}$. Otherwise, Sim aborts.

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 ($\mathcal{F}_{\mathsf{BB}}$,$\mathcal{F}_{\mathsf{COT}}$)-hybrid model.

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

For Step 2-3, Sim simulates sub-protocol Π_{MtA}. Sim emulates $\mathcal{F}_{\mathsf{COT}}$, sends $\boldsymbol{p} \leftarrow \mathbb{Z}_q^{nt}, \boldsymbol{p}' \leftarrow \mathbb{Z}_q^{2ns}$ to $\mathcal{A}$. Sim 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 4, Sim simulates sub-protocol $\Pi_{\mathsf{U-MtA}}$. Sim 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 6, Sim receives $\boldsymbol{m}$ from $\mathcal{A}$, and sends $\boldsymbol{u}_i^* \leftarrow \mathbb{Z}_q$ to $\mathcal{A}$. In Hybrid $\mathcal{H}_0$, if $\mathcal{A}$ acts honestly in previous steps, we have $\boldsymbol{u}_i + \boldsymbol{\phi}_1^i \cdot \mathsf{sk} + A_1^i + B_1^i = \boldsymbol{v}_i = (\mathsf{sk} + y_i)(\boldsymbol{\phi}_1^i + \boldsymbol{\phi}_2^i)$. Since $\boldsymbol{\phi}_2^i \leftarrow \mathbb{Z}_q$ to $\mathcal{A}$, we have $\boldsymbol{u}_i$ is uniform distributed over $\mathbb{Z}_q$ to $\mathcal{A}$. If $\mathcal{A}$ adds error $\boldsymbol{e}$ in step 4, an honest client computes $\boldsymbol{u}_i$ such that $\boldsymbol{u}_i + \boldsymbol{\phi}_1^i \cdot \mathsf{sk} + A_1^i - B_1^i = \boldsymbol{v}_i = (\mathsf{sk} + y_i)(\boldsymbol{\phi}_1^i + \boldsymbol{\phi}_2^i) + \mathsf{diff}_{\mathsf{i}}$. Since $\mathsf{diff}_{\mathsf{i}}$ is uniform distributed over $\mathbb{Z}_q$, $\boldsymbol{u}_i$ is uniform distributed over $\mathbb{Z}_q$ to $\mathcal{A}$ as well. If $\mathcal{A}$ uses $\mathsf{sk}' \neq \mathsf{sk}$ in step 2-3, $\boldsymbol{u}_i + \boldsymbol{\phi}_1^i \cdot \mathsf{sk}' + A_1^i + B_1^i = \boldsymbol{v}_i = (\mathsf{sk}' + y_i)(\boldsymbol{\phi}_1^i + \boldsymbol{\phi}_2^i)$. $\boldsymbol{u}_i$ is still uniformly distributed as $\boldsymbol{\phi}_2^i$ is distributed uniformly to $\mathcal{A}$. 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 Sim aborts at step 5 in the following conditions: 1) the q corresponding to the received V_S^* not equal to $g^{A_1^t}$; 2) $g^{\mathsf{sk}'} \neq \mathsf{pk}$. Sim also aborts at Step 8 in the following conditions: 1) $g^{\mathsf{sk}'} \neq \mathsf{pk}$; 2) $\boldsymbol{\phi}_1^i$s are not extractable; 3) $\boldsymbol{m}_i \neq \boldsymbol{\phi}_1^i \cdot \mathsf{sk}' + A_1^i + B_1^i$ or $\boldsymbol{h}_i^* \neq g^{\boldsymbol{\phi}_1^i/(\boldsymbol{m}_i + \boldsymbol{u}_i^*)}$.

For step 5 in hybrid $\mathcal{H}_1$, an honest client aborts if the received $V_S \neq V_R$. The client computes $V_R = H(\mathsf{pk}^{\boldsymbol{\phi}_2^t}/g^{A_2^t})$. For each $\mathsf{pk}^{\boldsymbol{\phi}_2^t}/g^{A_2^t}$, it equals to $g^{\mathsf{sk}\cdot\boldsymbol{\phi}_2^t - A_2^t} = g^{A_1^t}$, where sk is corresponded to pk. When adversary use correct sk but manipulate V_S by using inconsistent query $q \neq g^{A_1^t}$, an honest client aborts in hybrid $\mathcal{H}_1$, which is indistinguishable from condition (1) in hybrid $\mathcal{H}_2$. Adversary $\mathcal{A}$ might use invalid sk' that $g^{\mathsf{sk}'} \neq \mathsf{pk}$ in Π_{MtA}, then both parties holds equation $A_1^t + A_2^t = \mathsf{sk}' \cdot \boldsymbol{\phi}_2^t$. Thus, with V_S computed from A_1^t, $V_S = g^{A_1^t} = g^{\mathsf{sk}'\cdot\boldsymbol{\phi}_2^t - A_2^t} \neq V_R = \mathsf{pk} \cdot g^{\boldsymbol{\phi}_2^t - A_2^t}$. Moreover, because of the uniformity of $\boldsymbol{\phi}_2^t$ and A_2^t, $\mathcal{A}$ is not able to construct $A_1^{t'}$ that $g^{A_1^{t'}} = \mathsf{pk} \cdot g^{\boldsymbol{\phi}_2^t - A_2^t}$ either. Thus, the client aborts with all but negligible probability, which is indistinguishable from condition (2) in hybrid $\mathcal{H}_2$.

For step 8 in hybrid $\mathcal{H}_1$, an honest client aborts when $F_{\mathsf{sk}}(y_i)$ does not satisfy $e(g^{y_i} \cdot \mathsf{pk}, F_{\mathsf{sk}}(y_i)) = e(g, g)$, where $F_{\mathsf{sk}}(y_i) = \boldsymbol{h}_i \cdot g^{\boldsymbol{\phi}_2^i/(\boldsymbol{m}_i + \boldsymbol{u}_i)}$. If $g^{\mathsf{sk}'} \neq \mathsf{pk}$, the probability that $\boldsymbol{m}_i + \boldsymbol{u}_i = (\mathsf{sk} + y_i)(\boldsymbol{\phi}_1^i + \boldsymbol{\phi}_2^i)$ with negligible probability. Thus, the value of $\boldsymbol{h}_i \cdot g^{\boldsymbol{\phi}_2^i/(\boldsymbol{m}_i + \boldsymbol{u}_i)}$ satisfy the verification equation with all but negligible probability. The honest client aborts with all but negligible probability which is indistinguishable from condition (1) in hybrid $\mathcal{H}_2$. For the tth iteration, if $\mathcal{A}$ adds error $\boldsymbol{e}$ to $\boldsymbol{\tau}$ and $\boldsymbol{\phi}_2^t$ is not extractable, $\mathsf{diff}_{\mathsf{i}} \neq 0$ with all but negligible probability. Thus, $\boldsymbol{m}_i + \boldsymbol{u}_i \neq (\mathsf{sk} + y_i)(\boldsymbol{\phi}_i + \boldsymbol{\phi}_2^i)$

with all but negligible probability. The value of $\boldsymbol{h}_i \cdot g^{\boldsymbol{\phi}_2^i/(\boldsymbol{m}_i+\boldsymbol{u}_i)}$ satisfy the verification equation with all but negligible probability. The honest client aborts with all but negligible probability which is indistinguishable from condition (2) in hybrid $\mathcal{H}_2$. If $\mathcal{A}$ sends either wrong $\boldsymbol{m}_i$ or wrong $\boldsymbol{h}_i$ in protocol, it will result in wrong $F_{\mathsf{sk}}(y_i)$ that does not satisfy $e(g^{y_i} \cdot \mathsf{pk}, F_{\mathsf{sk}}(y_i)) = e(g, g)$. The honest client aborts with all but negligible probability which is indistinguishable from condition (3) in hybrid $\mathcal{H}_2$.

Therefore, this hybrid is identically distributed as the previous one. This completes the proof. □