









# Efficient Maliciously Secure Oblivious Exponentiations

Carsten Baum<sup>1</sup> , Jens Berlip<sup>9</sup>, Walther Chen<sup>9</sup>, Ivan B. Damgård<sup>2</sup> ,  
Kevin M. Esvelt<sup>3</sup> , Leonard Foner<sup>9</sup>, Dana Gretton<sup>3</sup> , Martin Kysel<sup>9</sup>,  
Ronald L. Rivest<sup>4</sup>, Lawrence Roy<sup>2</sup>, Francesca Sage-Ling<sup>9</sup> , Adi Shamir<sup>5</sup>,  
Vinod Vaikuntanathan<sup>4</sup>, Lynn Van Hauwe<sup>9</sup>, Theia Vogel<sup>9</sup>,  
Benjamin Weinstein-Raun<sup>9</sup>, Daniel Wichs<sup>6,10</sup>, Stephen Wooster<sup>9</sup>,  
Andrew C. Yao<sup>7</sup>  and Yu Yu<sup>8</sup>

<sup>1</sup> Technical University of Denmark, DTU Compute, Kgs. Lyngby, Denmark

<sup>2</sup> Aarhus University, Department of Computer Science, Aarhus, Denmark

<sup>3</sup> MIT, Media Lab, Cambridge, USA

<sup>4</sup> MIT, Computer Science & AI Lab, Cambridge, USA

<sup>5</sup> Weizmann Institute, Department of Computer Science, Rehovot, Israel

<sup>6</sup> Northeastern University, Khoury College of Computer Sciences, Boston, USA

<sup>7</sup> Tsinghua University, Institute for Interdisciplinary Information Sciences, Beijing, China

<sup>8</sup> Shanghai Jiao Tong University, Department of Computer Science and Engineering, Shanghai, China

<sup>9</sup> SecureDNA Foundation, Zug, Switzerland

<sup>10</sup> NTT Research, Cryptography & Information Security, Sunnyvale, USA

**Abstract.** Oblivious Pseudorandom Functions (OPRFs) allow a client to evaluate a pseudorandom function (PRF) on her secret input based on a key that is held by a server. In the process, the client only learns the PRF output but not the key, while the server neither learns the input nor the output of the client. The arguably most popular OPRF is due to Naor, Pinkas and Reingold (Eurocrypt 2009). It is based on an Oblivious Exponentiation by the server, with passive security under the Decisional Diffie-Hellman assumption. In this work, we strengthen the security guarantees of the NPR OPRF by protecting it against active attacks of the server. We have implemented our solution and report on the performance.

Our main result is a new batch OPRF protocol which is secure against maliciously corrupted servers, but is essentially as efficient as the semi-honest solution. More precisely, the computation (and communication) overhead is a multiplicative factor  $o(1)$  as the batch size increases. The obvious solution using zero-knowledge proofs would have a constant factor overhead at best, which can be too expensive for certain deployments.

Our protocol relies on a novel version of the DDH problem, which we call the Oblivious Exponentiation Problem (OEP), and we give evidence for its hardness in the Generic Group model. We also present a variant of our maliciously secure protocol that does not rely on the OEP but nevertheless only has overhead  $o(1)$  over the known semi-honest protocol. Moreover, we show that our techniques can also be used to efficiently protect threshold blind BLS signing and threshold ElGamal decryption against malicious attackers.

---

E-mail: [cabau@dtu.dk](mailto:cabau@dtu.dk) (Carsten Baum), [ivan@cs.au.dk](mailto:ivan@cs.au.dk) (Ivan B. Damgård), [ldr709@gmail.com](mailto:ldr709@gmail.com) (Lawrence Roy)



## 1 Introduction

Oblivious PRFs (OPRFs) are a well-known type of cryptographic protocols between a client and a server. In an OPRF, the client holds a secret input  $q$  while the server holds a PRF key  $K$  for a PRF  $f_K(\cdot)$ . At the end of the protocol, the client will have learned  $y = f_K(q)$  and no other information - in particular nothing about  $K$ . At the same time, the server learns no information about  $q$  or  $y$ . To protect the key  $K$  against attackers that could hack into a server, the role of the server in the OPRF protocol is sometimes distributed by secret-sharing  $K$  among multiple parties, requiring the client to interact with multiple servers to evaluate the PRF. This leads to a so-called Distributed OPRF (or DOPRF) protocol. (D)OPRFs have many applications in cryptography as parts of larger protocols, such as in Private-Set Intersection [HL08], (Distributed) Password Authentication [CLN15, ECS<sup>+</sup>15], Password-Authenticated Key Exchange [JKK14, JKKX17], single-sign on [AMMM18, BFH<sup>+</sup>20] or the well-known Privacy Pass [DGS<sup>+</sup>18] protocol to just name a few. We refer to [CHL22] for an excellent overview on (D)OPRFs.

The arguably most popular construction for a (D)OPRF is due to Naor, Pinkas and Reingold [NPR99] (called *Hashed-DH* in the following) and relies on the Decision Diffie-Hellman problem for its security. The Hashed-DH OPRF is defined as follows: the client  $\mathcal{C}$  has an input  $q \in \{0, 1\}^*$ , while the key  $K$  is chosen as  $K \in \mathbb{Z}_p$  for some prime  $p$ . Each server  $\mathcal{K}_i$  (in the following also denoted as *key servers*) holds a share  $k^{(i)}$  of  $K$ , computed using Shamir’s secret sharing scheme. We assume that there exists a group  $\mathbb{G}$  of order  $p$  where the Decisional Diffie Hellman problem is hard, as well as a fixed publicly known  $g \in \mathbb{G}$ . Furthermore, let  $H$  be a hash function (modeled as a random oracle) mapping from  $\{0, 1\}^*$  to  $\mathbb{G}$ .

The DOPRF value of  $q$  is defined as  $f_K(q) = H(q)^K$ . To compute it obliviously,  $\mathcal{C}$  initially chooses a uniformly random  $r \in \mathbb{Z}_p$ , computes  $X = H(q)$  and sends  $L = X^r$  to each  $\mathcal{K}_i$ , where the blinding factor  $r$  is meant to hide  $X$  from the key servers. Each  $\mathcal{K}_i$  now computes  $Y_i = L^{k^{(i)}}$  and sends  $Y_i$  to  $\mathcal{C}$ . Finally,  $\mathcal{C}$  uses Lagrange interpolation in the exponent to obtain  $L^K$  and then computes  $y = (L^K)^{r^{-1}}$ . It is well-known from [NPR99] that this construction has passive security against a corrupt client or server. At the same time, the Hashed-DH protocol clearly is insecure if any key server deviates from the protocol: a single deviating server can, by changing its share from  $Y_i = L^{k^{(i)}}$  to  $Y_i = L^{k^{(i)} + \epsilon}$  (or even just responding with a random group element), nullify the guarantee that the responses allow to reconstruct a PRF output<sup>1</sup>. This is problematic if e.g. output consistency for multiple clients is necessary. While Zero Knowledge-based solutions exist to tackle this problem, they impose a substantial computational overhead on (at least) the servers to demonstrate that they sent the correct response.

Such Zero Knowledge-based solutions appear to be too strong if the protocol can assume that the client is semi-honest. This makes sense in many cases, for instance when a user stores his secret key in shared form on a number of servers (in which case the client would always be the user’s own device, say his phone) or for applications of DOPRFs such as the SecureDNA protocol<sup>2</sup>.

More generally, the problem of checking that a server responded with a correct exponentiation  $Y_i = L^{k^{(i)}}$  is not a problem unique to the Hashed-DH DOPRF. When considering e.g. threshold BLS signature schemes, or decryption of El Gamal ciphertexts, the same problem of determining the correctness of exponentiation occurs when considering

<sup>1</sup>While computing the correct output for one incorrect response is still possible by guessing the cheater and excluding it, this approach performs poorly if the number of corrupt parties increases (as there are many subsets that have to be enumerated). In fact, even for the benign setting where  $n \geq 3t + 1$  errors cannot simply be decoded as the Shamir shares (or rather, Reed Solomon code symbols) are in the exponent, so Berlekamp-Welch or other decoding methods cannot be applied. [Pei06] showed that, in general, decoding RS codewords with errors in the exponent is a computationally hard problem.

<sup>2</sup>See <https://www.securedna.org> as well as [GWE<sup>+</sup>24, BBC<sup>+</sup>24] for more details.

potentially malicious servers.

## 1.1 Our contributions

In this work, we strengthen the security of the Hashed-DH DOPRF, in order to tolerate potentially malicious servers.

As our main contribution, we describe a new protocol for the interaction between the client and servers that can protect against malicious servers with essentially no overhead compared to a semi-honest secure solution. Namely, while it is easy to get a constant factor overhead using zero-knowledge proofs, our protocol achieves overhead  $o(1)$  assuming the client sends many queries in parallel to the servers (which is indeed the case in many applications). Our protocol works in every case where a set of servers hold shares of an exponent  $E$  and a client wants their help to compute securely  $g^E$  where  $g$  is a group element.

To demonstrate that this improvement is meaningful in practice, we have implemented our actively secure Hashed-DH DOPRF and benchmarked it against the passively secure version. The computational overhead of our solution is  $< 5\%$  in all of our tested scenarios, and often is close to the noise of the measurements. The communication overhead between the client and each key server is  $2 \mathbb{G}$ -elements, independent of the batch size.

Our actively secure DOPRF protocol has been deployed as part of the SecureDNA project [GWE<sup>+</sup>24, BBC<sup>+</sup>24]. SecureDNA uses a DOPRF as part of a privacy-preserving DNA synthesis order screening protocol. The use of our protocol protects the SecureDNA screening system against actively corrupted key servers, which in their setting allows them to mitigate insider threats. By using our protocol, this is done without the need to use additional computational resources as part of the screening.

We also present an unconditionally secure variant of the oblivious exponentiation protocol. The protocol is secure against a minority of maliciously corrupted servers, is slightly less efficient than the first protocol but still has  $o(1)$  factor server-side computation and overall communication overhead over the trivial semi-honest solution.

Finally, we show how our actively secure exponentiation protocols can be included into threshold blind BLS signing and threshold ElGamal decryption protocols to achieve active security against servers.

## 1.2 Technical Overview

Towards allowing verification, we let each  $\mathcal{K}_i$  be committed to  $k^{(i)}$  as  $g^{k^{(i)}}$ . Assume that  $\mathcal{C}$  sends  $m$  messages  $L_1 = X_1^{r_1}, \dots, L_m = X_m^{r_m}$ , to the key servers and wants to learn  $X_1^K, \dots, X_m^K$ . We will construct a protocol  $\mathcal{C}$  can verify that the result it obtains is correct.

Our key observation is that if  $\mathcal{C}$  can be assumed semi-honest, we do not need heavy zero-knowledge protocols if the group elements  $g^K, g^{k^{(1)}}, \dots, g^{k^{(m)}}$  are available to  $\mathcal{C}$ . The idea is that, by rerandomizing the pair  $g, g^K$  (into, say  $G, G^K$ ),  $\mathcal{C}$  can make random instances of inputs to the PRF ( $G$ ) where it knows what the answer should be (namely  $G^K$ ). Rerandomization can be done simply by raising both  $g, g^K$  to a random exponent chosen by  $\mathcal{C}$ . We exploit this as follows: Let  $X_j = H(q_j)$  for  $j \in [m]$ .  $\mathcal{C}$  can set

$$X_0 = G \cdot \left( \prod_{j=1}^m X_j \right)^{-1}$$

and use  $X_0, \dots, X_m$  as inputs to the exponentiation protocol. That is, it chooses  $r_0, \dots, r_m$ , sends  $X_0^{r_0}, \dots, X_m^{r_m}$  to the key servers, and will compute  $Y_0, \dots, Y_m$  from their responses using Lagrange interpolation.

Now, because  $X_0$  was constructed such that  $G = \prod_{j=0}^m X_j$ ,  $\mathcal{C}$  knows that if indeed  $Y_j = X_j^K$ , it should be the case that  $G^K = \prod_{j=0}^m Y_j$ , and this equation can be easily

verified. Interestingly, due to the use of the random blinding factors  $r_j$  that are applied before sending  $X_j$  to the key servers, we can show that checking  $G^K = \prod_{j=0}^m Y_j$  is sufficient to establish correctness of all the results. The protocol is trivially simulatable towards  $\mathcal{C}$ , while we show that key servers cannot cheat assuming that a certain problem which we call *Oblivious Exponentiation Problem* (OEP) is hard in  $\mathbb{G}$ . If the check does not go through,  $\mathcal{C}$  can do further checks (which do not require any extra communication to the servers) to find out which server(s) returned incorrect answers, as we describe in more detail later.

We show that the Oblivious Exponentiation problem is equivalent to a simpler one where you are given  $g, g^a, g^b$  and must output  $h, h^{ab}$ , so a variant of the Diffie-Hellman problem. We give evidence that this problem is hard by reducing it to the Discrete Logarithm problem in the Generic Group Model.

Note that, on the server side, the new DOPRF protocol is the same as the passive version, except that one extra input instance has been added. On the client side, we add  $O(m)$  multiplications in  $\mathbb{G}$ , which are negligible compared to the  $O(m)$  exponentiations we needed already in the passive version. So for the client we add 3 exponentiations, 2 to get  $(G, G^K)$  and 1 for blinding  $X_0$ . Hence, the overhead indeed vanishes as  $m$  increases.

We also show a modification of the aforementioned protocol. This variant is unconditionally sound and reminiscent of [BGR98]. It is the same as our first protocol on the server side and is marginally less efficient for the client. This variant is based on the classic idea of checking a batch of input instances by checking a random linear combination of them. In this case the linear combination happens in the exponent. The soundness of this variant is independent of the blinding that the client uses for obliviousness, so it can be used in general for making distributed exponentiation secure against malicious servers at small amortized cost.

### 1.3 Related Work

A protocol to prove correctness of exponentiation with respect to a committed value  $g^{k^{(i)}}$  was first introduced by Chaum [Cha91]. This approach, or similar  $\Sigma$ -protocols, have constant computation and communication overhead for the server in the batch size  $m$ . Even if this is only a constant factor overhead, it may be problematic in practical applications. While one can compress the proof size using batching techniques such as [GLSY04], this does not reduce the computational overhead as the server still has to compute at least two group exponentiations: one to compute  $Y_{j,i} = X_j^{k^{(i)}}$  from  $X_j$ , and another to raise  $X_j$  or  $Y_{j,i}$  to a random exponent for a linear combination as part of the amortization proof. This can also not be overcome by using folding techniques [BBB<sup>+</sup>18, AC20].

One might alternatively replace the Hashed-DH OPRF with other constructions to achieve malicious security. Two obvious alternative candidates for use of an OPRF are due to Naor & Reingold [NR04] as well as Dodis & Yampolskiy [DY05]. The Naor-Reingold OPRF exhibits better performance on the server-side than Hashed-DH due to possible precomputations. The OT-based solution can be optimized using OT extension [IKNP03]. To the best of our knowledge, none of the OT-based Naor-Reingold OPRF constructions (or batch constructions such as [KKRT16]) can efficiently be made secure against a malicious server or work in the threshold setting without a substantial increase in computation or communication. Although one can distribute the Dodis-Yampolskiy OPRF as shown in e.g. [MPR<sup>+</sup>20] the resulting construction only achieves active security using expensive NIZKs.

## Organization

In Section 2 we will describe necessary preliminaries for this work. We describe our techniques to achieve active security against a dishonest servers in Section 3. Then, we describe the Hashed-DH formally in Section 4, recap the security argument against passive attackers and combine it with our new techniques to achieve security against active

attackers. We provide experiments showing the practical efficiency of our approach to protecting Hashed-DH in 5 and describe other applications of our protocols in more detail in Section 6.

## 2 Preliminaries

In this work, we will denote the client as  $\mathcal{C}$  and the  $n$  key servers as  $\mathcal{K}_1, \dots, \mathcal{K}_n$ . We will assume that at most  $t$  of the  $n$  key servers are corrupted, where  $t < n/2$ . We implicitly assume the existence of a PKI, authenticated channels among all parties as well as the existence of a broadcast channel among the key servers. All corruptions are assumed to be static, and the adversary is allowed to be rushing in the communication model. We use  $\lambda$  to denote the security parameter. We use  $[x..y]$  as a shorthand for the set  $\{x, \dots, y\}$  and write  $[x]$  for  $[1..x]$ .

We assume that a group  $\mathbb{G}$  of prime order  $p$  together with generators  $g, h \in \mathbb{G}$  are provided as a CRS to all parties, where  $\log_g(h)$  is not known. We use multiplicative notation of  $\mathbb{G}$ . We further assume that the DDH problem holds in the group  $\mathbb{G}$ :

**Definition 1** (Decisional Diffie-Hellman (DDH)). Consider the following game between a challenger  $\mathcal{C}$  and an adversary  $\mathcal{A}$ :

1.  $\mathcal{C}$  on input  $1^\lambda, \mathbb{G}, g$  samples  $a, b, r \leftarrow \mathbb{Z}_p$ ,  $\delta \leftarrow \{0, 1\}$  and sets  $c \leftarrow a \cdot b$  if  $\delta = 0$  and  $c \leftarrow r$  if  $\delta = 1$ . It then sends  $(\mathbb{G}, g, g^a, g^b, g^c)$  to  $\mathcal{A}$ .
2.  $\mathcal{A}$ , on input  $(1^\lambda, \mathbb{G}, A, B, C)$  outputs a bit  $\hat{\delta}$ .
3. We say  $\mathcal{A}$  wins iff  $\delta = \hat{\delta}$ .

Then we say an adversary breaks DDH if it wins the aforementioned game with probability  $\gg 1/2 + \text{negl}(\lambda)$ .

By saying that the DDH problem holds in  $\mathbb{G}$ , we mean that no algorithm  $\mathcal{A}$  with runtime polynomial in  $\lambda$  can break DDH.

### 2.1 Shamir Sharing & Lagrange Interpolation

In this work, we use Shamir's secret sharing scheme to keep the DOPRF key  $K$  secret. This means that there implicitly exists a polynomial  $f$  of degree  $t$  such that  $f(0) = K$ , while each key server  $\mathcal{K}_i$  holds a share  $k^{(i)} = f(i)$ . As is known for Shamir's secret sharing scheme, this implies that given any  $t$  key shares, the value  $K$  looks uniformly random.

Assume a set  $A \subset \mathbb{Z}_p$  of size  $k = t + 1$ . Given such a *minimal qualified set*  $A = \{a_1, \dots, a_k\}$  we then define the Lagrange coefficient  $\lambda_{i,j}^A$  which allows to interpolate a polynomial  $f \in \mathbb{Z}_p[X]$  of degree  $t$  at the point  $j \notin A$  given  $\{f(i)\}_{i \in A}$  as

$$\lambda_{i,j}^A := \prod_{\substack{m \in A \\ m \neq i}} \frac{j - m}{i - m}.$$

Whenever  $A = [0..t]$  we write  $c_{i,j} = \lambda_{i,j}^A$  and we write  $c_{i,j}^+$  if  $A = [t + 1]$ . If additionally  $j = 0$  then we write  $c_i = \lambda_{i,0}^A$ . If  $A = [2t + 1]$  and  $j = 0$  then we write  $d_i = \lambda_{i,0}^A$ .

### 2.2 Universal Composability

We use the (Global) Universal Composability or (G)UC model [Can01] for analyzing security and refer interested readers to the original works for more details.

In UC protocols are run by interactive Turing Machines (iTMs) called *parties*. A protocol  $\pi$  will have  $n$  parties denoted as  $P$ . The *adversary*  $\mathcal{A}$ , which is also an iTM,

can corrupt a subset  $I \subset P$  as defined by the security model and gains control over these parties. We say that parties are passively (or semi-honestly) corrupted if they still follow the protocol flow but report their whole state to  $\mathcal{A}$ , while parties are actively (or maliciously) corrupted if they may deviate in any way from the protocol as instructed by  $\mathcal{A}$ .

The parties can exchange messages via resources, called *ideal functionalities* (which themselves are iTMs) and which are denoted by  $\mathcal{F}$ . For simplicity, we assume availability of private authenticated channels for communication between parties but do not specify these further.

As usual, we define security with respect to an iTM  $\mathcal{Z}$  called *environment*. The environment provides inputs to and receives outputs from the parties  $P$  as well as the adversary  $\mathcal{A}$ . To define security, let  $\pi^{\mathcal{F}_1, \dots} \circ \mathcal{A}$  be the distribution of the output of an arbitrary  $\mathcal{Z}$  when interacting with  $\mathcal{A}$  in a real protocol instance  $\pi$  using resources  $\mathcal{F}_1, \dots$ . Furthermore, let  $\mathcal{S}$  denote an *ideal world adversary* and  $\mathcal{F} \circ \mathcal{S}$  be the distribution of the output of  $\mathcal{Z}$  when interacting with parties which run with  $\mathcal{F}$  instead of  $\pi$  and where  $\mathcal{S}$  takes care of adversarial behavior.

**Definition 2.** We say that  $\mathcal{F}$  UC-securely implements  $\pi$  if there exists an iTM  $\mathcal{S}$  (with black-box access to  $\mathcal{A}$ ) for every iTM  $\mathcal{A}$  such that no PPT environment  $\mathcal{Z}$  can distinguish  $\pi^{\mathcal{F}_1, \dots} \circ \mathcal{A}$  from  $\mathcal{F} \circ \mathcal{S}$  with non-negligible probability in  $\lambda$ .

## 2.3 Setup Functionalities

In our construction we assume a Global Programmable and Observable Random Oracle functionality  $\mathcal{G}_{\text{RO}}^R$  as depicted in Fig. 1. In comparison to [CDG<sup>+</sup>18] we parameterize the Random Oracle functionality by a finite set  $R$  that allows to efficiently sample uniformly random elements and efficient membership testing. For simplicity, we define

1.  $\mathcal{G}_{\text{RO}-\mathbb{G}} = \mathcal{G}_{\text{RO}}^{\mathbb{G}}$  as a Random Oracle that outputs  $\mathbb{G}$ -elements; and
2.  $\mathcal{G}_{\text{RO}-\mathbb{Z}_p} = \mathcal{G}_{\text{RO}}^{\mathbb{Z}_p}$  as a Random Oracle that outputs  $\mathbb{Z}_p$ -elements.

For simplicity, we will write  $H(x)$  in this work whenever we mean that a party queries a  $\mathcal{G}_{\text{RO}}^S$ -functionality on input  $x$  and where the output set  $S$  is clear.

**Key Registration.** In Fig. 2 we present the key generation and party registration functionality  $\mathcal{F}_{\text{KeyReg}}$ . It generates key shares and verification keys for every Key Server  $\mathcal{K}_i$ . It can also be used by every party to obtain a verification key.  $\mathcal{F}_{\text{KeyReg}}$  does allow the simulator  $\mathcal{A}$  to obtain the key  $k^{(i)}$  used by an adversarially controlled key server. We allow  $\mathcal{A}$  to corrupt up to  $t$  of the  $n \geq 2t + 1$  Key Servers statically in  $\mathcal{F}_{\text{KeyReg}}$ , as well as any party.

## 3 Oblivious Exponentiation with actively corrupted servers

We now present two approaches to protect exponentiation protocols against actively corrupted servers.

Assume we have a client  $C$  and a server  $S$ . The server holds a secret exponent  $E \in \mathbb{Z}_p$ , and the client has as input  $m$  group elements  $X_1, \dots, X_m \in \mathbb{G}$ , and we assume throughout that none of them are 1. The goal is that  $C$  learns  $X_1^E, \dots, X_m^E$  (and nothing else), while  $S$  learns nothing new.  $S$  may be malicious, while  $C$  is semi-honest.

Of course, if  $S$  were honest, we would use the passively secure protocol from the previous section: for  $j \in [m]$ ,  $C$  chooses  $r_j \in \mathbb{Z}_p$  at random and sends  $X_j^{r_j}$  to  $S$ . Then



**Functionality  $\mathcal{G}_{\text{RO}}^R$** 

$\mathcal{G}_{\text{RO}}^R$  is parameterized by an efficiently samplable finite set  $R$  with efficient membership testing. The functionality keeps initially empty lists  $\text{List}_{\mathcal{H}}, \text{prog}$ .

**Query:** On input (HASH-QUERY,  $m$ ) from party  $(P, \text{sid})$  or  $\mathcal{A}$ , parse  $m$  as  $(s, m')$  and proceed as follows:

1. Look up  $h$  such that  $(m, h) \in \text{List}_{\mathcal{H}}$ . If no such  $h$  exists, sample  $h \xleftarrow{\$} R$  and set  $\text{List}_{\mathcal{H}} = \text{List}_{\mathcal{H}} \cup \{(m, h)\}$ .
2. If this query is made by  $\mathcal{A}$ , or if  $s \neq \text{sid}$ , then add  $(s, m', h)$  to the (initially empty) list of illegitimate queries  $\mathcal{Q}_s$ .
3. Send (HASH-CONFIRM,  $h$ ) to the caller.

**Observe:** On input (OBSERVE,  $\text{sid}$ ) from  $\mathcal{A}$ , if  $\mathcal{Q}_{\text{sid}}$  does not exist yet, set  $\mathcal{Q}_{\text{sid}} = \emptyset$ . Output (LIST-OBSERVE,  $\mathcal{Q}_{\text{sid}}$ ) to  $\mathcal{A}$ .

**Program:** On input (PROGRAM-RO,  $m, h$ ) with  $h \in R$  from  $\mathcal{A}$ , ignore the input if there exists  $h' \in R$  where  $(m, h') \in \text{List}_{\mathcal{H}}$  and  $h \neq h'$ . Otherwise, set  $\text{List}_{\mathcal{H}} = \text{List}_{\mathcal{H}} \cup \{(m, h)\}$ ,  $\text{prog} = \text{prog} \cup \{m\}$  and send (PROGRAM-CONFIRM) to  $\mathcal{A}$ .

**IsProgrammed:** On input (ISPROGRAMMED,  $m$ ) from a party  $P$  or  $\mathcal{A}$ , if the input was given by  $(P, \text{sid})$  then parse  $m$  as  $(s, m')$  and, if  $s \neq \text{sid}$ , ignore this input. Set  $b = 1$  if  $m \in \text{prog}$  and  $b = 0$  otherwise. Then send (ISPROGRAMMED,  $b$ ) to the caller.

**Figure 1:** Restricted observable and programmable global random oracle functionality  $\mathcal{G}_{\text{RO}}^R$  from [CDG<sup>+</sup>18].

$S$  returns  $(X_j^{r_j})^E$ , so  $C$  can compute  $X_j^E = ((X_j^{r_j})^E)^{r_j^{-1}}$ . This requires 2 messages,  $2m$  exponentiations on the client side and  $m$  exponentiations on the server side. We want an actively secure protocol with a minimal overhead compared to this passively secure solution.

### 3.1 A computationally sound protocol

One approach that comes to mind is the well-known idea of checking that a certain operation was applied correctly to a set of objects by taking a random linear combination of them and then only the result of the linear combination is checked. The obvious way to apply this idea in our setting is to do the linear combination in the exponent, so we ask  $C$  to choose random exponents  $d_j$  and compute  $\prod_j X_j^{d_j}$  and  $\prod_j (X_j^E)^{d_j} = (\prod_j X_j^{d_j})^E$ . Then,  $S$  can prove in Zero-Knowledge that this pair of group elements is of the right form. However, this requires at least two extra messages and  $2m$  extra exponentiations on the client side. We shall now see that we can do much better. In a nutshell,  $C$  can arrange it such that she already knows what the result of the linear combination should be, so no zero-knowledge is needed. Furthermore, at the expense of assuming hardness of a specific computational problem, the exponentiations  $C$  needs to do to raise  $X_j$  to  $r_j$ , which is necessary just for passive security, can already be leveraged for the check.

In the following, we will assume that  $C$  is initially given a pair of group elements that is guaranteed to be of form  $(g_0, g_0^E)$ . For instance, this can be generated by  $S$  and proved to be correct in zero-knowledge using a standard protocol or by using  $\mathcal{F}_{\text{KeyReg}}$  in the distributed

**Functionality**  $\mathcal{F}_{\text{KeyReg}}$ 

This functionality communicates with  $\mathcal{K}_1, \dots, \mathcal{K}_n$  as well as parties  $P_1, \dots, P_\ell$  and the ideal adversary  $\mathcal{A}$ .  $\mathcal{F}_{\text{KeyReg}}$  uses a group  $\mathbb{G}$  of prime order  $p$  with fixed generator  $g \in \mathbb{G}$ . The functionality has a secret variable  $K$  that is initially not set.  $\mathcal{A}$  may corrupt up to  $t$  of the  $n \geq 2t + 1$  key servers and any party. We denote the set of corrupted key servers as  $I$ .

**Generate Keys:** Upon first input (`GenerateKey`,  $sid$ ) by an uncorrupted  $\mathcal{K}_i$ :

1. Mark  $\mathcal{K}_i$  as initialized and send (`GenerateKey`,  $sid, i$ ) to  $\mathcal{S}$ .
2. If  $t + 1 - |I|$  honest key servers are initialized then sample a uniformly random  $K \in \mathbb{Z}_p$  and send (`GenerateKey`,  $sid, g^K$ ) to  $\mathcal{A}$ .
3.  $\mathcal{A}$  responds with (`Shares`,  $I, \{k^{(i)}\}_{i \in I}$ ). Then set  $f$  to be a random degree- $t$  polynomial such that  $f(0) = K$  and  $f(i) = k^{(i)}$ . For  $i \in [n] \setminus I$  set  $k^{(i)} = f(i)$  and mark every  $\mathcal{K}_i$  as registered.
4. Send (`GenerateKeyOk`,  $sid, k^{(i)}, g^K, g^{k^{(1)}}, \dots, g^{k^{(n)}}$ ) to every honest  $\mathcal{K}_i$  and  $\mathcal{A}$ .

**Obtain Verification Key:** Upon input (`GetVerKey`,  $sid$ ) by any  $P$ :

1. Send (`ReqKey`,  $sid, P$ ) to  $\mathcal{A}$ .
2. Send (`VerKey`,  $sid, g^K, g^{k^{(1)}}, \dots, g^{k^{(n)}}$ ) to  $P$ .

**Figure 2:** Functionality  $\mathcal{F}_{\text{KeyReg}}$  for Key Generation and Party registration of the DVDOPRF

setting. This is only needed once and for all and so does not affect the efficiency if  $S$  is to serve many requests from  $C$ , which is indeed the case in our application. We observe that  $C$  can sample random pairs of the form  $(g, g^E)$ , by raising  $(g_0, g_0^E)$  to a random exponent. Then, in the protocol, we will ask  $S$  to raise one additional element to the power  $E$  such that the product of all outputs has to be  $g^E$ . The protocol, which we call  $\pi_{\text{MSEP}}$ , is described in Fig. 3.

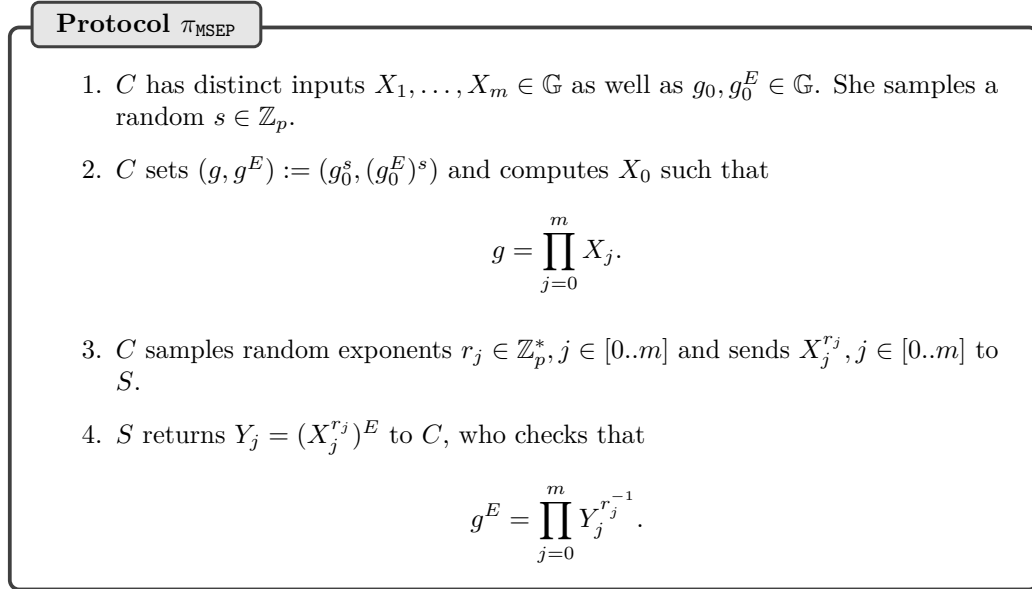
Note that, compared to the passively secure protocol,  $\pi_{\text{MSEP}}$  adds a constant number of exponentiations, no matter how large  $m$  is. Therefore, the amortized overhead for active security can be made arbitrarily small.

We also note that the assumption that all inputs are different is necessary: if  $S$  knows where repetitions are, then there is a simple attack. For simplicity, let  $m = 2$  and  $X_2 = X_1 = X$ . This means that  $S$  gets  $X^{r_1}$  and  $X^{r_2}$  as part of the message in Step 3. Instead of  $Y_1 = (X^{r_1})^E$  and  $Y_2 = (X^{r_2})^E$  it can also return  $Y_1' = (X^{r_1})^E \cdot X^{r_1}$  and  $Y_2' = (X^{r_2})^E \cdot X^{-r_2}$ , both of which can easily be computed. The check done by  $C$  in Step 4 will pass since

$$\begin{aligned} (Y_1')^{r_1^{-1}} \cdot (Y_2')^{r_2^{-1}} &= ((X^{r_1})^E \cdot X^{r_1})^{r_1^{-1}} \cdot ((X^{r_2})^E \cdot X^{-r_2})^{r_2^{-1}} \\ &= X^{2E} = (Y_1)^{r_1^{-1}} \cdot (Y_2)^{r_2^{-1}} \end{aligned}$$

We will now show that  $\pi_{\text{MSEP}}$  is sound if the computational problem (OEP) that we define below is hard in  $\mathbb{G}$ . In our application it may be the case that a corrupt  $S$  has side information on the  $X_j$ 's, he may get this information from another corrupt party. It might even be the case that this other party has some influence on the choice of the  $X_j$ . Namely, although the  $X_j$  are output from a random oracle, the inputs to the oracle may be adversarially chosen. This is equivalent to giving the adversary a polynomial size set of random group elements from which the  $X_j$ 's should be chosen. This is the model used the specification of OEP below.





**Figure 3:** The Maliciously Secure Exponentiation Protocol  $\pi_{\text{MSEP}}$

**Definition 3** (The Oblivious Exponentiation Problem (OEP)). The Oblivious Exponentiation problem is defined as the following game between a challenger  $\mathcal{C}$  and an adversary  $\mathcal{A}$ .

1.  $\mathcal{C}$  samples a uniformly random set  $M \subset \mathbb{G}$  of polynomial size and sends it to  $\mathcal{A}$ .
2.  $\mathcal{A}$  chooses distinct elements  $X_0, \dots, X_m \in M$  and sends this choice to  $\mathcal{C}$ .
3.  $\mathcal{C}$  for  $j \in [0..m]$  chooses random  $r_j \in \mathbb{Z}_p^*$  and sends  $X_j^{r_j}$  to  $\mathcal{A}$ .
4.  $\mathcal{A}$  wins if she outputs  $Z_j \in \mathbb{G}, j \in [0..m]$  such that at least one  $Z_j$  is different from 1, and  $\prod_{j=0}^m Z_j^{r_j^{-1}} = 1$ .

We believe that the OEP problem can reasonably be conjectured to be hard, and give evidence for this below.

Note that since the underlying group  $\mathbb{G}$  is exponentially large, a polynomial size set of random elements in the group (such as  $M$ ) will all be distinct except with negligible probability. For simplicity, we will assume this about  $M$  in following without explicit discussion. Note also that OEP is not hard unless the adversary is required to choose distinct elements as  $X_0, \dots, X_m$ .

In line with the definition of OEP, we define the following game played by a corrupt key server  $S$ :

1.  $\mathcal{C}$  samples a uniformly random set  $M \subset \mathbb{G}$  of polynomial size and sends it to  $\mathcal{A}$ .
2.  $\mathcal{A}$  chooses distinct elements  $X_1, \dots, X_m \in M$  and sends this choice to  $\mathcal{C}$ .
3.  $S$  runs the protocol  $\pi_{\text{MSEP}}$  where the honest client uses  $X_1, \dots, X_m$  as input. We say that corrupt  $S$  is successful if the client accepts, but at least one output  $Y_j$  is incorrect.

**Lemma 1.** *A corrupt  $S$  that is successful in the aforementioned game with probability  $\epsilon$  can be used to solve OEP with probability  $\epsilon$  and essentially the same running time.*

*Proof.* To see this, assume we have a successful corrupt key server, and a set  $M$  as input to the OEP problem. We send  $M$  to  $S$  and let  $X_1, \dots, X_m$  be the set of selected elements, we select a new element in  $M$ , call it  $X_0$ , send  $X_0, \dots, X_m$  to the OEP challenger and receive  $X_j^{r_j}$  for  $j \in [0..m]$ .

We can define  $g = \prod_{j=0}^m X_j$  and note that the joint distribution of  $g$  and the  $X_j$ 's is the same as in the protocol, namely  $M$  contains uniformly random group elements, so choosing a random  $g$  first and computing  $X_0$  to match the equation is equivalent to choosing  $X_0$  first. Thus if we send the  $X_j^{r_j}$  to the server, we get answers back with the same distribution as in the protocol, so they represent a successful cheat with probability  $\epsilon$ .

Define  $Z_j$  by  $Y_j = (X_j^{r_j})^E Z_j$ . In other words,  $Z_j$  is the factor by which the answer from the server is off from what it should be.

A successful cheat means that not all  $Z_j$  are 1 (at least one answer is incorrect), but still the client is happy, that is, we have

$$g^E = \prod_{j=0}^m Y_j^{r_j^{-1}} = \prod_{j=0}^m ((X_j^{r_j})^E Z_j)^{r_j^{-1}} = \prod_{j=0}^m X_j^E Z_j^{r_j^{-1}}$$

From this and  $g = \prod_{j=0}^m X_j$ , it follows immediately that  $\prod_{j=0}^m Z_j^{r_j^{-1}} = 1$  so we have solved the problem.  $\square$

**Lemma 2.** *The view of a semi-honest  $C$  executing the  $\pi_{\text{MSEP}}$  protocol with an honest  $S$  can be perfectly simulated given  $\{X_j^E\}_{i \in [m]}$ .*

*Proof.* To do the simulation, we can simply emulate  $C$ 's side of the protocol as it stands, this is possible because we are given the responses  $Y_j = X_j^E$  of  $S$  for  $j \in [m]$ . Whereas we are not given  $Y_0$ , we know that in the real execution, the equation  $g^E = \prod_{j=0}^m Y_j^{r_j^{-1}}$  always holds, so we just solve this equation for  $Y_0$ .  $\square$

### 3.1.1 Analysis of the OEP

We justify the hardness of OEP by presenting a another problem, Chosen-Base CDH<sup>3</sup>, and showing that OEP can be reduced to it. Chosen-Base CDH is a simpler problem that is easier to understand and analyse.

**Definition 4** (The Chosen-Base CDH Problem). The The Chosen-Base CDH Problem is defined as the following game between a challenger  $\mathcal{C}$  and an adversary  $\mathcal{A}$ .

1.  $\mathcal{C}$  samples a random generator  $g$  of a group  $\mathbb{G}$  of prime order  $p$ , as well as random  $x, y \in \mathbb{Z}_p^*$ .
2.  $\mathcal{C}$  gives  $\mathcal{A}$   $g, g^x$ , and  $g^y$ .
3.  $\mathcal{A}$  chooses a non-identity element  $h \in \mathbb{G}$ , and wins if she outputs  $h^{xy}$ .

We conjecture that the Chosen-Base CDH problem is hard. Informally, the problem is similar enough to CDH that it appears to be hard. More formally, it is hard in the Generic Group Model (GGM), which provides evidence towards its hardness for concrete groups.

**Lemma 3.** *Any adversary in the GGM performing at most  $q$  group operations has probability at most  $\frac{\frac{3}{2}(q+3)^2+3}{p-1}$  of solving Chosen-Base CDH in a prime group of order  $p$ .*

<sup>3</sup>Note that a very similar name was given to a completely different problem defined in [AP05]. That problem was later broken in [Szy06].

*Proof.* We can rephrase the Chosen-Base CDH problem in terms of an adversary  $\mathcal{A}$  obtaining evaluations of degree-1 polynomials in the exponent of  $g$  in  $x, y$ , while having to compute a degree-2 polynomial in  $x, y$  in the exponent of  $g$  from these.

Using this, the proof follows from the interactive GGM master theorem [BFF<sup>+</sup>19, Theorem 5], which shows that the bound above holds unless there exists  $a_0, b_0, c_0, a_1, b_1, c_1 \in \mathbb{Z}_p$  such that

$$\begin{aligned} XY(a_0 + b_0X + c_0Y) &= a_1 + b_1X + c_1Y \\ a_0 + b_0X + c_0Y &\neq 0 \end{aligned}$$

holds over  $\mathbb{Z}_p[X, Y]$ . The left side of the equality (which  $\mathcal{A}$  has to compute) must have total degree 2 or 3, while the right side (which is what  $\mathcal{A}$  obtains from  $\mathcal{C}$ ) can only have total degree 0 or 1, so no such solution can exist.  $\square$

**Lemma 4.** *Any adversary that solves OEP with probability  $\epsilon$  can be used to solve Chosen-Base CDH with probability  $\epsilon/2$ , at the additional cost of  $O(|M|)$  group exponentiations.*

*Proof.* Without loss of generality, we can assume that the adversary chooses the whole set  $M$ , i.e. that  $M = \{X_0, \dots, X_m\}$ , because she can always set  $Z_j = 1$  for elements she doesn't want to use. The reduction starts by receiving  $g, X = g^x$ , and  $Y = g^y$  from the challenger, and for each  $j \leq m$  samples bits  $b_j$  and random numbers  $u_j, v_j \in \mathbb{Z}_p$ . It then sets  $X_j = X^{u_j}$  and  $X_j^{r_j} = g^{v_j}$  if  $b_j = 0$ , and otherwise sets  $X_j = g^{u_j}$  and  $X_j^{r_j} = Y^{v_j}$ , and gives these to the adversary. Essentially, the reduction has set  $r_j = \frac{v_j}{u_j x}$  if  $b_j = 0$ , and  $r_j = \frac{v_j y}{u_j}$  otherwise.<sup>4</sup> If there is a collision for some  $X_k$  with a previous  $X_j$ , the reduction resamples  $u_k$  until the collision is avoided. Note that the distribution of  $(X_j, r_j)_{j \in [0..m]}$  is identical to their distribution with the OEP.

The adversary now outputs  $\{Z_j\}_{j \in [0..m]}$ , where there is some  $Z_k$  such that  $Z_k \neq 1$ . She solves the OEP if  $1 = \prod_{j=0}^m Z_j^{r_j^{-1}}$ . Equivalently,

$$\begin{aligned} 1 &= \prod_{\substack{j=0 \\ b_j=0}}^m Z_j^{\frac{u_j x}{v_j}} \prod_{\substack{j=0 \\ b_j=1}}^m Z_j^{\frac{u_j}{v_j y}} \\ 1 &= \left( \prod_{\substack{j=0 \\ b_j=0}}^m Z_j^{\frac{u_j}{v_j}} \right)^{xy} \prod_{\substack{j=0 \\ b_j=1}}^m Z_j^{\frac{u_j}{v_j}} = h^{xy} A, \end{aligned}$$

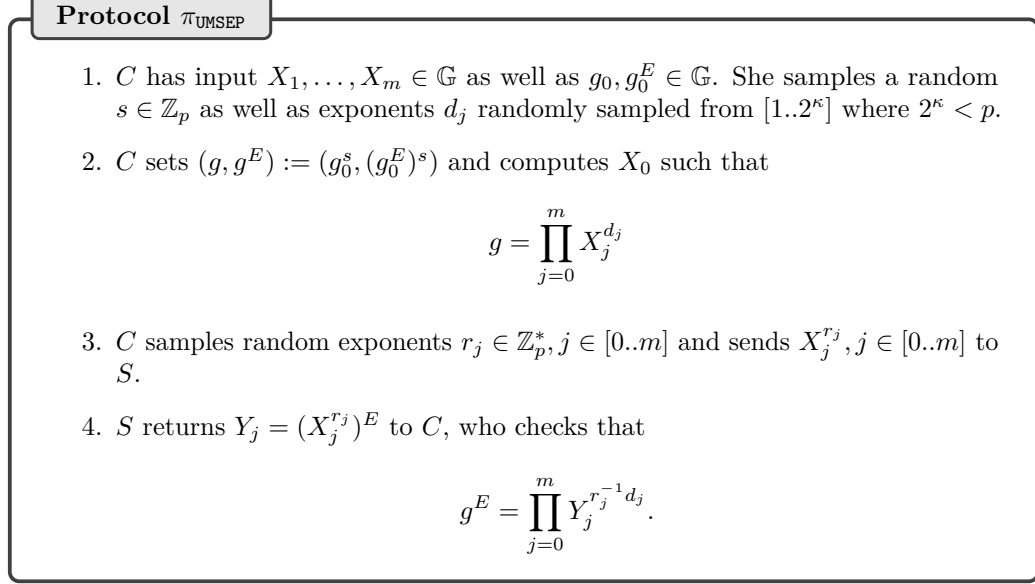
for elements  $h$  and  $A$  that the reduction can compute. The reduction then outputs both  $h$  and  $A^{-1}$ , which equals  $h^{xy}$  if the adversary wins her game. Therefore, the reduction succeeds if the adversary wins and  $h \neq 1$ .

We now argue that  $\Pr[h \neq 1 \mid \text{adversary wins}] \geq \frac{1}{2}$ . Both the adversary's view (the  $X_j$  and  $X_j^{r_j}$ ) and whether she wins are entirely determined by  $u_j$  and  $r_j$  for all  $j$ . However, the bits  $b_j$  are independent of these because  $r_j$  is completely masked by  $v_j$ . These bits are also independent from  $x$  and  $y$  for the same reason. Therefore,  $h$  will be the product of a random subset (chosen by  $b_j = 0$ ) of the group elements  $R_j = Z_j^{x^{-1} r_j^{-1}}$  (which equals  $Z_j^{u_j/v_j}$  when  $b_j = 0$ ). We have that at least one  $R_k \neq 1$ , so  $b_k = 0$  and  $b_k = 1$  must lead to two different values of  $h$  in either case, which therefore cannot both be 1. Hence,  $h$  must differ from 1 with probability at least  $\frac{1}{2}$ .  $\square$

<sup>4</sup>Note that the reduction cannot actually compute  $r_j$ .

### 3.2 An unconditionally sound protocol

We now present an alternative protocol that is unconditionally sound, but somewhat less efficient on the client side. The idea is not to use the  $r_j$  values for the random linear combination, but use independent coefficients  $d_j$  from a bounded set. Set-up and notation is the same as for  $\pi_{\text{MSEP}}$ , and the construction can be found in Fig. 4.



**Figure 4:** The Unconditionally and Maliciously Secure Exponentiation Protocol  $\pi_{\text{UMSEP}}$

Completeness of  $\pi_{\text{UMSEP}}$  follows from the fact that if both parties are honest, we have

$$\prod_j Y_j^{r_j^{-1} d_j} = \prod_j X_j^{r_j E r_j^{-1} d_j} = \prod_j (X_j^{d_j})^E = g^E$$

$\pi_{\text{UMSEP}}$  has the same overhead as  $\pi_{\text{MSEP}}$  for  $S$  but adds  $2m$  exponentiations for  $C$ . However, we shall see that the soundness error is  $2^{-\kappa}$ , and it will usually be sufficient to have  $2^\kappa \ll p$ . This means that  $d_j \ll p$  so the added exponentiations add only a small constant factor.

We proceed to show that  $\pi_{\text{UMSEP}}$  is sound:

**Lemma 5.** *If at least one of the values sent by  $S$  in  $\pi_{\text{UMSEP}}$  is incorrect, then  $C$  rejects, except with probability  $2^{-\kappa}$ .*

*Proof.* Since the group  $\mathbb{G}$  has prime order, and we assume  $X_j \neq 1$ ,  $X_j$  generates the group, so we can always write  $Y_j^{r_j^{-1}} = X_j^{E+e_j}$  where  $e_j$  is an error introduced by  $S$ , and where of course  $e_j = 0$  if  $S$  is honest. Now, the equation  $C$  checks can be rewritten as follows:

$$\begin{aligned} g^E &= \prod_{j=0}^m Y_j^{r_j^{-1} d_j} = \prod_{j=0}^m X_j^{(E+e_j) d_j} \\ &= \prod_{j=0}^m X_j^{E \cdot d_j} X_j^{e_j d_j} = \prod_{j=0}^m (X_j^{d_j})^E X_j^{e_j d_j} \\ &= g^E \prod_{j=0}^m X_j^{e_j d_j} \end{aligned}$$

So the check goes through if and only if  $\prod_{j=0}^m X_j^{e_j d_j} = 1$ . We can write each  $X_j$  as a power of, say  $g_0$ , as  $X_j = g_0^{u_j}$ . Plugging this into the condition for the errors, we see that the check goes through exactly if

$$\sum_{j=0}^m d_j \cdot (u_j e_j) \bmod p = 0$$

Clearly, the  $d_j$ 's are chosen independently of the  $u_j$ 's and the  $e_j$ 's<sup>5</sup>. Furthermore, all  $u_j$  are different from 0, since no  $X_j$  is 1. So, if some  $e_k \neq 0$  is non-zero, also  $e_k u_k \neq 0$  is non-zero. But then the above equation is satisfied only if

$$d_k = (e_k u_k)^{-1} \sum_{j \neq k} d_j \cdot u_j e_j$$

which happens with probability  $2^{-\kappa}$ . □

Using a similar argument as for Lemma 2, we can show:

**Lemma 6.** *The view of a semi-honest  $C$  executing  $\pi_{\text{UMSEP}}$  with an honest  $S$  can be perfectly simulated given  $\{X_j^E\}_{j \in [m]}$ .*

## 4 Hashed-DH secure against active attackers

In this section we describe the UC-secure Verifiable DOPRF  $\mathcal{F}_{\text{DOPRF}}$  and recap its proof of security against passive attacks in the Key Registration and Global Random Oracle model. Afterwards, we will show how to apply the techniques from Section 3 to make it secure against actively corrupted servers.

In Fig. 5 we describe a DOPRF that can be statically corrupted. For simplicity, we only allow *batch queries* by the user.

The protocol which realizes  $\mathcal{F}_{\text{DOPRF}}$  with security against passive key servers is described in Fig. 6. It follows the standard approach outlined in the introduction.

We will now prove security of  $\pi_{\text{DOPRF}}$  when both the parties and the key servers can only be passively corrupted. The proof is a standard argument that one can find e.g. in [JKKX17], and we just include it for completeness. In the next subsection, we then modify the protocol to make it secure against actively corrupted key servers.

**Theorem 1.** *The protocol  $\pi_{\text{DOPRF}}$  UC-securely implements the functionality  $\mathcal{F}_{\text{DOPRF}}$  in the  $\mathcal{G}_{\text{RD-G}}, \mathcal{F}_{\text{KeyReg}}$ -hybrid model with security against static passive corruptions assuming the DDH problem holds in  $\mathbb{G}$ .*

*Proof.* To prove the theorem, we have to construct a simulator  $\mathcal{S}$  which in the presence of  $\mathcal{A}$  as well as with access to  $\mathcal{F}_{\text{DOPRF}}$  simulates the messages of uncorrupted “honest” parties in the protocol. Since parties in our case are only passively corrupted, we assume that  $\mathcal{S}$  obtains the input and randomness of these honest parties controlled by  $\mathcal{A}$  during simulation<sup>6</sup>.

<sup>5</sup>Note that this is true despite the fact that we choose  $X_0$  so that the equation  $g = \prod_{j=0}^m X_j^{d_j}$  holds. This is because  $g$  is a fresh random group element. Therefore, an equivalent random experiment would be to choose  $X_0$  (and  $d_0$ ) independently at random and define  $g$  by  $g = \prod_{j=0}^m X_j^{d_j}$ .

<sup>6</sup>This may seem strange at first, but is straightforward when keeping in mind what a simulation proof does: it shows that the view of dishonest parties can be simulated given only their inputs and the outputs of the ideal functionality. Non-UC simulation proofs actively have to choose the randomness of dishonest parties themselves, so making this randomness and the inputs accessible to  $\mathcal{S}$  is necessary in the passive setting.

**Functionality**  $\mathcal{F}_{\text{DOPRF}}$ 

The functionality is parameterized by a group  $\mathbb{G}$  of prime order  $p$ . This functionality communicates with  $\mathcal{K}_1, \dots, \mathcal{K}_n$  as well as parties  $P = \{\mathcal{C}_1, \dots, \mathcal{C}_\ell\}$  and an ideal adversary  $\mathcal{A}$ .  $\mathcal{A}$  may initially corrupt up to  $t < n/2$  of the Key Servers as well as any party in  $P$ . We denote the corrupted key servers as  $I$ . The functionality internally stores a list  $T$  that is initially empty.

**Init:** Upon first input  $(\text{Init}, \text{sid}, i)$  by  $\mathcal{K}_i$  or  $(\text{Init}, \text{sid}, i)$  from  $\mathcal{A}$  for  $i \in I$ :

1. Send  $(\text{Init}, \text{ssid}, \mathcal{K}_i)$  to  $\mathcal{A}$ .
2. If  $t + 1$  Init messages were received then mark  $\mathcal{F}_{\text{DOPRF}}$  as ready.

**Query:** Upon input  $(\text{Query}, \text{sid}, \text{ssid}, \{q_1, \dots, q_m\})$  by party  $\mathcal{C}$  or  $\mathcal{A}$  for a previously unused  $\text{ssid}$  where  $q_j \in \{0, 1\}^*$  and if  $\mathcal{F}_{\text{DOPRF}}$  is marked as ready:

1. Send  $(\text{Query}, \text{sid}, \text{ssid}, \mathcal{C}, m)$  to each honest  $\mathcal{K}_i$  and  $\mathcal{A}$ . Wait until each honest  $\mathcal{K}_i$  and  $\mathcal{A}$  responds with  $(\text{Ok}, \text{sid}, \text{ssid})$ .
2. For any  $q_j, j \in [m]$ : if  $(\text{sid}, q_j, y'_j) \in L$  then set  $y_j \leftarrow y'_j$ . Otherwise sample  $y_j \xleftarrow{\$} \mathbb{G}$  uniformly at random and add  $(\text{sid}, q_j, y_j)$  to  $L$ .
3. Send  $(\text{Response}, \text{sid}, \text{ssid}, \{q_j, y_j\}_{j \in [m]})$  to  $\mathcal{C}$ .

**Figure 5:** Functionality  $\mathcal{F}_{\text{DOPRF}}$  representing a distributed OPRF

We construct  $\mathcal{S}$  for a fixed setting of  $I, S, m$  to simplify notation. Namely, we assume that exactly  $t$  key servers, for simplicity  $\mathcal{K}_1, \dots, \mathcal{K}_t$ , are corrupted. This is because the proof easily generalizes to other parties or smaller thresholds being corrupted (the simulator can just “pretend” that  $t$  parties are corrupted). Moreover, we assume that  $S = [t + 1]$  is chosen by each sender and that  $m = 1$ . Again, this is for the sake of simplicity and the same argument works for any choice  $S, m$ .  $\mathcal{S}$  then runs as follows:

- $\mathcal{S}$  will simulate the hybrid functionalities, i.e.  $\mathcal{G}_{\text{RO-G}}$  and  $\mathcal{F}_{\text{KeyReg}}$ .
- Whenever  $\mathcal{F}_{\text{DOPRF}}$  outputs  $(\text{Init}, \text{sid}, \mathcal{K}_j)$  to  $\mathcal{S}$  for an uncorrupted  $\mathcal{K}_j$  then send  $(\text{GenerateKey}, \text{sid})$  in the name of  $\mathcal{K}_j$  to  $\mathcal{F}_{\text{KeyReg}}$  and simulate its behavior. Whenever  $\mathcal{A}$  sends Shares for a set  $J$  to  $\mathcal{F}_{\text{KeyReg}}$  then forward Init for each party in  $J$  to  $\mathcal{F}_{\text{DOPRF}}$ .
- Whenever  $\mathcal{S}$  obtains  $(\text{Query}, \text{sid}, \text{ssid}, \mathcal{C}, 1)$  from  $\mathcal{F}_{\text{DOPRF}}$  (for an honest  $\mathcal{C}$  that queried the DOPRF) then sample a uniformly random group element  $h$  from  $\mathbb{G}$  and send  $(\text{DOPRF} - \text{Compute}, \text{sid}, \text{ssid}, h, S)$  to all dishonest key servers. Then upon obtaining the responses from the corrupt key servers, send  $(\text{Ok}, \text{sid}, \text{ssid})$  to  $\mathcal{F}_{\text{DOPRF}}$ .
- For a query  $(\text{DOPRF} - \text{Compute}, \text{sid}, \text{ssid}, h)$  from a dishonest party  $\mathcal{C}$  we have the input  $q$  and randomness  $r$  by assumption because  $\mathcal{C}$  can only be semi-honest. Then  $\mathcal{S}$  sends  $(\text{Query}, \text{sid}, \text{ssid}, \{q\})$  to  $\mathcal{F}_{\text{DOPRF}}$ . Moreover, send  $(\text{Ok}, \text{sid}, \text{ssid}, I)$ . Upon obtaining  $(\text{Response}, \text{sid}, \text{ssid}, q, y)$  we know that the dishonest key servers will generate shares  $Y_1 = L^{k^{(i)} \cdot \lambda_{1,0}^S}, \dots, Y_t = L^{k^{(t)} \cdot \lambda_{t,0}^S}$ , and we set the last share as  $Y_{t+1} = y^r / (Y_1 \cdots Y_t)$ .

Clearly,  $\mathcal{S}$  runs in polynomial time as all computations are straightforward. Towards indistinguishability, we define the following distributions:

$\mathcal{I}$ : This is the view of  $\mathcal{Z}$  in  $\mathcal{S}$ .



**Protocol**  $\pi_{\text{DOPRF}}$ 

The protocol  $\pi_{\text{DOPRF}}$  runs between Key Servers  $\mathcal{K}_1, \dots, \mathcal{K}_n$  and parties  $P = \{\mathcal{C}_1, \dots, \mathcal{C}_\ell\}$ . The protocol is defined in the  $\mathcal{G}_{\text{RO-G}}, \mathcal{F}_{\text{KeyReg}}$ -hybrid model where parties communicate via authenticated channels.

**Init:** Upon first input ( $\text{Init}, \text{sid}$ ) to a Key Server  $\mathcal{K}_i$ :

1. Send ( $\text{GenerateKey}, \text{sid}$ ) to  $\mathcal{F}_{\text{KeyReg}}$ .
2. If  $\mathcal{F}_{\text{KeyReg}}$  responds with ( $\text{GenerateKeyOk}, \text{sid}, k^{(i)}, G, G^{(1)}, \dots, G^{(t)}$ ) then store  $k^{(i)}$  locally and output  $\text{Init}$ .

**Query:** Upon input ( $\text{Query}, \text{sid}, \text{ssid}, \{q_1, \dots, q_m\}$ ) to party  $\mathcal{C}$ , where  $q_j \in \{0, 1\}^*$ :

1. Choose a set  $S \subseteq [n]$  of size  $t + 1$  uniformly at random.
2. For each  $j \in [m]$  send ( $\text{Hash-Query}, q_j$ ) to  $\mathcal{G}_{\text{RO-G}}$  to obtain ( $\text{Hash-Confirm}, X_j$ ). Also check ( $\text{IsProgrammed}, q_j$ ) and abort if  $\mathcal{G}_{\text{RO-G}}$  returns ( $\text{IsProgrammed}, q_j, 1$ ).
3. For each  $j \in [m]$  sample  $r_j \xleftarrow{\$} \mathbb{Z}_p^*$  and compute  $L_j \leftarrow X_j^{r_j}$  in  $\mathbb{G}$ .
4.  $\mathcal{C}$  sends ( $\text{DOPRF-Compute}, \text{sid}, \text{ssid}, \{L_1, \dots, L_m\}, S$ ) to each  $\mathcal{K}_i, i \in S$ .
5. Upon receiving ( $\text{DOPRF-Compute}, \text{sid}, \text{ssid}, \{L_1, \dots, L_m\}, S$ ) from  $\mathcal{C}$  for which each receiving  $\mathcal{K}_i$  has  $k^{(i)}$ ,  $\mathcal{K}_i$  computes its Lagrange coefficient  $\lambda_{i,0}^S$ , computes  $Y_{i,j} \leftarrow L_j^{k^{(i)} \cdot \lambda_{i,0}^S}$  for each  $j \in [m]$  and sends ( $\text{DOPRF-Response}, \text{sid}, \text{ssid}, \{Y_{i,1}, \dots, Y_{i,m}\}$ ) back to  $\mathcal{C}$ .
6. Upon having received ( $\text{DOPRF-Response}, \text{sid}, \text{ssid}, \{Y_{i,1}, \dots, Y_{i,m}\}$ ) from each  $\mathcal{K}_i$ ,  $\mathcal{C}$  computes and outputs  $y_j = \left( \prod_{i \in [n]} Y_{i,j} \right)^{1/r_j}$  for each  $j \in [m]$ .

**Figure 6:** Protocol  $\pi_{\text{DOPRF}}$  that implements the distributed OPRF

$\mathcal{H}_1$ : Is the same as  $\mathcal{I}$  except that we replace the random group element  $h$  being sent for honest queries with a message as it is being sent in the protocol.

$\mathcal{H}_2$ : Is the same as  $\mathcal{H}_1$  except that  $\mathcal{F}_{\text{DOPRF}}$  now uses the random value  $K \in \mathbb{Z}_p$  chosen by  $\mathcal{G}_{\text{RO-G}}$  and outputs  $H(q)^K$  instead of a uniformly random group element from  $\mathbb{G}$ .

$\mathcal{H}_3$ : Is the same as  $\mathcal{H}_2$  except that we replace  $Y_{t+1}$  with the correctly formed message according to the shared key  $K$ .

$\mathcal{R}$ : Is the view of  $Z$  in the real protocol.

Towards indistinguishability, we first note that queries of honest parties  $\mathcal{C}$  or corrupted such parties always yield the same output, so any distinguishing environment  $\mathcal{Z}$  must distinguish based on the protocol messages and distribution of protocol outputs. Observe that  $\mathbb{G}$  is of prime order so every element except 1 is a generator of  $\mathbb{G}$ . Since the output of  $\mathcal{G}_{\text{RO-G}}$  on query  $q$  is a random group element  $X$  which is a generator (except with probability  $1/p$ ),  $X^r$  is a random group element in  $\mathbb{G}$  which is not 1. Moreover, if  $X = 1$  then  $X^r = 1$  as well. Hence,  $h$  has the same distribution as the message  $H(q)^r$  and  $\mathcal{I}$  and  $\mathcal{H}_1$  are perfectly indistinguishable.

Concerning  $\mathcal{H}_1$  and  $\mathcal{H}_2$  we can make a hybrid argument, replacing (consistently for re-queries) the first query  $q_1$  to  $\mathcal{F}_{\text{DOPRF}}$  with  $H(q_1)^K$  etc. Then, any  $\mathcal{Z}$  distinguishing two

such consecutive hybrids is exactly solving the DDH problem. Concerning  $\mathcal{H}_2$  and  $\mathcal{H}_3$  observe that  $Y_{t+1}$  is uniquely determined by the constraint that  $Y = Y_1 \cdots Y_{t+1}$  so this change is just of notation and perfectly indistinguishable. But then,  $\mathcal{H}_3$  is identical to  $\mathcal{R}$  and the claim follows.  $\square$

#### 4.1 Using $\pi_{\text{MSEP}}$ in $\pi_{\text{DOPRF}}$

We use  $\pi_{\text{MSEP}}$  in a two-step process in the Hashed-DH protocol to achieve active security. First, recall that the client sends a set of blinded requests  $\{X_j^{r_j}\}$  to all the key servers and then combines the responses to form what should be  $\{(X_j^{r_j})^K\}$ . We can abstractly think of the entire process, starting from the  $X_j^{r_j}$  and ending with the  $(X_j^{r_j})^K$  values, as one (possibly corrupt) server  $S$  executing exponentiations to power  $E = K$ . If we make sure that a correct pair  $(g_0, g_0^K)$  is obtained from  $\mathcal{F}_{\text{KeyReg}}$ , we can then use the method from  $\pi_{\text{MSEP}}$  to check the final output of the client.

If this check fails, we can instead check the individual responses from the key servers, namely we note that each key server is supposed to raise the inputs it gets to a particular exponent. So we can look at the responses from each individual key server and apply the check from  $\pi_{\text{MSEP}}$ , where the key server plays the role of  $S$  and the key server's share of the global key plays the role of  $E$ . This, of course, assumes that a correct pair of form  $(g_0, g_0^{f(i)})$  is obtained from  $\mathcal{F}_{\text{KeyReg}}$ . The protocol is described in Fig. 7.

To see why protocol  $\pi_{\text{DOPRF-A}}$  will not incorrectly identify an honest key server, observe that  $(G^{(i)})^{s \cdot \lambda_{i,0}^S} = g^{k^{(i)} \cdot \lambda_{i,0}^S}$ , so by  $g = \prod_{j=0}^m X_j$  the check is true for every honest key server. Furthermore, if the per-key server check in Step 9 is reached, then we must have that at least one party will be identified - otherwise, the previous check in Step 8 trivially would have been true to begin with. In many deployments, one can expect that data from the key servers will be correct most of the time, so it pays off to optimistically check the global answer first.

**Theorem 2.** *The protocol  $\pi_{\text{DOPRF-A}}$  UC-securely implements the functionality  $\mathcal{F}_{\text{DOPRF}}$  in the  $\mathcal{G}_{\text{RO-G}}, \mathcal{F}_{\text{KeyReg}}$ -hybrid model with security against static passive corruptions of  $\mathcal{C}$  and active corruptions of  $\mathcal{K}_i$  assuming the DDH and OEP problems hold in  $\mathbb{G}$ .*

*Proof.* The proof is almost identical to the proof of Theorem 1. The only difference is that for a simulated honest party, we now only accept a response set  $Y_{i,j}$  sent by  $\mathcal{A}$  for any dishonest key server iff each  $Y_{i,j}$  is exactly  $L_j^{k^{(i)} \cdot \lambda_{i,0}^S}$ , whereas in  $\pi_{\text{DOPRF-A}}$  the honest party also accepts as long as the checks in Steps 8 and 9 hold. By adding an additional hybrid for this difference, any distinguishing environment must break the OEP problem as proven in Lemma 1. The additional message  $Y_{i,0}$  that a simulated honest key server sends to a dishonest  $\mathcal{C}$  furthermore reveals no information, as proven in Lemma 2.  $\square$

## 5 Implementation & Experiments

In this section, we report on experiments on the overhead of our approach to active security. Towards this, we have implemented the DOPRF protocol with only passive security ( $\pi_{\text{DOPRF}}$ ) as a baseline. We then implemented two versions of  $\pi_{\text{DOPRF-A}}$ , one which utilizes  $\pi_{\text{MSEP}}$  to achieve active security against corrupted Servers ( $\pi_{\text{DOPRF-A}}$ ) and one that uses  $\pi_{\text{UMSEP}}$ . For the experiments, we have not implemented the key generation functionality  $\mathcal{F}_{\text{KeyReg}}$  but instead assume distributed server verification information as a setup. The code is publicly available on <https://github.com/SecureDNA/SecureDNA>, together with scripts and instructions how to re-run the experiments (<https://github.com/SecureDNA/SecureDNA/tree/main/test/perftest>).

**Protocol**  $\pi_{\text{DOPRF-A}}$ 

The protocol  $\pi_{\text{DOPRF-A}}$  runs between Key Servers  $\mathcal{K}_1, \dots, \mathcal{K}_n$  and parties  $\mathcal{C}_1, \dots, \mathcal{C}_\ell$ . The protocol is defined in the  $\mathcal{G}_{\text{RO-G}}, \mathcal{F}_{\text{KeyReg}}$ -hybrid model where parties communicate via authenticated channels.

**Init:** Upon first input (Init,  $sid$ ) to a Key Server  $\mathcal{K}_i$ :

1. Send (GenerateKey,  $sid$ ) to  $\mathcal{F}_{\text{KeyReg}}$ .
2. If  $\mathcal{F}_{\text{KeyReg}}$  responds with (GenerateKeyOk,  $sid, k^{(i)}, G^{(0)}, \dots, G^{(n)}$ ) then store  $k^{(i)}$  locally and output Init.

**Query:** Upon input (Query,  $sid, ssid, \{q_1, \dots, q_m\}$ ) to party  $\mathcal{C}$ , where  $q_j \in \{0, 1\}^*$ :

1. Send (GetVerKey,  $sid$ ) to  $\mathcal{F}_{\text{KeyReg}}$  to obtain (VerKey,  $G^{(0)}, \dots, G^{(n)}$ ). Denote the fixed generator of  $\mathcal{F}_{\text{KeyReg}}$  as  $g_0$ .
2. Choose a set  $S \subseteq [n]$  of size  $k = t + 1$  uniformly at random.
3. For each  $j \in [m]$  send (Hash – Query,  $q_j$ ) to  $\mathcal{G}_{\text{RO-G}}$  to obtain (Hash – Confirm,  $X_j$ ). Also check (IsProgrammed,  $q_j$ ) and abort if  $\mathcal{G}_{\text{RO-G}}$  returns (IsProgrammed,  $q_j, 1$ ).
4.  $\mathcal{C}$  samples a uniformly random  $s \in \mathbb{Z}_p^*$  and computes  $(g, G) = (g_0^s, (G^{(0)})^s)$  as well as  $X_0 = g / (\prod_{j=1}^m X_j)$ .
5. For each  $j \in [0..m]$  sample  $r_j \xleftarrow{\$} \mathbb{Z}_p^*$  and compute  $L_j \leftarrow X_j^{r_j}$  in  $\mathbb{G}$ .
6.  $\mathcal{C}$  sends (DOPRF – Compute,  $sid, ssid, \{L_0, \dots, L_m\}, S$ ) to each  $\mathcal{K}_i, i \in S$ .
7. Upon receiving (DOPRF – Compute,  $sid, ssid, \{L_0, \dots, L_m\}, S$ ) from  $\mathcal{C}$  for which each receiving  $\mathcal{K}_i$  has  $k^{(i)}$ ,  $\mathcal{K}_i$  computes its Lagrange coefficient  $\lambda_{i,0}^S$ , computes  $Y_{i,j} \leftarrow L_j^{k^{(i)} \cdot \lambda_{i,0}^S}$  for each  $j \in [0..m]$  and sends (DOPRF – Response,  $sid, ssid, \{Y_{i,0}, \dots, Y_{i,m}\}$ ) back to  $\mathcal{C}$ .
8. Upon having received (DOPRF – Response,  $sid, ssid, \{Y_{i,0}, \dots, Y_{i,m}\}$ ) from each  $\mathcal{K}_i$ ,  $\mathcal{C}$  first for each  $j \in [0..m]$  computes  $Y_j = \prod_{i \in S} Y_{i,j}$ . Then she checks that  $G = \prod_{j=0}^m Y_j^{r_j^{-1}}$ . If this holds then she outputs  $Y_j^{r_j^{-1}}$  for  $j \in [m]$ .
9. If the check did not hold, then  $\mathcal{C}$  for each  $i \in S$  checks that  $(G^{(i)})^s \cdot \lambda_{i,0}^S = \prod_{j=0}^m Y_{i,j}^{r_j^{-1}}$ . It then reruns the protocol with a new set  $S'$  that does not contain the key servers for which this check did not hold.

**Figure 7:** Protocol  $\pi_{\text{DOPRF-A}}$  that implements the distributed OPRF with security against actively corrupted key servers.

**Setup.** We implemented our protocols in Rust 1.76, implementing  $\mathbb{G}$  with the library `curve25519-dalek 4.1.1`. As hash function that implements the random oracle to  $\mathbb{G}$ , we use `sha3 0.10.8` in combination with `Ristretto`.

All experiments were performed on an AMD Ryzen 9 5950X, 16 cores and 128 GiB of RAM. No GPUs or other hardware accelerators were employed. The machine is running Ubuntu 22.04.4 LTS, with protocol parties simulated as Docker containers with a virtualized network. We limited the Client and Server to 1 core and simulated network delay using `tc`.

For each experiment, we first loaded all respective parties as containers and ran the test once to avoid cache misses in the experiment. Then we ran each experiment 10 times and took the average.

**Experiments.** We conducted two types of experiments:

1. We ran both  $\pi_{\text{DOPRF}}$  and  $\pi_{\text{DOPRF-A}}$  with 0–1 ms communication delay. We implemented both the regular  $\pi_{\text{DOPRF-A}}$  and a version based on  $\pi_{\text{UMSEP}}$  with 40 bits of statistical security. For 5 Servers, we measured the cost of active security for 5.000, 10.000 and 20.000 DOPRF inputs to measure the impact of the number of inputs on active security. See Table 1 for the results.
2. We ran both  $\pi_{\text{DOPRF}}$  and  $\pi_{\text{DOPRF-A}}$  with 0–1 ms and 100 ms communication delay. For 5.000 DOPRF inputs, we measured the runtime of the protocols with 1, 3, 5, 7, 10 or 20 servers, for passive and active security. See Table 2 for the results.

**Table 1:** Time (in ms) to run DOPRF protocol with 5 Servers, 0–1 ms latency.

Inputs	5.000	10.000	20.000
$\pi_{\text{DOPRF}}$	337	677	1.405
$\pi_{\text{DOPRF-A}}$ (with $\pi_{\text{MSEP}}$ )	339	681	1.405
$\pi_{\text{DOPRF-A}}$ (with $\pi_{\text{UMSEP}}$ )	348	705	1444
Overhead (with $\pi_{\text{MSEP}}$ )	0, 59%	0, 59%	0, 00%
Overhead (with $\pi_{\text{UMSEP}}$ )	3, 26%	4, 14%	2, 78%

**Table 2:** Time (in ms) to check 5.000 inputs, different number of servers.

Servers	1	3	5	7	10	20
$\pi_{\text{DOPRF}}$ , 0–1 ms latency	253	295	337	372	421	570
$\pi_{\text{DOPRF-A}}$ (with $\pi_{\text{MSEP}}$ ), 0–1 ms latency	255	299	339	374	421	572
$\pi_{\text{DOPRF-A}}$ (with $\pi_{\text{UMSEP}}$ ), 0–1 ms latency	261	304	348	381	429	585
Overhead (with $\pi_{\text{MSEP}}$ )	0, 79%	1, 36%	0, 59%	0, 54%	0, 00%	0, 35%
Overhead (with $\pi_{\text{UMSEP}}$ )	3, 16%	3, 05%	3, 26%	2, 42%	1, 90%	2, 63%
$\pi_{\text{DOPRF}}$ , 100 ms latency	427	463	501	537	585	774
$\pi_{\text{DOPRF-A}}$ (with $\pi_{\text{MSEP}}$ ), 100 ms latency	429	466	501	537	586	765
$\pi_{\text{DOPRF-A}}$ (with $\pi_{\text{UMSEP}}$ ), 100 ms latency	446	479	514	550	595	773
Overhead (with $\pi_{\text{MSEP}}$ )	0, 47%	0, 65%	0, 00%	0, 00%	0, 17%	–1, 16%
Overhead (with $\pi_{\text{UMSEP}}$ )	4, 45%	3, 46%	2, 59%	2, 42%	1, 71%	–0, 13%

In our experiments, neither RAM nor network communication between parties was the bottleneck. The overhead for active security with  $\pi_{\text{MSEP}}$  is existent but within the noise of measurement. For active security with  $\pi_{\text{UMSEP}}$  there is a more pronounced overhead but still it’s mostly within noise. Moreover, the overhead stays essentially the same as the number of inputs is increased (see Table 1).

When running our protocols with different numbers of servers (Table 2) it can be seen, as expected, that the overhead from active security against malicious servers is essentially independent of the number of servers. Again, the overhead from  $\pi_{\text{MSEP}}$  is within noise while  $\pi_{\text{UMSEP}}$  has noticeable overhead, but consistently below 5%. It can also be seen that network latency leads to a larger variation in the noise, but doesn’t have any impact on the overhead itself. Again, this is to be expected from the protocols.

## 6 Protecting other protocols that use oblivious exponentiation

We now describe two other cryptographic protocols that use oblivious exponentiation. These can also easily be upgraded to security against active attacks during exponentiation, by following the same steps as in Section 3. Our examples are threshold blind signatures for BLS [BLS04] and threshold decryption of El Gamal [ElG85] ciphertexts.

### 6.0.1 Threshold BLS signatures

In a threshold blind signature algorithm, a client  $\mathcal{C}$  interacts with a set of  $n$  servers  $\mathcal{K}_1, \dots, \mathcal{K}_n$  to obtain a signature on a message  $q$ . The client holds a verification key  $\mathbf{vk}$ , while the servers hold a secret sharing of the corresponding signing key  $\mathbf{sk}$ . The client learns the signature  $\sigma$  from  $t + 1$  or more correct responses from the servers (and nothing else), while the servers learn no information about  $q$ .

A popular threshold blind signature algorithm can be constructed from the so-called BLS [BLS04] signature scheme. BLS uses two groups  $\mathbb{G}, \mathbb{G}_T$  equipped with a bilinear pairing  $e : \mathbb{G} \times \mathbb{G} \rightarrow \mathbb{G}_T$  as well as a hash function  $H : \{0, 1\}^* \rightarrow \mathbb{G}$  modeled as a random oracle. We assume that  $|\mathbb{G}| = p$  and  $g \in \mathbb{G}$  is a generator.

To initialize the threshold signature scheme, a key generation algorithm  $\text{KeyGen}$  samples a secret  $K \in \mathbb{Z}_p$  as well as a random degree- $t$  polynomial  $f(X)$  subject to the constraint that  $K = f(0)$ . Each server  $\mathcal{K}_i$  obtains  $k^{(i)} = f(i)$  as its share of the key, while  $\mathbf{vk} = g^K$  is the public verification key.

To sign a message  $q$  blindly,  $\mathcal{C}$  samples a random  $r \in \mathbb{Z}_p$  and sends  $L = H(q)^r$  to all servers  $\mathcal{K}_1, \dots, \mathcal{K}_n$ . Each  $\mathcal{K}$  then locally computes  $Y_i = L^{k^{(i)}}$  and sends  $Y_i$  back to  $\mathcal{C}$ . From  $t + 1$  responses  $(Y_i)_{i \in S}$ ,  $\mathcal{C}$  can reconstruct the signature as follows: it first computes the Lagrange coefficients  $\lambda_{i,0}^S$  for the set  $S$ . Then it outputs  $\sigma \leftarrow (\prod_{i \in S} Y_i^{\lambda_{i,0}^S})^{1/r}$  as the signature. To then verify  $\sigma$  using  $q, \mathbf{vk}$ , one can simply check that  $e(\mathbf{vk}, H(q)) = e(g, \sigma)$ . The threshold signing algorithm is clearly incorrect if  $\mathcal{K}_i$  ever returns  $Y'_i \neq Y_i$ .

**Adding protection against cheating servers.** Protection against a cheating  $\mathcal{K}_i$  can be achieved in the aforementioned scheme by letting the  $\text{KeyGen}$  algorithm also output  $n$  values  $\mathbf{vk}_1, \dots, \mathbf{vk}_n$  where  $\mathbf{vk}_i := g^{k^{(i)}}$ . Then,  $\mathcal{C}$  can always check if a response  $Y_i$  from  $\mathcal{K}_i$  was correct or not by testing that  $e(\mathbf{vk}_i, L) = e(g, Y_i)$ . However, this requires it to compute a pairing. When computing a batch of  $m$  signatures, one would then have to compute  $m$  pairings to verify the results.

We instead observe that the aforementioned blind signing algorithm is *identical* to the HashedDH DOPRF algorithm from Fig. 6. Therefore, we can apply the exact same approach that was described in Section 4.1 but instead with  $\pi_{\text{UMSEP}}$ . This leads to a protocol  $\pi_{\text{BLS-A}}$  with active security against the corrupted  $\mathcal{K}_i$  while only having to perform an additional  $m$  small exponentiations (to values of size at most  $2^\kappa$ ) instead of the mentioned  $m$  pairings when checking for corruptions. Due to the unconditional security of  $\pi_{\text{UMSEP}}$  the security follows from Lemma 5 even though we are now in a setting with a bilinear pairing. Following Lemma 6, the resulting algorithm does not leak any information about the key shares  $k^{(i)}$  to  $\mathcal{C}$ . We describe the full protocol  $\pi_{\text{BLS-A}}$  in Fig. 8.

### 6.0.2 Threshold ElGamal decryption

In a threshold cryptosystem, a client  $\mathcal{C}$  interacts with a set of  $n$  servers  $\mathcal{K}_1, \dots, \mathcal{K}_n$  to decrypt a ciphertext  $c$ . The public key  $\mathbf{pk}$  for the encryption scheme is known, while the servers hold a secret sharing of the corresponding secret key  $\mathbf{sk}$ . The client learns the

**Protocol**  $\pi_{\text{BLS-A}}$ 

The protocol runs between  $n$  servers  $\mathcal{K}_1, \dots, \mathcal{K}_n$  and a party  $\mathcal{C}$ . It is defined in the  $\mathcal{G}_{\text{RO-G}}, \mathcal{F}_{\text{KeyReg}}$ -hybrid model where parties communicate via authenticated channels.

**Init:** Upon first input (Init,  $sid$ ) to a Server  $\mathcal{K}_i$ :

1. Send (GenerateKey,  $sid$ ) to  $\mathcal{F}_{\text{KeyReg}}$ .
2. If  $\mathcal{F}_{\text{KeyReg}}$  responds with (GenerateKeyOk,  $sid, k^{(i)}, \text{vk}, \text{vk}_1, \dots, \text{vk}_n$ ) then store  $k^{(i)}$  locally and output Init.

**Sign:** Upon input (Sign,  $sid, ssid, \{q_1, \dots, q_m\}$ ) to party  $\mathcal{C}$ , where  $q_j \in \{0, 1\}^*$ :

1. Send (GetVerKey,  $sid$ ) to  $\mathcal{F}_{\text{KeyReg}}$  to obtain (VerKey,  $\text{vk}, \text{vk}_1, \dots, \text{vk}_n$ ). Denote the fixed generator of  $\mathcal{F}_{\text{KeyReg}}$  as  $g_0$ .
2. For each  $j \in [m]$  send (Hash-Query,  $q_j$ ) to  $\mathcal{G}_{\text{RO-G}}$  to obtain (Hash-Confirm,  $X_j$ ). Also check (IsProgrammed,  $q_j$ ) and abort if  $\mathcal{G}_{\text{RO-G}}$  returns (IsProgrammed,  $q_j, 1$ ).
3.  $\mathcal{C}$  samples uniformly random  $d_0, \dots, d_m \in [1..2^\kappa]$  and  $s \in \mathbb{Z}_p^*$  and computes  $(g, G) = (g_0^s, (\text{vk})^s)$  as well as  $X_0 = (g / (\prod_{j=1}^m X_j^{d_j}))^{1/d_0}$ .
4. For each  $j \in [0..m]$  sample  $r_j \xleftarrow{\$} \mathbb{Z}_p^*$  and compute  $L_j \leftarrow X_j^{r_j}$  in  $\mathbb{G}$ .
5.  $\mathcal{C}$  sends (BlindBLS,  $sid, ssid, \{L_0, \dots, L_m\}$ ) to each  $\mathcal{K}_i, i \in [n]$ .
6. Upon receiving (BlindBLS,  $sid, ssid, \{L_0, \dots, L_m\}$ ) from  $\mathcal{C}$ ,  $\mathcal{K}_i$  computes  $Y_{i,j} \leftarrow L_j^{k^{(i)}}$  for each  $j \in [0..m]$  and sends (BlindBLS,  $sid, ssid, \{Y_{i,0}, \dots, Y_{i,m}\}$ ) back to  $\mathcal{C}$ .
7. Upon having received (BlindBLS,  $sid, ssid, \{Y_{i,0}, \dots, Y_{i,m}\}$ ) from  $t+1$  servers denoted as the set  $S$ ,  $\mathcal{C}$  computes the Lagrange coefficients  $\lambda_{i,0}^S$  for each  $i \in S$ .
8. Then for each  $j \in [0..m]$  it computes  $Y_j = \prod_{i \in S} Y_{i,j}^{\lambda_{i,0}^S}$ . Then she checks that  $G = \prod_{j=0}^m Y_j^{r_j^{-1} d_j}$ . If this holds then she outputs  $Y_j^{r_j^{-1}}$  for  $j \in [m]$ .
9. If the check did not hold, then for each  $i \in [n]$  that sent a response,  $\mathcal{C}$  checks that  $\text{vk}_i^s = \prod_{j=0}^m Y_{i,j}^{r_j^{-1} d_j}$ . She then reconstructs the output as in the previous step, based on the Lagrange coefficients for the correct responses.

**Figure 8:** Protocol for actively secure threshold BLS blind signatures.

message  $q$  from  $t+1$  or more correct responses from the servers, while the servers learn no information about  $q$ .

We first consider a setting where we require the additional property that the servers do not learn  $c$  either. This may be important so that an adversary cannot learn which ciphertexts a client wishes to decrypt.

A popular threshold cryptosystem can be constructed from the so-called ElGamal [ElG85] asymmetric encryption scheme. ElGamal requires the use of a finite Abelian group  $\mathbb{G}$  of prime order  $p$  together with a fixed generator  $g \in \mathbb{G}$ . Messages will be elements from  $\mathbb{G}$ .

To initialize the threshold ElGamal cryptosystem, a key generation algorithm **KeyGen** samples a secret  $K \in \mathbb{Z}_p$  as well as a random degree- $t$  polynomial  $f(X)$  subject to the



constraint that  $K = f(0)$ . Each server  $\mathcal{K}_i$  obtains  $k^{(i)} = f(i)$  as its share of the key, while  $\mathbf{pk} = g^K$  is the public key. To encrypt a message  $q \in \mathbb{G}$ , one samples  $x \in \mathbb{Z}_p$  uniformly at random and outputs  $c = (g^x, q \cdot h^x)$  as the ciphertext.

To decrypt a ciphertext  $c = (c_0, c_1)$  without leaking it to any  $\mathcal{K}_i$ ,  $\mathcal{C}$  samples  $r \in \mathbb{Z}_p$  uniformly at random, computes  $L = c_0^r$  and sends it to each  $\mathcal{K}_i$ . Each  $\mathcal{K}_i$  then returns  $Y_i := L^{k^{(i)}}$  to  $\mathcal{C}$ . Using Lagrange interpolation, after having obtained responses  $(Y_i)_{i \in S}$  where  $|S| = t + 1$ , the client computes the Lagrange coefficients  $\lambda_{i,0}^S$  and outputs  $c_1 \cdot (\prod_{i \in S} Y_i^{\lambda_{i,0}^S})^{-1/r}$ .

The protocol can easily be shown to be passively secure against any attacker corrupting at most  $t$  parties. It is also clear that the decryption is not correct as soon as a corrupt  $\mathcal{K}_i$  outputs a value  $Y_i' \neq Y_i$ .

**Adding protection against cheating servers.** To protect batch ElGamal decryption against active corruptions, we again observe that the passively secure decryption algorithm is essentially the same as  $\pi_{\text{DOPRF}}$ . We can therefore modify it as described in Section 4.1:

- We extend the public key to  $\mathbf{pk} = (h, G_1, \dots, G_n)$  where  $G_i = g^{k^{(i)}}$ .
- To decrypt  $m$  ciphertexts  $c^{(1)} = (c_0^{(1)}, c_1^{(1)}), \dots, c^{(m)} = (c_0^{(m)}, c_1^{(m)})$ , the decrypting party runs the protocol  $\pi_{\text{MSEP}}$  on inputs  $X_i = c_1^{(i)}$  and  $(g_0, g_0^E) := (g, G)$  with each server  $\mathcal{K}_j$ . After obtaining  $t + 1$  accepting instances, it decrypts as before.

Using Lemmas 1 and 2 one can trivially show that this protocol modification does not leak any additional information about the secret key shares  $k^{(i)}$  to  $\mathcal{C}$ , while the output must be correct assuming hardness of the CDH problem in the group  $\mathbb{G}$ .

Finally, we consider the case where it is not required that the ciphertext to decrypt is hidden from the servers. Here, we can instead use  $\pi_{\text{UMSEP}}$ . This does introduce a computational overhead for the client compared to the obvious semi-honest solution, in that it needs to do an exponentiation with a small exponent for each ciphertext to decrypt. However, all other overheads are  $o(1)$ , and compared to the naive actively secure protocol using standard zero-knowledge proofs, the client does much less work in  $\pi_{\text{UMSEP}}$ . This is because verification of each zero-knowledge proof requires a full-scale exponentiation.

## Acknowledgements

Financial support was obtained from the Open Philanthropy Project (to MIT and Aarhus University), an anonymous philanthropist from mainland China (to Tsinghua University), the Aphorism Foundation (to MIT), and Effective Giving (to MIT). The funders had no role in the writing of this work.

## References

- [AC20] Thomas Attema and Ronald Cramer. Compressed  $\Sigma$ -protocol theory and practical application to plug & play secure algorithmics. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology – CRYPTO 2020, Part III*, volume 12172 of *Lecture Notes in Computer Science*, pages 513–543. Springer, Cham, August 2020. doi:10.1007/978-3-030-56877-1\_18.
- [AMMM18] Shashank Agrawal, Peihan Miao, Payman Mohassel, and Pratyay Mukherjee. PASTA: PASSword-based threshold authentication. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th*

- Conference on Computer and Communications Security*, pages 2042–2059. ACM Press, October 2018. doi:10.1145/3243734.3243839.
- [AP05] Michel Abdalla and David Pointcheval. Simple password-based encrypted key exchange protocols. In Alfred Menezes, editor, *Topics in Cryptology – CT-RSA 2005*, volume 3376 of *Lecture Notes in Computer Science*, pages 191–208. Springer, Berlin, Heidelberg, February 2005. doi:10.1007/978-3-540-30574-3\_14.
- [BBB<sup>+</sup>18] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy*, pages 315–334. IEEE Computer Society Press, May 2018. doi:10.1109/SP.2018.00020.
- [BBC<sup>+</sup>24] Carsten Baum, Jens Berlips, Walther Chen, Hongrui Cui, Ivan Damgard, Jiangbin Dong, Kevin M. Esvelt, Leonard Foner, Mingyu Gao, Dana Gretton, Martin Kysel, Juanru Li, Xiang Li, Omer Paneth, Ronald L. Rivest, Francesca Sage-Ling, Adi Shamir, Yue Shen, Meicen Sun, Vinod Vaikuntanathan, Lynn Van Hauwe, Theia Vogel, Benjamin Weinstein-Raun, Yun Wang, Daniel Wichs, Stephen Wooster, Andrew C. Yao, Yu Yu, Haoling Zhang, and Kaiyi Zhang. A system capable of verifiably and privately screening global dna synthesis, 2024. URL: <https://arxiv.org/abs/2403.14023>, arXiv:2403.14023.
- [BFF<sup>+</sup>19] Gilles Barthe, Edvard Fagerholm, Dario Fiore, John C. Mitchell, Andre Scedrov, and Benedikt Schmidt. Automated analysis of cryptographic assumptions in generic group models. *Journal of Cryptology*, 32(2):324–360, April 2019. doi:10.1007/s00145-018-9302-3.
- [BFH<sup>+</sup>20] Carsten Baum, Tore Frederiksen, Julia Hesse, Anja Lehmann, and Avishay Yanai. Pesto: proactively secure distributed single sign-on, or how to trust a hacked server. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 587–606. IEEE, 2020. doi:10.1109/EuroSP48549.2020.00044.
- [BGR98] Mihir Bellare, Juan A. Garay, and Tal Rabin. Fast batch verification for modular exponentiation and digital signatures. In Kaisa Nyberg, editor, *Advances in Cryptology – EUROCRYPT’98*, volume 1403 of *Lecture Notes in Computer Science*, pages 236–250. Springer, Berlin, Heidelberg, May / June 1998. doi:10.1007/BFb0054130.
- [BLS04] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. *Journal of Cryptology*, 17(4):297–319, September 2004. doi:10.1007/s00145-004-0314-9.
- [Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science*, pages 136–145. IEEE Computer Society Press, October 2001. doi:10.1109/SFCS.2001.959888.
- [CDG<sup>+</sup>18] Jan Camenisch, Manu Drijvers, Tommaso Gagliardoni, Anja Lehmann, and Gregory Neven. The wonderful world of global random oracles. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018, Part I*, volume 10820 of *Lecture Notes in Computer Science*, pages 280–312. Springer, Cham, April / May 2018. doi:10.1007/978-3-319-78381-9\_11.

- [Cha91] David Chaum. Zero-knowledge undeniable signatures. In Ivan Damgård, editor, *Advances in Cryptology – EUROCRYPT’90*, volume 473 of *Lecture Notes in Computer Science*, pages 458–464. Springer, Berlin, Heidelberg, May 1991. doi:10.1007/3-540-46877-3\_41.
- [CHL22] Silvia Casacuberta, Julia Hesse, and Anja Lehmann. Sok: Oblivious pseudo-random functions. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*, pages 625–646. IEEE, 2022. doi:10.1109/EuroSP53844.2022.00045.
- [CLN15] Jan Camenisch, Anja Lehmann, and Gregory Neven. Optimal distributed password verification. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015: 22nd Conference on Computer and Communications Security*, pages 182–194. ACM Press, October 2015. doi:10.1145/2810103.2813722.
- [DGS<sup>+</sup>18] Alex Davidson, Ian Goldberg, Nick Sullivan, George Tankersley, and Filippo Valsorda. Privacy pass: Bypassing internet challenges anonymously. *Proceedings on Privacy Enhancing Technologies*, 2018(3):164–180, July 2018. doi:10.1515/popets-2018-0026.
- [DY05] Yevgeniy Dodis and Aleksandr Yampolskiy. A verifiable random function with short proofs and keys. In Serge Vaudenay, editor, *PKC 2005: 8th International Workshop on Theory and Practice in Public Key Cryptography*, volume 3386 of *Lecture Notes in Computer Science*, pages 416–431. Springer, Berlin, Heidelberg, January 2005. doi:10.1007/978-3-540-30580-4\_28.
- [ECS<sup>+</sup>15] Adam Everspaugh, Rahul Chatterjee, Samuel Scott, Ari Juels, and Thomas Ristenpart. The pythia PRF service. In Jaeyeon Jung and Thorsten Holz, editors, *USENIX Security 2015: 24th USENIX Security Symposium*, pages 547–562. USENIX Association, August 2015.
- [ElG85] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, 31(4):469–472, 1985. doi:10.1109/TIT.1985.1057074.
- [GLSY04] Rosario Gennaro, Darren Leigh, R. Sundaram, and William S. Yerazunis. Batching Schnorr identification scheme with applications to privacy-preserving authorization and low-bandwidth communication devices. In Pil Joong Lee, editor, *Advances in Cryptology – ASIACRYPT 2004*, volume 3329 of *Lecture Notes in Computer Science*, pages 276–292. Springer, Berlin, Heidelberg, December 2004. doi:10.1007/978-3-540-30539-2\_20.
- [GWE<sup>+</sup>24] Dana Gretton, Brian Wang, Rey Edison, Leonard Foner, Jens Berlips, Theia Vogel, Martin Kysel, Walther Chen, Francesca Sage-Ling, Lynn Van Hauwe, Stephen Wooster, Benjamin Weinstein-Raun, Erika A. DeBenedictis, Andrew B. Liu, Emma Chory, Hongrui Cui, Xiang Li, Jiangbin Dong, Andres Fabrega, Christianne Dennison, Otilia Don, Cassandra Tong Ye, Kaveri Uberoy, Ronald L. Rivest, Mingyu Gao, Yu Yu, Carsten Baum, Ivan Damgard, Andrew C. Yao, and Kevin M. Esvelt. Random adversarial threshold search enables automated dna screening, 2024. URL: <https://www.biorxiv.org/content/early/2024/04/02/2024.03.20.585782>, arXiv:<https://www.biorxiv.org/content/early/2024/04/02/2024.03.20.585782.full.pdf>, doi:10.1101/2024.03.20.585782.

- [HL08] Carmit Hazay and Yehuda Lindell. Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries. In Ran Canetti, editor, *TCC 2008: 5th Theory of Cryptography Conference*, volume 4948 of *Lecture Notes in Computer Science*, pages 155–175. Springer, Berlin, Heidelberg, March 2008. doi:[10.1007/978-3-540-78524-8\\_10](https://doi.org/10.1007/978-3-540-78524-8_10).
- [IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *Advances in Cryptology – CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 145–161. Springer, Berlin, Heidelberg, August 2003. doi:[10.1007/978-3-540-45146-4\\_9](https://doi.org/10.1007/978-3-540-45146-4_9).
- [JKK14] Stanislaw Jarecki, Aggelos Kiayias, and Hugo Krawczyk. Round-optimal password-protected secret sharing and T-PAKE in the password-only model. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology – ASIACRYPT 2014, Part II*, volume 8874 of *Lecture Notes in Computer Science*, pages 233–253. Springer, Berlin, Heidelberg, December 2014. doi:[10.1007/978-3-662-45608-8\\_13](https://doi.org/10.1007/978-3-662-45608-8_13).
- [JKKX17] Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. TOPSS: Cost-minimal password-protected secret sharing based on threshold OPRF. In Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi, editors, *ACNS 17: 15th International Conference on Applied Cryptography and Network Security*, volume 10355 of *Lecture Notes in Computer Science*, pages 39–58. Springer, Cham, July 2017. doi:[10.1007/978-3-319-61204-1\\_3](https://doi.org/10.1007/978-3-319-61204-1_3).
- [KKRT16] Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, and Ni Trieu. Efficient batched oblivious PRF with applications to private set intersection. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, pages 818–829. ACM Press, October 2016. doi:[10.1145/2976749.2978381](https://doi.org/10.1145/2976749.2978381).
- [MPR<sup>+</sup>20] Peihan Miao, Sarvar Patel, Mariana Raykova, Karn Seth, and Moti Yung. Two-sided malicious security for private intersection-sum with cardinality. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology – CRYPTO 2020, Part III*, volume 12172 of *Lecture Notes in Computer Science*, pages 3–33. Springer, Cham, August 2020. doi:[10.1007/978-3-030-56877-1\\_1](https://doi.org/10.1007/978-3-030-56877-1_1).
- [NPR99] Moni Naor, Benny Pinkas, and Omer Reingold. Distributed pseudo-random functions and KDCs. In Jacques Stern, editor, *Advances in Cryptology – EUROCRYPT’99*, volume 1592 of *Lecture Notes in Computer Science*, pages 327–346. Springer, Berlin, Heidelberg, May 1999. doi:[10.1007/3-540-48910-X\\_23](https://doi.org/10.1007/3-540-48910-X_23).
- [NR04] Moni Naor and Omer Reingold. Number-theoretic constructions of efficient pseudo-random functions. *Journal of the ACM (JACM)*, 51(2):231–262, 2004. doi:[10.1145/972639.972643](https://doi.org/10.1145/972639.972643).
- [Pei06] Chris Peikert. On error correction in the exponent. In Shai Halevi and Tal Rabin, editors, *TCC 2006: 3rd Theory of Cryptography Conference*, volume 3876 of *Lecture Notes in Computer Science*, pages 167–183. Springer, Berlin, Heidelberg, March 2006. doi:[10.1007/11681878\\_9](https://doi.org/10.1007/11681878_9).

- [Szy06] Michael Szydło. A note on chosen-basis decisional Diffie-Hellman assumptions. In Giovanni Di Crescenzo and Avi Rubin, editors, *FC 2006: 10th International Conference on Financial Cryptography and Data Security*, volume 4107 of *Lecture Notes in Computer Science*, pages 166–170. Springer, Berlin, Heidelberg, February / March 2006. doi:[10.1007/11889663\\_14](https://doi.org/10.1007/11889663_14).