UNIVERSITY OF CALIFORNIA,
IRVINE

Password-Protected Cryptosystems

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Computer Science

by

Jiayu Xu

Thesis Committee:
Professor Stanislaw Jarecki, Chair
Professor Michael Goodrich
Professor Alfred Kobsa

2015

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

First of all, I would like to thank my committee chair, Professor Stanislaw Jarecki, who has advised be for more than a year. He is a passionate teacher and a patient supervisor, and I always enjoy learning from him and discussing with him. In addition, he led me into the wonderful world of cryptography; without his help this thesis would not have been possible.

I would also like to thank my committee members, Professor Michael Goodrich and Professor Alfred Kobsa. While reading this thesis, they raised valuable comments, recommendations and questions, which guide me to further thoughts on this topic.

ABSTRACT OF THE THESIS

Password-Protected Cryptosystems

By

Jiayu Xu

Master of Science in Computer Science

University of California, Irvine, 2015

Professor Stanislaw Jarecki, Chair

Password-Protected Secret-Sharing (PPSS) schemes (parameterized by t and n) are a type of cryptographic protocols executed by a user and n servers, where the user shares a secret key (which in turn protects some secret information) among the servers and can reconstruct the key by interacting with $t + 1$ servers later. Any efficient adversary who breaks into t servers learns no information about the user's key. In this work, we take an existing PPSS scheme and improve it in terms of computational costs. We analyze the security of the new scheme in detail.

The PPSS scheme uses a functionality called Oblivious PRF (OPRF) as a crucial block. In addition, OPRF itself might be of interest in other areas. We formalize this OPRF functionality in the Universal-Composability (UC) framework, and realize it efficiently. This forms an improvement of a previous realization. We reduce the computational costs significantly, without changing the basic functions of OPRF.

Chapter 1

Introduction

It is well-known that passwords have been widely used to protect data privacy. Suppose a user U has some secret information s. A simple approach is to store s, or some long keys protecting its security, at a remote server S. Once U wants to extract its information, he gets access to S using a short password pw. However, this scheme is inherently vulnerable to off-line dictionary attacks: if an adversary manages to break into S, he can simply guess a password pw' in the space of all possible passwords D (the "dictionary"), run the user's protocol locally on pw', and check whether his guess is correct (i.e. whether $\mathsf{pw}' = \mathsf{pw}$). Although the dictionary D is typically large, the fact that the entropy of pw is usually low makes such attack above possible and even easy. A huge number of passwords are leaked in this way every year.

One possible improvement to increase the security is to store the secret information s at a set of remote servers $S_1, ..., S_n$, and then get access to them using a single password. If we do this in a naive way, then things become even worse: compromising any of the servers will lead to the leakage of s. Therefore, some more complex schemes are needed. In particular, each S_i $(i = 1, ..., n)$ should store a share of s rather than s itself, and U should reconstruct s after

collecting the shares by interacting with $S_1, ..., S_n$. Such scheme, called Password-Protected Secret Sharing (PPSS), is defined in [1] and [7]. A PPSS scheme is parameterized by a pair (t, n), where $0 \leq t \leq n-1$. The user, who holds some secret information, communicates with n servers. For any polynomial-time adversary who compromises and controls t of the servers, the only way to learn some information on pw is to guess the password correctly and run an online attack using it. Comparing with the naive scheme mentioned above, PPSS improves its security significantly. More details follow.

1.1 Prior Work and Our Contributions

This work is based on [7], and consists of two parts. [7] presents a round-optimal PPSS scheme, i.e. there is only one message sent from user to server, and one from server to user. One crucial block of the scheme is a functionality called Verifiable Oblivious PRF (V-OPRF) in the random oracle model. We improve the construction of V-OPRF by significantly decreasing the communication cost between user and sender while remaining its basic functions. The new functionality is simply named Oblivious PRF (OPRF). Our second contribution is applying OPRF to build a PPSS scheme. We discuss these two parts separately below.

OPRF. Intuitively, an OPRF is a functionality between some users, who hold values to compute on, and senders, who hold keys of some PRF families. After running the protocol, the user gets the PRF value, while the sender learns nothing.

The concept of OPRF generates from [5]. In [5], an OPRF is defined as a two-party protocol between a user (client) U and a sender (server) S:

- Inputs: U inputs a value x; S inputs a key k of some PRF family f.

- Outputs: U outputs $f_k(x)$; S outputs nothing.

As pointed out in [7], this notion does not meet our requirements, mainly because of two reasons: (1) In our PPSS scheme, we need concurrent OPRF computations, which is not supported here, and (2) In our PPSS scheme, there is no guarantee of authenticated channels, which is required in the definition above. Therefore, [7] introduces the notion of verifiable OPRF as an ideal functionality in the Universal Composability (UC) framework (introduced in [3]). Loosely speaking, the functionality $\mathcal{F}_{\text{VOPRF}}$ involves multiple users denoted as U and senders denoted as S, and the input and output interfaces are as follows:

- Inputs: U inputs a value x and a specific sender S; each sender registers a unique public parameter.
- Outputs: There are three cases: (1) The ideal world adversary $\mathcal{A}^*$ lets the computation pass: U outputs (π, ρ) where π is the parameter of S and ρ is $f_\pi(x)$, a random string. (2) $\mathcal{A}^*$ wants to prevent the computation: U outputs $(\pi, \perp)$. (3) $\mathcal{A}^*$ interferes the computation process and replaces $f_\pi(x)$ with some $M(x)$ where M is a circuit registered previously. U outputs $(\pi, M(x))$.

[7] presents three efficient realizations in the Random Oracle Model (ROM), all using two hash functions:

- 2HashDH-NIZK: assume the one-more Diffie-Hellman (DH) assumption (introduced in [2]) over the underlying group.
- 2HashDH-GAP: assume the one-more gap DH assumption over the underlying group.
- 2HashRSA: assume the one-more RSA assumption (introduced in [2]) over the underlying group.

We improve 2HashDH-NIZK in terms of computational costs, and does not consider the other two protocols here. That is due to the fact that 2HashDH-NIZK has two advantages over 2HashDH-GAP and 2HashRSA. First, the one-more DH assumption is weaker than both one-more gap DH assumption and one-more RSA assumption; therefore, 2HashDH-NIZK is potentially the most secure one among all three protocols provided. Second, in general the bandwidthes of a gap DH group and an RSA group are much larger than that of a DH group, so 2HashDH-NIZK is the most efficient.

In 2HashDH-NIZK, a Non-Interactive Zero-Knowledge (NIZK) proof is sent from the sender to the user. Since there are only two elements in a cyclic group transmitted between the two parties besides the proof itself, the NIZK proof forms a great burden. Dropping it will significantly decrease the number of bits sent between the user and the sender; therefore, it is very attractive. Naturally, deleting the NIZK proof forces us to modify the functionality itself, but we can do this without changing its basic functions.

PPSS. [7] presents a PPSS scheme for any $0 \leq t \leq n-1$ using functionality $\mathcal{F}_{\text{VOPRF}}$ as a block. This work follows the same route: we construct a PPSS scheme, which is similar to the one in [7], using $\mathcal{F}_{\text{OPRF}}$. The basic idea of the scheme generates from [4], which we briefly introduce in an informally here:

Suppose U wants to protect his secret key K using password pw. U secret-shares K with a randomly chosen string s into $(s_1, ..., s_n)$, encrypts s_i under $f_i(\mathsf{pw})$ using one-time pad (where $f_i(\cdot)$ is the PRF corresponding to S_i), collects the ciphertexts $\boldsymbol{e} := (e_1, ..., e_n)$, and sends them to the senders. To reconstruct K, each server sends $\boldsymbol{e}$ back to U, and U decrypts e_i under $f_i(\mathsf{pw})$, collects the message s_i, and recovers K from s_i's. In addition, U also commits to the ciphertexts $\boldsymbol{e}$ as well as the password, and sends the result C to the senders as well; in the reconstruction process, the senders sends C back to U together with $\boldsymbol{e}$, and U checks C is the valid commitment of pw and $\boldsymbol{e}$. Note that in order to recover K, U needs to

get access to $t+1$ among the n servers; therefore, any adversary who breaks into t servers is not able to recover U's key K. The commitment C is used to prevent the adversary from sending some other $\boldsymbol{e}'$ instead of $\boldsymbol{e}$; in case this happens, U's check will not pass.

Note that in $\mathcal{F}_{\text{OPRF}}$, U is not able to verify the sender on which he computes the PRF value of pw. This enables a type of attacks which do not occur in the scheme in [7], i.e. the adversary can manipulate the $f_i(\mathsf{pw}$ values by using some other server in place of S_i. Therefore, the security proof is more complicated than that in [7]. However, all other parts in the proof are similar.

1.2 Organization

Chapter 2 introduces the OPRF functionality, $\mathcal{F}_{\text{OPRF}}$, defined in the UC framework. Chapter 3 gives an efficient realization of $\mathcal{F}_{\text{OPRF}}$ in the random oracle model called 2HashDH. Chapter 4 reviews the definition of PPSS and its properties, especially security. Chapter 5 introduces a realization of PPSS. We analyze the scheme's security in detail.

Chapter 2

The Functionality $\mathcal{F}_{\text{OPRF}}$

We present our $\mathcal{F}_{\text{OPRF}}$ functionality below in Figure 2.1. In this functionality, the parties besides the $\mathcal{F}_{\text{OPRF}}$ (henceforth $\mathcal{F}$) itself are as follows: some users denoted by U, some senders denoted by S, and an ideal world adversary $\mathcal{A}^*$. Let l be the security parameter, i.e. the length of the output. After initialization, there are two phases: key generation and evaluation. Now we briefly explain how $\mathcal{F}_{\text{OPRF}}$ works, and analyze the differences between $\mathcal{F}_{\text{OPRF}}$ and $\mathcal{F}_{\text{VOPRF}}$ in [7]. The number of senders is denoted by P, which is initialized as 0. Each sender is associated with a unique pointer, which is an integer between 1 and P. A table T is used to record the function values for all senders: $T(p, x)$ is defined as $f_p(x)$, where $f_p(\cdot)$ is the function corresponding to the sender whose pointer is p. T is initialized as empty.

The Key Generation process reflects the process of sender registration. There are two kinds of senders: honest senders and corrupt senders generated by the adversary $\mathcal{A}^*$. In either case, when a sender registers, a new pointer is assigned, and $\mathcal{A}^*$ knows the correlation between senders and pointers.

The Evaluation process cycles through three activations: Eval that provides interface

Initialization. Set $P := 0$. Initialize functions $\mathsf{pointer}$ and T to "undefined" on all inputs.

Key Generation.

- Upon receiving (KeyGen, *sid*) from an honest sender S, if $\mathsf{pointer}(S)$ is not defined then set $P := P + 1$, $\mathsf{tickets}(P) := 0$ and $\mathsf{pointer}(S) := P$; send (KeyGen, *sid*) to S and (KeyGen, *sid*, S, P) to $\mathcal{A}^*$. Otherwise ignore this message.
- Upon receiving (KeyGen, *sid*) from $\mathcal{A}^*$, set $P := P+1$; insert P in $\mathsf{CorruptPointers}$; send (KeyGen, *sid*, P) to $\mathcal{A}^*$.

Evaluation.

- Upon receiving (Eval, *sid*, S, x) from user U, record $\langle U, x\rangle$ and send (Eval, *sid*, U, S) to $\mathcal{A}^*$.
- Upon receiving (SenderComplete, *sid*, S) from $\mathcal{A}^*$, if S is honest and $\mathsf{pointer}(S) = p$ for some $p \in \{1, ..., P\}$ then set $\mathsf{tickets}(p) := \mathsf{tickets}(p) + 1$; send (SenderComplete, *sid*) to S. Otherwise ignore this message.
- Upon receiving (UserComplete, *sid*, U, p) from $\mathcal{A}^*$, if $p \notin \{1, ..., P\}$ or $\mathsf{tickets}(p) = 0$ then ignore this message. Otherwise recover $\langle U, x\rangle$ stored in the Eval process and
  - If $T(p, x)$ is defined, then send (Eval, $T(p, x)$) to U.
  - Otherwise sample ρ at random from $\{0, 1\}^l$, set $T(p, x) := \rho$ and send (Eval, ρ) to U. If $\mathsf{pointer}(S) = p$ for some honest S, then also set $\mathsf{tickets}(p) := \mathsf{tickets}(p) - 1$.

Figure 2.1: Functionality $\mathcal{F}_{\text{OPRF}}$.

between U and $\mathcal{F}$, SenderComplete that denotes the completion of S's computation, and UserComplete that denotes the completion U's computation. Note that in the whole process, no information about U's input x is leaked to $\mathcal{A}^*$.

When U wants to compute the function value of x on a specific sender S, $\mathcal{A}^*$ is able to influence the function value by choosing between two options:

- Let the computation pass and output $f_p(x)$ (where p is the pointer of S), which is a random string in $\{0, 1\}^l$;

- Interfere the computation process and let it output $f_{p'}(x)$ instead, where p' is any other pointer. This can be further divided into two cases. First, $\mathcal{A}^*$ may use the pointer of an existing sender, either honest or corrupt. Second, $\mathcal{A}^*$ may create a fresh corrupt sender by going through the Key Generation process, and use the pointer of this sender. In either case, if $f_{p'}(x)$ is undefined, a new random string in $\{0,1\}^l$ is assigned to it.

Also note that $\mathcal{A}^*$ is able to generate an arbitrary number of corrupt users, and each time when such a user completes an interaction with an honest sender, $\mathcal{A}^*$ gets one function value of that sender.

One critical technique in the construction of this functionality is the "ticket machanism," which is introduced in the notion of $\mathcal{F}_{\text{VOPRF}}$ in [7]. Each honest sender is assigned a "ticket" value when it registers, which is 0 at the beginning (corrupt senders have no "ticket" value). In the Evaluation process, each time when an honest sender completes his interaction with a user, the corresponding ticket increases by 1; each time when a user, either honest or corrupt, completes his interaction with an honest sender, the corresponding ticket decreases by 1, provided that it is non-zero. In this way, the number of function values of honest senders that the ideal world adversary can get is limited.

There are two major differences between our functionality and $\mathcal{F}_{\text{VOPRF}}$ in [7]. First, in $\mathcal{F}_{\text{VOPRF}}$, each sender has a unique public parameter π, and when the computation ends, the user gets the function value as well as π (i.e. receives (EVAL, π, ρ) from $\mathcal{F}_{\text{VOPRF}}$). The user may or may not know the correspondence between senders and their parameters; if he knows (as in the case of the PPSS scheme in [7]), he can check whether the function value is computed via the correct PRF (i.e. the PRF of the sender he denotes at the beginning of Evaluation), and abort if it is not. In $\mathcal{F}_{\text{OPRF}}$, there is no π (replaced by an pointer p which is an integer), and only the function value is returned to the user, so the user cannot do the verification anymore.

Second, in $\mathcal{F}_{\text{VOPRF}}$, each time the ideal world adversary $\mathcal{A}^*$ registers a new corrupt sender, a circuit M is send to the functionality as well, and when $\mathcal{A}^*$ decides to "interfere" in the Evaluation process using this corrupt sender, the function value is determined as $M(x)$. In other words, $\mathcal{A}^*$ is able to control the functions of corrupt senders. In $\mathcal{F}_{\text{OPRF}}$, $M(x)$ is replaced by a fresh random string; therefore, for corrupt senders here, their functions are random.

Note that $\mathcal{F}_{\text{OPRF}}$ is weaker than $\mathcal{F}_{\text{VOPRF}}$ in the sense that the user cannot verify the sender in the end, but it is stronger in the sense that the adversary is not able to control the functions of corrupt senders.

Chapter 3

Realization of $\mathcal{F}_{\text{OPRF}}$

We present an efficient realization of $\mathcal{F}_{\text{OPRF}}$ in the random oracle model, based on the 2HashDH-NIZK construction of $\mathcal{F}_{\text{VOPRF}}$ in [7].

This construction relies on a cyclic group of prime order m. Let g be a generator of the group. The private key k is chosen at random from $\mathbb{Z}_m$. Each user U maintains a table T_U which consists of tuples of the form (S, x, r, f). Let $\mathcal{Z}$ be the environment. The construction uses two hash functions, H_1 and H_2.

The PRF is defined as $f_k(x) = H_2(x, H_1(x)^k)$. The protocol is simple: for each value x the user U wants to evaluate on, U picks a random element r in $\mathbb{Z}_m$, which remains the same among different senders. When U wants to compute $f_k(x)$ where k is the private key of a specific sender S, he sends $a = H_1(x)^r$ to S; S sends back $b = a^k = H_1(x)^{rk}$ to U, and U outputs $f = H_2(x, b^{1/r}) = H_2(x, H_1(x)^k)$.

Protocol 2HashDH is essentially the same with protocol 2HashDH-NIZK in [7], except that in 2HashDH-NIZK, each sender S with private key k is also assigned a public key $y = g^k$, and in the computation, S sends a Non-Interactive Zero-Knowledge (NIZK) proof of the

- Upon receiving (EVAL, S, x) from $\mathcal{Z}$, U scans its table T_U and does the following:
  - If there exists r and f such that $(S, x, r, f) \in T_U$, then U outputs f.
  - Otherwise if there exists S', r and f such that $S' \neq S$ and $(S', x, r, f) \in T_U$, then U sends $a = H_1(x)^r$ to S.
  - Otherwise U picks r at random from $\mathbb{Z}_m$ and sends $a = H_1(x)^r$ to S; adds $(S, x, r, *)$ to T_U.
- Upon receiving a from U, S sends $b = a^k$ to U where k is his key and outputs (SENDERCOMPLETE, sid) to $\mathcal{Z}$.
- Upon receiving b from S, U outputs (EVAL, $f = H_2(x, b^{1/r})$) to $\mathcal{Z}$; finds $(S, x, r, *)$ in T_U and changes it to (S, x, r, f).

Figure 3.1: Protocol 2HashDH.

fact DDH(g, y, a, b) (i.e. $DL(g, y) = DL(a, b)$ where DL is discrete log in group $\langle g \rangle$) ζ along with b to U, and then U verifies ζ. U continues computing only if the test passes. In our protocol, ζ is simply dropped. Intuitively, ζ allows U to check whether the value he receives comes from the sender with public key y; this corresponds to the situation in $\mathcal{F}_{\text{VOPRF}}$ where U gets the sender's public parameter π from $\mathcal{F}$, and aborts if π does not corresponds to the sender which U specifies at the beginning of Evaluation. Therefore, when there is no ζ anymore, in the functionality U loses the ability of verifying the public parameter (in fact there is no public parameter anymore). That explains one difference between $\mathcal{F}_{\text{VOPRF}}$ and $\mathcal{F}_{\text{OPRF}}$. Since we drop the NIZK proof, we simply name our protocol as 2HashDH.

We prove the security of the 2HashDH under the (N, Q) one-more Diffie-Hellman (DH) assumption: for any polynomial-time adversary $\mathcal{A}$,

$$\Pr_{k \leftarrow \mathbb{Z}_m, g_1, ..., g_N \leftarrow \langle g \rangle}[\mathcal{A}^{(\cdot)^k, \mathsf{DDH}(\cdot,\cdot,\cdot,\cdot)}(g, g^k, g_1, ..., g_N) = (g_{j_s}, g_{j_s}^k) | s = 1, ..., Q+1]$$

is negligible, where Q is the number of queries to the $(\cdot)^k$ oracle by $\mathcal{A}$ and $j_s \in \{1, ..., N\}$ $(s = 1, ..., Q+1)$. In words, suppose $\mathcal{A}$ is equipped with a "the kth power" oracle and a

DDH oracle, and its number of queries to the former is limited by Q. $\mathcal{A}$ is given N random elements in $\langle g \rangle$. Since $\mathcal{A}$ is allowed to query the $(\cdot)^k$ oracle Q times, it is able to compute the kth power of any Q of the N elements. The assumption states that the probability that $\mathcal{A}$ computes the kth power of any $Q+1$ of the N elements (i.e. computes the kth power of "one more" element) is negligible.

Theorem 1. *Suppose the (N, Q)-one more DH assumption holds for $\langle g \rangle$, where Q is the number of functionality executions and $N = Q + q_1$ where q_1 is the number of $H_1(\cdot)$ queries. Then protocol 2Hash-DH UC-realizes the $\mathcal{F}_{\text{OPRF}}$ functionality.*

In other words, for any adversary $\mathcal{A}$ against the protocol in Section 2, there is a simulator SIM *that produces a view in the simulated world that no environment can distinguish with advantage better than $q_S \cdot \epsilon_{\mathsf{omdh},\langle g \rangle}(N, Q) + N^2/m$, where q_S is the number of senders, $\epsilon_{\mathsf{omdh},\langle g \rangle}(N, Q)$ is the probability of violating the (N, Q)-one more DH assumption and $m = |\langle g \rangle|$.*

Proof. For any environment $\mathcal{Z}$ and any adversary $\mathcal{A}$, we construct a simulator SIM as in Figure 3.2.

We now argue that the SIM above generates a view to $\mathcal{Z}$ which is indistinguishable from the view in the real world. Without loss of generality, suppose $\mathcal{A}$ is a "dummy" adversary who merely pass through all its computation to $\mathcal{Z}$.

Consider case 1 where $\mathcal{Z}$ is in the simulated world and is able to control U, S and $\mathcal{A}$ using the following interfaces:

- U: $\mathcal{Z}$ sends (Eval, *sid*, S, x) to U (and U transmits this message to $\mathcal{F}$); U outputs (Eval, ρ) to $\mathcal{Z}$.
- S: $\mathcal{Z}$ sends (KeyGen, *sid*) to S (and S transmits this message to $\mathcal{F}$); S sends (KeyGen, *sid*) and (SenderComplete, *sid*) to $\mathcal{Z}$.

1. Pick and record N random elements $r_1, ..., r_N \in \mathbb{Z}_m$, and set $g_1 := g^{r_1}, ..., g_N := g^{r_N}$. Set counter $j := 1$.
2. Upon receiving (KeyGen, *sid*, S, p) from F, record $\langle S, p \rangle$.
3. Every time when there is a fresh query to $H_1(\cdot)$, answer it with g_j and record (x, r_j, g_j). Set $j := j + 1$.
4. Upon receiving (Eval, *sid*, U) from $\mathcal{F}$, send g_j to $\mathcal{A}$ as U's message to some sender and record $\langle U, r_j, g_j \rangle$. Set $j := j + 1$.
5. Upon receiving a from $\mathcal{A}$ as a message from some user to some sender S,

- If there exists a pair $\langle S, k \rangle$ stored in this step, then send a^k to $\mathcal{A}$.
- Otherwise pick a random element k in $\mathbb{Z}_m$, record $\langle S, k \rangle$ and send a^k to $\mathcal{A}$.

In either case, send (SenderComplete, *sid*, S) to $\mathcal{F}$. 6. Upon receiving b from $\mathcal{A}$ as some sender's message to a user U, recover r_j and g_j corresponding to U (as stored in step 4) and do the following:

- If there exists a pair $\langle S, k \rangle$ stored in step 5 such that $b = g_j^k$, then recover p corresponding to S (as stored in step 2) send (UserComplete, *sid*, U, p) to $\mathcal{F}$.
- Otherwise if there exists $p \in \{1, ..., P\}$ such that y_p is defined and $b^{1/r_j} = y_p$, then send (UserComplete, *sid*, U, p) to $\mathcal{F}$.
- Otherwise send (KeyGen, *sid*) to $\mathcal{F}$, and after getting $\mathcal{F}$'s response (KeyGen, *sid*, p), send (UserComplete, *sid*, U, p) to $\mathcal{F}$. Set $y_p := b^{1/r_j}$.

7. Every time when there is a fresh query (x, u) to $H_2(\cdot, \cdot)$,

- If there exists a triple (x, r_j, g_j) stored in step 3 and either of the following holds:
  - there exists a pair $\langle S, k \rangle$ stored in step 5 such that $u = g_j^k$, or
  - there exists $p \in \{1, ..., P\}$ such that $u = y_p^{r_j}$,

  then send (Eval, *sid*, x) to $\mathcal{F}$ on behalf of a corrupted user U^* and immediately follows it with (UserComplete, *sid*, U^*, p), where $p = \mathsf{pointer}(S)$ when the first case holds. If $\mathcal{F}$ ignores this message then abort and output Fail. Otherwise after getting $\mathcal{F}$'s response (Eval, ρ), set $H_2(x) := \rho$ and record $(g_j, u = g_j^k)$.
- Otherwise set $H_2(x, u)$ to be a random string in $\{0, 1\}^l$.

Figure 3.2: The Simulator SIM for the 2HashDH Protocol.

- $\mathcal{A}$: $\mathcal{Z}$ sends a to $\mathcal{A}$ when $\mathcal{A}$ acts as a user (and $\mathcal{A}$ transmits this message to SIM who acts as a sender), and sends b to $\mathcal{A}$ when $\mathcal{A}$ acts as a sender (and $\mathcal{A}$ transmits this message to SIM who acts as a user). $\mathcal{A}$ sends whatever it receives from SIM to $\mathcal{Z}$.

The view of $\mathcal{Z}$ in this case is as follows:

- (KeyGen, sid) from S,
- (SenderComplete, sid) from S,
- (Eval, ρ) from U,
- g_j from $\mathcal{A}$ when $\mathcal{A}$ acts as a sender,
- a^k from $\mathcal{A}$ when $\mathcal{A}$ acts as a user, and
- responses to $H_1(\cdot)$ and $H_2(\cdot,\cdot)$ queries.

Now consider case 2 where $\mathcal{Z}$ is in the real world and is able to control U, S and $\mathcal{A}$ using the same interfaces as in the case 1. The view of $\mathcal{Z}$ in this case is as follows:

- (KeyGen, sid) from S,
- (SenderComplete, sid) from S,
- (Eval, $f = H_2(x, H_1(x)^k)$) from U,
- $H_1(x)^r$ from $\mathcal{A}$ when $\mathcal{A}$ acts as a sender,
- a^k from $\mathcal{A}$ when $\mathcal{A}$ acts as a user, and
- responses to $H_1(\cdot)$ and $H_2(\cdot,\cdot)$ queries.

We now compare the two views generated in the two cases:

- (KeyGen, *sid*): In both cases, S gets (KeyGen, *sid*) after it generates its key. So there is no difference.

- (SenderComplete, *sid*): In case 1, $\mathcal{A}$ gets (SenderComplete, *sid*) in step 5: after SIM receives a from $\mathcal{A}$, it sends a^k to $\mathcal{A}$ and (SenderComplete, *sid*, S) to $\mathcal{F}$, and $\mathcal{F}$ sends (SenderComplete, *sid*) to S (who in turn transmits this message to $\mathcal{Z}$). In case 2, after S receives a from $\mathcal{A}$, it sends a^k to $\mathcal{A}$ and (SenderComplete, *sid*) to $\mathcal{Z}$. From $\mathcal{Z}$'s point of view, the two cases are exactly the same.

- (Eval, ρ) and (Eval, $f = H_2(x, H_1(x)^k)$): In case 1, U outputs (Eval, ρ) in step 6. The only way to distinguish between ρ and f is to query the H_2 oracle. However, once $(x, H_1(x)^k)$ is queried, SIM is able to detect this: If k is the key of an honest sender, then $u = g_j^k$ holds, which is the first condition of the first case in step 7. If k is the key of a sender generated by SIM, then $u = g_j^k = b = y_p^{r_j}$ holds, which is the second condition of the first case in step 7. After that, SIM can get the ρ value by interacting with $\mathcal{F}$, i.e. these two values will be the same (provided that Fail does not happen).

- g_j and $H_1(x)^r$: In case 1, $\mathcal{A}$ gets g_j in step 4, which is a random element in $\langle g \rangle$. In case 2, $\mathcal{A}$ gets $H_1(x)^r$ where r is a random value in $\mathbb{Z}_m$ chosen by U and not revealed to $\mathcal{A}$. Therefore, from $\mathcal{Z}$'s point of view, $H_1(x)^r$ is random, and cannot be distinguished from the random g_j.

- a^k: In case 1, $\mathcal{A}$ sends a to SIM and gets a^k from SIM, where k is a random value. In case 2, $\mathcal{A}$ sends a to S and gets a^k from S, where k is the sender's key, which is randomly chosen. The two cases are essentially the same.

- answers to $H_1(\cdot)$ and $H_2(\cdot, \cdot)$ queries: In case 1, the answers are random from $\mathcal{Z}$'s point of view. Therefore, $\mathcal{Z}$ cannot distinguish them from answers in case 2.

We conclude that if Fail does not happen, the $\mathcal{Z}$ is not able to distinguish between case 1 (the simulated world) and case 2 (the real world). Now we upper bound Pr[Fail]. Fail

happens in the first case of step 7, where SIM sends a (UserComplete, sid, U^*, p) message to $\mathcal{F}$ but $\mathcal{F}$ ignores that. This can only happen when the first condition holds and $\mathsf{tickets}(p) = 0$ (where $p = \mathsf{pointer}(S)$). For every $p' \in \{1, ..., P\}$, let Fail(p') be the particular event that the (UserComplete, sid, U^*, p') message is ignored. We can see that $\Pr[\text{Fail}] \leq \sum_{p'=1}^{P} \Pr[\text{Fail}(p')]$, and $P \leq q_S$, where q_S is the number of senders.

We now upper bound $\Pr[\text{Fail}(p')]$ by reducing Fail(p') to the one-more DH problem. Using the reduction $\mathcal{R}_{p'}$ in Figure 3.3, we can see the following two facts:

- Every time the DDH oracle is used (in step 5), a (SenderComplete, sid, S') message is sent to $\mathcal{F}$, so $\mathsf{tickets}(p')$ increases.
- Every time $\mathsf{tickets}(p')$ decreases (in the first case of step 6 or the first condition of the first case of step 7), a pair $(z, z^{k'})$ is recorded, where z is either $H_1(x)$ or the message sent to $\mathcal{A}$ in step 4; in either case, z is in $g_1, ..., g_N$.

Therefore, if Fail(p') happens, the number of pairs recorded is more than the number of DDH oracle queries by one (assuming that there is no collision in $g_1, ..., g_N$), so the one more DH-assumption is violated. That is, $\Pr[\text{Fail}(p')|\text{There is no collision in } g_1, ..., g_N] = \epsilon_{\mathsf{omdh},\langle g\rangle}(N, Q)$. So

$$\Pr[\text{Fail}|\text{There is no collision in } g_1, ..., g_N] \leq q_S \cdot \epsilon_{\mathsf{omdh},\langle g\rangle}(N, Q)$$

On the other hand, the probability that there is collision in $g_1, ..., g_N$ is upper bounded by N^2/m. Thus, we have

$$\Pr[\text{Fail}] \leq q_S \cdot \epsilon_{\mathsf{omdh},\langle g\rangle}(N, Q) + N^2/m.$$

That concludes our proof. □

Suppose $(Q, g, y = g^{k'}, g_1, ..., g_N)$ is an instance of the one-more DH problem. We run the simulator, with the following revisions:

- In step 1, only set $j := 1$ and delete all other processes.
- In step 2, if $p = p'$, then record the corresponding S as S'.
- In step 3 and step 4, use the challenges $g_1, ..., g_N$ instead of random elements in $\langle g \rangle$ to answer $H_1(\cdot)$ queries (in step 3) and prepare the messages sent to $\mathcal{A}$ (in step 4). Besides, since there is no r_j, record (x, g_j) instead of (x, r_j, g_j) (in step 3), and $\langle U, g_j \rangle$ instead of $\langle U, r_j, g_j \rangle$ (in step 4).
- In step 5, if a' acts as a message to S', then use the DDH oracle to compute $b' := a'^{k'}$. Record (a', b') instead of $\langle S, k \rangle$.
- In step 6, do the following instead: Upon receiving b from $\mathcal{A}$ as some sender's message to a user U, recover a corresponding g_j (as stored in step 4), record (g_j, b), and do as follows:
  - If $\mathrm{DDH}(g_j, b, a', b')$ then send (UserComplete, *sid*, U, p') to $\mathcal{F}$.
  - Otherwise if there exists a pair $\langle S, k \rangle$ stored in step 5 such that $b = a^k$, then send (UserComplete, *sid*, U, p) to F, where $p = \mathsf{pointer}(S)$.
  - Otherwise if there exists $p \in \{1, ..., P\}$ such that a_p and b_p are defined and $\mathrm{DDH}(g_j, b, a_p, b_p)$, then send (UserComplete, *sid*, U, p) to $\mathcal{F}$.
  - Otherwise send (KeyGen, *sid*) to $\mathcal{F}$, and after getting $\mathcal{F}$'s response (KeyGen, *sid*, p), send (UserComplete, *sid*, U, P) to $\mathcal{F}$. Set $a_p := g_j, b_p := b$.
- In the first case of step 7, use these three conditions instead:
  - $\mathrm{DDH}(g_j, u, a', b')$,
  - there exists a pair $\langle S, k \rangle$ stored in step 5 such that $u = g_j^k$, or
  - there exists $p \in \{1, ..., P\}$ such that $\mathrm{DDH}(g_j, u, a_p, b_p)$.

Figure 3.3: The Reduction $\mathcal{R}_{p'}$ to the One-More DH Problem.

Remark. Although in this work $\mathcal{F}_{\text{OPRF}}$ is used to construct a PPSS scheme, we mention here that OPRF is of interest in many other areas, such as keyword search [5] and secure function evaluation of set intersection [6, 8]. Therefore, our protocol 2HashDH may have further applications.

Chapter 4

Password-Protected Secret Sharing Based on OPRF

4.1 Definition

Password-Protected Secret Sharing (PPSS) is originally introduced in [1], where such a scheme is used to protect an encrypted secret message. [7] defines PPSS in terms of protecting a random key which in turn protects a secret message in an encryption scheme, rather than protecting the message itself. We follow the definition in [7], partially because it is more flexible: a secret key can be used not only in encryption protocols, but also in signatures, key exchange protocols, and so on.

Definition 1. *A PPSS scheme is a tuple* $(\mathsf{ParGen}, \mathsf{SKeyGen}, \mathsf{Init}, \mathsf{Rec})$ *involving* $n+1$ *parties, a user denoted by* U*, and* n *servers denoted by* $S_1, ..., S_n$*:*

- ParGen*: Takes* 1^k *where* k *is the security parameter as input, and outputs a public string* CRS*.*

- SKeyGen*: Each S_i $(i = 1, ..., n)$ takes* CRS *as input, and outputs its private state σ_i.*
- Init*: A protocol executed by U and $S_1, ..., S_n$.*
  - *U runs algorithm* $\mathsf{U}_{\mathsf{Init}}$*, which takes* CRS *and a password* pw *in a dictionary D as input, and outputs a private key $K \in \{0,1\}^k$.*
  - *Each S_i $(i = 1, ..., n)$ runs algorithm* $\mathsf{S}_{\mathsf{Init}}$*, which takes* CRS *and σ_i as input, and outputs a private user-specific information ω_i.*
- Rec*: A protocol executed by U and $S_1, ..., S_n$.*
  - *U runs algorithm* $\mathsf{U}_{\mathsf{Rec}}$*, which takes* CRS *and* pw *as input, and outputs K' which is a k-bit string or the rejection symbol $\perp$.*
  - *Each S_i $(i = 1, ..., n)$ runs algorithm* $\mathsf{S}_{\mathsf{Rec}}$*, which takes* CRS*, σ_i and ω_i as input. There is no output for S_i.*

4.2 Properties

PPSS is designed as a scheme with a "threshold" parameter t, where $0 \leq t \leq n - 1$. It should have the following properties:

- U is able to (successfully) reconstruct its key K assuming he can interact with any $t + 1$ servers among $S_1, ..., S_n$, and
- U's key K remains pseudorandom to any polynomial-time adversary $\mathcal{A}$ who breaks into any t servers among $S_1, ..., S_n$.

Now we analyze the properties of PPSS:

Correctness. This is the most basic property, which means that U reconstructs the same key as generated in the initialization process; that is, for any k, any $\mathsf{CRS} \leftarrow \mathsf{ParGen}(1^k)$, any $(\sigma_1, ..., \sigma_n) \leftarrow \mathsf{SKeyGen}(\mathsf{CRS})$, any $\mathsf{pw} \in D$, any $K \leftarrow \mathsf{U}_{\mathsf{Init}}(\mathsf{CRS}, \mathsf{pw})$, and any $K' \leftarrow \mathsf{U}_{\mathsf{Rec}}(\mathsf{CRS}, \mathsf{pw})$, we have $K' = K$.

Security. Now suppose there is an adversary $\mathcal{A}$ who acts as the "man-in-the-middle" between U and $S_1, ..., S_n$. Assume that t out of the n servers are corrupt. In addition to getting all public values, $\mathcal{A}$ can also see the private states and information σ_i and ω_i for corrupt S_i's $(i = 1, ..., n)$.

Intuitively, the security of PPSS means that the key generated by U, K, should be pseudorandom except for some specific probability (see below for more details on this); that is, $\mathcal{A}$ is not able to distinguish K from a random string in $\{0,1\}^k$. More formally, suppose $\mathcal{A}$ runs S_i's protocol and interacts with U. At the beginning of Rec, a random bit $b \in \{0,1\}$ is chosen. If $b = 1$, then U follows the code of $\mathsf{U}_{\mathsf{Rec}}$ and outputs $K^{(1)}$, which is either a k-bit string or $\perp$. If $b = 0$, then U outputs $\perp$ if $\mathsf{U}_{\mathsf{Rec}}$ outputs $\perp$, and a random string $K^{(0)} \in \{0,1\}^k$ otherwise. (To simplify, we define such code of U to be $\mathsf{U}^{\diamond}_{\mathsf{Rec}}(\mathsf{CRS}, \mathsf{pw}, b, K^{(b)})$.) Furthermore, $\mathcal{A}$ is able run U's protocol and interact with S_i's. The probability that $\mathcal{A}$ wins the game, i.e. the guesses b correctly, should be $1/2$ plus some specific value.

Suppose $\mathcal{A}$ interacts with $\mathsf{U}_{\mathsf{Rec}}$ and $\mathsf{S}_{\mathsf{Rec}}$ q_U and q_S times, respectively. Note that any PPSS scheme is inherently vulnerable to the following two types of attacks:

(1) $\mathcal{A}$ may guess a $\mathsf{pw}' \in D$ and run S_i's protocol interacting with U. If U does not reject, then $\mathcal{A}$ concludes that $\mathsf{pw}' = \mathsf{pw}$. (And obviously, once pw is leaked, $\mathcal{A}$ is able to run $\mathsf{U}_{\mathsf{Init}}$ on pw and learn K.) Since $\mathcal{A}$ is able to attack q_U times, and each time its probability of success (i.e. guessing pw correctly) is $1/|D|$, the total probability that $\mathcal{A}$ successfully attacks the scheme using this method is $q_U/|D|$.

(2) $\mathcal{A}$ may guess a $\mathsf{pw}' \in D$ and run U's protocol interacting with the t corrupt servers and any one of the uncorrupt servers. Since $\mathcal{A}$ interacts with $t+1$ servers in total, he is able to run $\mathsf{U}_{\mathsf{Rec}}$ locally; if the output is not $\perp$, $\mathcal{A}$ can conclude that $\mathsf{pw}' = \mathsf{pw}$. Since $\mathcal{A}$ is able to attack q_S times, and each time its probability of success (i.e. guessing pw correctly) is $1/|D|$, the total probability that $\mathcal{A}$ successfully attacks the scheme using this method is $q_S/|D|$.

In sum, the inherent insecurity (i.e. the probability that $\mathcal{A}$ wins the security game) of any PPSS scheme is

$$(q_U + q_S)/|D|,$$

and if a PPSS scheme is secure, its insecurity should be only negligibly larger than that value.

We are now ready to present the formal definition of the security of PPSS:

Definition 2. *A PPSS scheme with parameters (t, n) is (q_U, q_S, ϵ)-secure if for any polynomial-time algorithm $\mathcal{A}$ and any $B \in \{1, ..., n\}$ s.t. $|B| = t$, we have*

$$\mathbf{Adv}_{\mathcal{A}}^{\mathbf{ppss}} \leq 1/2[(q_U + q_S)/|D| + \epsilon],$$

where $\mathbf{Adv}_{\mathcal{A}}^{\mathbf{ppss}}$ is the probability that $\mathcal{A}$ wins the following game:

(1) Set $b \xleftarrow{\$} \{0,1\}$, $\mathsf{pw} \xleftarrow{\$} D$, $\mathsf{CRS} \leftarrow \mathsf{ParGen}(1^k)$ and $\sigma_i \leftarrow \mathsf{SKeyGen}(\mathsf{CRS})$ for $i \notin B$. Send CRS to $\mathcal{A}$.

(2) Run $\mathsf{U}_{\mathsf{Init}}(\mathsf{CRS}, \mathsf{pw})$ and $\mathsf{S}_{\mathsf{Init}}(\mathsf{CRS}, \sigma_i)$ for $i \notin B$. $\mathcal{A}$ is able to control S_i for $i \in B$, while the channels between U and $S_1, ..., S_n$ here are authenticated. Let $K^{(0)} \xleftarrow{\$} \{0,1\}^k$ and $K^{(1)} \leftarrow \mathsf{U}_{\mathsf{Init}}(\mathsf{CRS}, \mathsf{pw})$. Send $K^{(b)}$ to $\mathcal{A}$.

(3) Let $\mathcal{A}$ have oracle access to $\mathsf{U}^{\diamond}_{\mathsf{Rec}}(\mathsf{CRS}, \mathsf{pw}, b, K^{(b)})$ and $\mathsf{S}_{\mathsf{Rec}}$. $\mathcal{A}$ is allowed to query $\mathsf{U}^{\diamond}_{\mathsf{Rec}}(\mathsf{CRS}, \mathsf{pw}, b, K^{(b)})$ and $\mathsf{S}_{\mathsf{Rec}}$ q_U and q_S times, respectively.

(4) $\mathcal{A}$ outputs $b' \in \{0,1\}$. $\mathcal{A}$ wins if $b' = b$.

Soundness. We briefly discuss another desired property of a PPSS scheme called soundness here. Basically it means that no polynomial-time adversary is able to make U accept the wrong key; that is, U outputs K in Init and K' in Rec, while $K' \neq \perp$ and $K \neq K'$. We do not present the formal definition.

In fact soundness is implied by security. Suppose there is an adversary $\mathcal{A}$ who successfully attack the scheme's soundness. Then let $\mathcal{A}'$ run $\mathcal{A}$, and output 0 if U's output K in Init and K' in Rec are the same, and output 1 if $K \neq K'$. Clearly $\mathcal{A}'$ successfully attacks the security of the scheme.

Chapter 5

A PPSS Scheme Based on OPRF

In this chapter we present a secure PPSS scheme. It uses the functionality $\mathcal{F}_{\text{OPRF}}$ as part of the construction, and works on any realization of $\mathcal{F}_{\text{OPRF}}$ (in particular 2HashDH). Our scheme works in the general model; however, since we only have realization of $\mathcal{F}_{\text{OPRF}}$ in ROM, it is still unknown whether PPSS can be realized in the general model.

We show our PPSS scheme in Figure 5.1.

For the sake of completeness, we recall the concepts of commitment binding, hiding and non-malleability here:

- binding (with bound ϵ_B): for any polynomial-time algorithm $\mathcal{A}$,

$$\Pr[(C, m, m', r, r') \leftarrow \mathcal{A} \text{ s.t. } m \neq m' \wedge C = \mathsf{COM}(m; r) = \mathsf{COM}(m'; r')] \leq \epsilon_B.$$

- hiding (with bound ϵ_H): for any polynomial-time algorithm $\mathcal{A}$ and any two messages m_0 and m_1,

$$\Pr[b \leftarrow \mathcal{A}(C) | C = \mathsf{COM}(m_b; r), r \xleftarrow{\$}] \leq 1/2(1 + \epsilon_H).$$

$\mathsf{ParGen}(k)$ Set CRS to be an instance of a commitment COM with security parameter k, which has the properties of binding, hiding and non-malleability with respect to decommitment.
$\mathsf{SKeyGen}(\mathsf{CRS})$ For each S_i $(i = 1, ..., n)$, set σ_i to be the unique PRF key which S_i holds to the realization of $\mathcal{F}_{\text{OPRF}}$ (henceforth $\mathcal{F}$).
Init 1. Let binary extension field $\mathbb{F} := GF(2^l)$. U sets $s \xleftarrow{\$} \mathbb{F}$, and parses s as $[r||K]$.
2. U generates $(s_1, ..., s_n)$ as a (t, n) Shamir's secret-sharing of s over $\mathbb{F}$.
3. U sends (EVAL, sid, S_i, pw) $(i = 1, ..., n)$ to $\mathcal{F}$, where $sid = (S_1, ..., S_n)$. $\mathcal{F}$ sends (EVAL, ρ_i) to U as response.
4. U computes $e_i := s_i \oplus \rho_i$ $(i = 1, ..., n)$. Set $\boldsymbol{e} := (e_1, ..., e_n)$.
5. U computes $C := \mathsf{COM}((\mathsf{pw}, \boldsymbol{e}); r)$.
6. U sends $\omega_i := (\boldsymbol{e}, C)$ to S_i $(i = 1, ..., n)$. Note that all ω_i's are the same, so we simply denote it as ω.
Rec 1. Each S_i $(i = 1, ..., n)$ sends $\omega_i' := \omega$ to U. U finds a set $P \subset \{1, ..., n\}$ such that $|P| = t + 1$ and ω_p' for $p \in P$ are the same (let ω' be such ω_p's). If there is no such set, U aborts. Otherwise let $\omega' = (\boldsymbol{e'}, C') = ((e_1', ..., e_n'), C')$.
2. U sends (EVAL, sid, S_p, pw) $(p \in P)$ to $\mathcal{F}$. $\mathcal{F}$ sends (EVAL, ρ_p') to U.
3. U computes $s_p' := e_p' \oplus \rho_p'$.
4. U recovers s' from s_p' $(p \in P)$ using (t, n) Shamir's secret-sharing and parses it as $[r'||K']$.
5. U checks whether $C' = \mathsf{COM}((\mathsf{pw}, \boldsymbol{e'}); r')$. If so, then U outputs K'. Otherwise U rejects.

Figure 5.1: PPSS Scheme Based on $\mathcal{F}_{\text{OPRF}}$.

- non-malleability (with bound ϵ_{NM}): for any polynomial-time algorithms $\mathcal{A}_0$, $\mathcal{A}_1$, any relation R and any distribution D on the message space M, define two games:
  - $\mathbf{G_0}$: $m \xleftarrow{D} M$; $m' \leftarrow \mathcal{A}_0$.
  - $\mathbf{G_1}$: $m \xleftarrow{D} M$; $r \xleftarrow{\$}$; $C := \mathsf{COM}(m; r)$; $C' \leftarrow \mathcal{A}_1(C)$ s.t. $C' \neq C$; $(m', r') \leftarrow \mathcal{A}_1(m, r)$ s.t. $C' = \mathsf{COM}(m'; r')$.

  Let $p_0 = \Pr[R(m, m')||\mathbf{G_0}]$ and $p_1 = \Pr[R(m, m')||\mathbf{G_1}]$. Then $|p_0 - p_1| \leq \epsilon_{NM}$.

It is easy to see the correctness of this scheme: in Rec, P is any $(t + 1)$-element subset of $\{1, ..., n\}$, $\omega' = \omega$ [i.e. $e_i' = e_i$ $(i = 1, ..., n)$ and $C' = C$], $\rho_p' = \rho_p$ $(p \in P)$, $s_p' = s_p$, $r' = r$, $K' = K$ and U's check in step 5 will pass. Then U's output $K' = K$, which satisfies

the definition of correctness. Now we prove the security of our scheme. Before proving the theorem, we introduce a property of commitments:

Lemma 1. *Supporse* COM *is a* ϵ_B*-binding and* ϵ_H*-hiding commitment. Then for any polynomial-time algorithm* $\mathcal{A}$ *and any message* m *in the message space, we have*

$$\Pr_{r\xleftarrow{\$},C:=\mathsf{COM}(m;r),r'\leftarrow\mathcal{A}(m,C)}[C=\mathsf{COM}(m;r')]\leq\epsilon_B+\epsilon_H.$$

Proof. Suppose there is a polynomial-time algorithm $\mathcal{A}$ such that $C=\mathsf{COM}(m;r')$ where $r'\leftarrow\mathcal{A}(m,C)$. We construct a reduction to COM's hiding property in Figure 5.2:

1. $\mathcal{R}$ sets $m_0:=m$ and m_1 to be any message different from m. $\mathcal{R}$ outputs m_0 and m_1.
2. $\mathcal{R}$ receives $C=\mathsf{COM}(m_b;r)$ where $r\xleftarrow{\$}$ and $b\xleftarrow{\$}\{0,1\}$.
3. $\mathcal{R}$ sends m and C to $\mathcal{A}$.
4. $\mathcal{A}$ sends r' to $\mathcal{R}$.
5. $\mathcal{R}$ outputs 0 if $C=\mathsf{COM}(m_0;r')$, and 1 otherwise.

Figure 5.2: Reduction to COM's Hiding Property.

Let F be the event that $\mathcal{A}$ wins his game but $\mathcal{R}$ loses his game. Then we have

$$\mathbf{Adv}_{\mathcal{A}}\leq\mathbf{Adv}_{\mathcal{R}}+\Pr[F].$$

But according to COM's hiding property, we know that $\mathbf{Adv}_{\mathcal{R}}\leq\epsilon_H$. Thus,

$$\mathbf{Adv}_{\mathcal{A}}\leq\Pr[F]+\epsilon_H.$$

Now we upper bound $\Pr[F]$. F may happen in two cases:

- $\mathcal{R}$ outputs 1, but $b=0$: this means

$$C\neq\mathsf{COM}(m_0;r')\text{ (since }\mathcal{R}\text{ outputs 1)},\ C=\mathsf{COM}(m_0;r)\text{ (since }b=0),$$

this contradicts the fact that $\mathcal{A}$ wins. Therefore, this case can never happen.

- $\mathcal{R}$ outputs 0, but $b = 1$: this means

$$C = \mathsf{COM}(m_0; r') \text{ (since } \mathcal{R} \text{ outputs 0)}, C = \mathsf{COM}(m_1; r) \text{ (since } b = 0),$$

so

$$\mathsf{COM}(m_0; r') = \mathsf{COM}(m_1; r).$$

But the probability that this happens is upper-bounded by ϵ_B: there is another reduction to COM's binding property using $\mathcal{A}$. We omit the details.

So we have $\Pr[F] \leq \epsilon_B + \epsilon_H$. This concludes our proof. □

Theorem 2. *Suppose* COM *is* ϵ_B*-binding,* ϵ_H*-hiding and* $\epsilon_N M$*-non-malleable. Then the scheme above is* (q_U, q_S, ϵ)*-secure for* $\epsilon = (q_U + 1)\epsilon_H + (q_U + 1)\epsilon_B + q_U\epsilon_{NM}$.

Proof. We first explicitly point out what the security experiment is. Among all n servers, t of them are corrupt; without loss of generality, we suppose they are $S_1, ..., S_t$, and $S_{t+1}, ..., S_n$ are honest. Of the four phases of the execution of the scheme, ParGen and $\mathsf{SKeyGen}$ are trivial, so we omit it here. As for Init and Rec, note that the adversary $\mathcal{A}$ is not able to interfere the Init. process; in Rec, $\mathcal{A}$ is able to interact with two oracles, $\mathsf{U}_{\mathsf{Rec}}$ and $\mathsf{S}_{\mathsf{Rec}}$, which run the code of U and $S_1, ..., S_n$ in Rec, respectively, and,

- While interacting with $\mathsf{U}_{\mathsf{Rec}}$, $\mathcal{A}$ has the following ability:
  - $\mathcal{A}$ can send $\omega'_1, ..., \omega'_n$ to U in step 1. Note that U continues execution only if there exists a $(t+1)$-element set $P \subset \{1, ..., n\}$ such that ω'_p for $p \in P$ are the same; otherwise U aborts. This is equivalent to the scenerio where $\mathcal{A}$ explicitly specifies a $(t+1)$-element set $P \subset \{1, ..., n\}$ and sends an $\omega' = \omega'_p$ for $p \in P$ to U. Suppose $\omega' = (\boldsymbol{e'}, C') = ((e'_1, ..., e'_n), C')$.

– $\mathcal{A}$ can send (UserComplete, sid, U, p') for $p' \in P'$ to $\mathcal{F}$ in step 2, where P' is a $(t+1)$-element set of pointers. That is, $\mathcal{A}$ makes $\rho'_p = F_{p'}(\mathsf{pw})$. Since UserComplete, $sid = (S_1, ..., S_n)$ and U remains the same during the whole session, we simply use P' as $\mathcal{A}$'s input here.

In sum, the output of $\mathsf{U}_{\mathsf{Rec}}$ is decided by $\mathcal{A}$'s inputs ω', P and P'.

- While interacting with $\mathsf{S}_{\mathsf{Rec}}$, $\mathcal{A}$ has the following ability: it can send (Eval, S_i, pw') for any $i \in \{1, ..., n\}$ and any $\mathsf{pw}' \in D$ to $\mathcal{F}$ in step 2 (that is, $\mathcal{A}$ computes $F_i(\mathsf{pw}')$). (Note that we do not have to send ω to $\mathcal{A}$ as in step 1, because $\mathcal{A}$ already receives ω in step 5 of Init.) In sum, the output of $\mathsf{S}_{\mathsf{Rec}}$ is decided by $\mathcal{A}$'s input i and pw'.

We now show the whole security experiment. Since interacting with $\mathsf{S}_{\mathsf{Rec}}$ is simply computing the F_i value on some string, we omit this and only show the Init process and the interaction with $\mathsf{U}_{\mathsf{Rec}}$. In addition, since the ParGen and $\mathsf{SKeyGen}$ processes are trivial, we omit them and only show Init and Rec. Furthermore, recall that at the beginning of the game, a random bit $b \xleftarrow{\$} \{0,1\}$ is generated; if $b = 0$, the output of $\mathsf{U}^{\diamond}_{\mathsf{Rec}}$ is a random string (provided that $\mathsf{U}_{\mathsf{Rec}}$ does not output $\perp$), while if $b = 1$, the output of $\mathsf{U}^{\diamond}_{\mathsf{Rec}}$ is the output of $\mathsf{U}_{\mathsf{Rec}}$. We only describe the case where $b = 1$ below, and for the case where $b = 0$, the security game is easy to understand. Basically, $\mathcal{A}$ wins the game if he can distinguish the output K' below with a random string.

We define a series of security games below. In each of these games, we define three events that may lead to the adversary's success:

- E_S: On some interaction with $\mathsf{S}_{\mathsf{Rec}}$, $\mathcal{A}$'s input i is the pointer of some honest server, and $\mathsf{pw}' = \mathsf{pw}$. (That is, $\mathcal{A}$ computes $F_i(\mathsf{pw})$.)

- E_U: On some interaction with $\mathsf{U}_{\mathsf{Rec}}$,

$$\omega' \neq \omega \wedge C' = \mathsf{COM}((\mathsf{pw}, \boldsymbol{e'}); r'),$$

Init 1. Pick $\mathsf{pw} \xleftarrow{\$} D$.
2. Set $s \xleftarrow{\$} \mathbb{F}$, and parse s as $[r||K]$.
3. Generate $(s_1, ..., s_n)$ as a (t, n) Shamir's secret-sharing of s over $\mathbb{F}$.
4. Send (EVAL, sid, S_i, pw) $(i = 1, ..., n)$ to $\mathcal{F}$, where $sid = (S_1, ..., S_n)$. Receive (EVAL, ρ_i) from U as response.
5. Compute $e_i := s_i \oplus \rho_i$ $(i = 1, ..., n)$. Set $\boldsymbol{e} := (e_1, ..., e_n)$.
6. Compute $C := \mathsf{COM}((\mathsf{pw}, \boldsymbol{e}); r)$.
7. Send $\omega := (\boldsymbol{e}, C)$ to $\mathcal{A}$.
Rec (repeated q_U times)
1. $\mathcal{A}$ inputs $\omega' = (\boldsymbol{e'}, C')$ and $P \subset \{1, ..., n\}$, and sends P' to $\mathcal{F}$.
2. For each $p \in P$, compute $\rho'_p := F_{p'}(\mathsf{pw})$ where p' is a pointer in P' corresponding to p.
3. Compute $s'_p := e'_p \oplus \rho'_p$.
4. Recover s' from s'_p $(p \in P)$ using (t, n) Shamir's secret-sharing and parses it as $[r'||K']$.
5. If $C' = \mathsf{COM}((\mathsf{pw}, \boldsymbol{e'}); r')$, then output K'. Otherwise reject (i.e. output $\perp$).

Figure 5.3: The Security Game $\mathbf{G_0}$.

where $r' = s'[L]$, s' is recovered from $(e'_p \oplus \rho'_p)$ $(p \in P)$ using (t, n) Shamir's secret-sharing, $\rho'_p = F_{p'}(\mathsf{pw})$. (In the security game, this means that $\mathcal{A}$ sends something other than ω in step 1 of Rec., but U's check passes.)

- E_F: On some interaction with $\mathsf{U}_{\mathsf{Rec}}$,

$$\omega' = \omega \wedge P' \neq P \wedge C = \mathsf{COM}((\mathsf{pw}, \boldsymbol{e}); r'),$$

where r' is the same as above. (In the security experiment, this means that (1) $\mathcal{A}$ does not interfere step 1 of Rec, and (2) $\mathcal{A}$ uses some other servers instead of servers whose pointers are in P in step 2 of Rec, but U's check passes.)

Let $E = E_S \vee E_U \vee E_F$.

Let $\mathbf{G_0}$ be the security experiment. Let $\mathbf{G_1}$ be a modification of $\mathbf{G_0}$, which outputs K in $mathsfInit$ if $\omega' = \omega$ and $P' = P$, and rejects otherwise.

Note that

Init 1. Pick $\mathsf{pw} \xleftarrow{\$} D$.
2. Set $s \xleftarrow{\$} \mathbb{F}$, and parse s as $[r||K]$.
3. Generate $(s_1, ..., s_n)$ as a (t, n) Shamir's secret-sharing of s over $\mathbb{F}$.
4. Send (EVAL, sid, S_i, pw) $(i = 1, ..., n)$ to $\mathcal{F}$, where $sid = (S_1, ..., S_n)$. Receive (EVAL, ρ_i) from U as response.
5. Compute $e_i := s_i \oplus \rho_i$ $(i = 1, ..., n)$. Set $\boldsymbol{e} := (e_1, ..., e_n)$.
6. Compute $C := \mathsf{COM}((\mathsf{pw}, \boldsymbol{e}); r)$.
7. Send $\omega := (\boldsymbol{e}, C)$ to $\mathcal{A}$.
Rec (repeated q_U times)
1. $\mathcal{A}$ inputs $\omega' = (\boldsymbol{e'}, C')$ and $P \subset \{1, ..., n\}$, and sends P' to $\mathcal{F}$.
2. If $\omega' = \omega$ and $P' = P$, then output K. Otherwise reject.

Figure 5.4: The Game $\mathbf{G_1}$.

$$\neg E_U \wedge \neg E_F = (\omega' = \omega \wedge P' = P) \vee C' \neq \mathsf{COM}((\mathsf{pw}, \boldsymbol{e'}); r'),$$

and

- If $\omega' = \omega \wedge P' = P$, then both $\mathbf{G_0}$ and $\mathbf{G_1}$ output K;
- If $\neg(\omega' = \omega \wedge P' = P) \wedge C' \neq \mathsf{COM}((\mathsf{pw}, \boldsymbol{e'}); r')$, then both $\mathbf{G_0}$ and $\mathbf{G_1}$ reject.

Therefore, $\mathbf{G_1}$ is identical to $\mathbf{G_0}$ assuming neither E_U nor E_F happens.

Let $\mathbf{G_2}$ be a modification of $\mathbf{G_1}$, where $(s_1, ..., s_n)$ is the (t, n) Shamir's secret-sharing of 0 instead of s.

Assume E_S does not happen. Then $\mathcal{A}$ is able to learn information about at most t shares among $s_1, ..., s_n$, since $\mathcal{A}$ does not compute ρ_i for any corrupt server's pointer i. Therefore, s is random from $\mathcal{A}$'s view, and can be replaced by any element in $\mathbb{F}$, e.g. 0.

Let $\mathbf{G_3}$ be a modification of $\mathbf{G_2}$, where e_i $(i = 1, ..., n)$ are random strings in $\{0, 1\}^l$ instead of $s_i \oplus \rho_i$. Since ρ_i does not appear now, we can simply skip generating it.

Again, if E_S does not happen, $\mathbf{G_3}$ is identical to $\mathbf{G_2}$ from $\mathcal{A}$'s view. That is because $\mathcal{A}$ is

Init 1. Pick $\mathsf{pw} \xleftarrow{\$} D$.
2. Set $s \xleftarrow{\$} \mathbb{F}$, and parse s as $[r||K]$.
3. Generate $(s_1, ..., s_n)$ as a (t, n) Shamir's secret-sharing of 0 over $\mathbb{F}$.
4. Send (EVAL, sid, S_i, pw) $(i = 1, ..., n)$ to $\mathcal{F}$, where $sid = (S_1, ..., S_n)$. Receive (EVAL, ρ_i) from U as response.
5. Compute $e_i := s_i \oplus \rho_i$ $(i = 1, ..., n)$. Set $\boldsymbol{e} := (e_1, ..., e_n)$.
6. Compute $C := \mathsf{COM}((\mathsf{pw}, \boldsymbol{e}); r)$.
7. Send $\omega := (\boldsymbol{e}, C)$ to $\mathcal{A}$.
Rec (repeated q_U times)
1. $\mathcal{A}$ inputs $\omega' = (\boldsymbol{e'}, C')$ and $P \subset \{1, ..., n\}$, and sends P' to $\mathcal{F}$.
2. If $\omega' = \omega$ and $P' = P$, then output K. Otherwise reject.

Figure 5.5: The Game $\mathbf{G_2}$.

Init 1. Pick $\mathsf{pw} \xleftarrow{\$} D$.
2. Set $s \xleftarrow{\$} \mathbb{F}$, and parse s as $[r||K]$.
3.Generate $(s_1, ..., s_n)$ as a (t, n) Shamir's secret-sharing of 0 over $\mathbb{F}$.
4. Set $e_i \xleftarrow{\$} \{0, 1\}^l$ $(i = 1, ..., n)$. Set $\boldsymbol{e} := (e_1, ..., e_n)$.
5. Compute $C := \mathsf{COM}((\mathsf{pw}, \boldsymbol{e}); r)$.
6. Send $\omega := (\boldsymbol{e}, C)$ to $\mathcal{A}$.
Rec (repeated q_U times)
1. $\mathcal{A}$ inputs $\omega' = (\boldsymbol{e'}, C')$ and $P \subset \{1, ..., n\}$, and sends P' to $\mathcal{F}$.
2. If $\omega' = \omega$ and $P' = P$, then output K. Otherwise reject.

Figure 5.6: The Game $\mathbf{G_3}$.

able to compute at most t values among $\rho_1, ..., \rho_n$, and for any t shares among $s_1, ..., s_n$, they appears to be random to $\mathcal{A}$; therefore, all e_i's are random from $\mathcal{A}$'s view.

Note that $\mathbf{G_3}$'s output K is independently random from everything else. Therefore, we have

$$\mathbf{Adv}_{\mathcal{A}}^{\mathbf{G_3}} = 1/2.$$

Furthermore, $\mathbf{G_0}$ and $\mathbf{G_3}$ are the same if E does not happen, so

$$\Pr[E||\mathbf{G_0}] = \Pr[E||\mathbf{G_3}],$$

$$\mathbf{Adv}^{\mathbf{G_0}}_{\mathcal{A}}|\neg E = \mathbf{Adv}^{\mathbf{G_3}}_{\mathcal{A}}|\neg E.$$

Now we upper bound $\Pr[(E_U \vee E_F) \wedge \neg E_S||\mathbf{G_3}]$. We know that

$$\Pr[(E_U \vee E_F) \wedge \neg E_S||\mathbf{G_3}] \leq \Pr[E_U \wedge \neg E_S||\mathbf{G_3}] + \Pr[E_F \wedge \neg E_S||\mathbf{G_3}].$$

We break E_U into two sub-events:

- $E_{U,C}$: $C' = C$, while $\boldsymbol{e'} \neq \boldsymbol{e}$. Then we have

  $$C = \mathsf{COM}((\mathsf{pw}, \boldsymbol{e}); r) = \mathsf{COM}((\mathsf{pw}, \boldsymbol{e'}); r').$$

  We argue

  $$\Pr[E_{U,C} \wedge \neg E_S||\mathbf{G_3}] \leq \epsilon_B$$

  by constructing a reduction to COM's binding property, as shown in Figure 5.7.

1. $\mathcal{R}$ sets $\mathsf{pw} \xleftarrow{\$} D$.
2. $\mathcal{R}$ sets $s \xleftarrow{\$} \mathbb{F}$, and parse s as $[r||K]$.
3. $\mathcal{R}$ sets $e_i \xleftarrow{\$} \{0,1\}^l$ $(i = 1, ..., n)$. Set $\boldsymbol{e} := (e_1, ..., e_n)$.
4. $\mathcal{R}$ computes $C := \mathsf{COM}((\mathsf{pw}, \boldsymbol{e}); r)$.
5. $\mathcal{R}$ sends $\omega := (\boldsymbol{e}, C)$ to $\mathcal{A}$.
6. $\mathcal{A}$ sends $\omega' = (\boldsymbol{e'}, C')$ and P to $\mathcal{R}$ and P' to $\mathcal{F}$.
7. $\mathcal{R}$ serves all $\mathsf{U_{Rec}}$ queries in $\mathbf{G_3}$, until $E_{U,C}$ happens. Then $\mathcal{R}$ sends (EVAL, *sid*, p, pw) $(p \in P)$ to $\mathcal{F}$. $\mathcal{F}$ sends (EVAL, ρ'_p) to $\mathcal{R}$, where $\rho'_p = F_{p'}(\mathsf{pw})$ $(p' \in P')$.
8. $\mathcal{R}$ recovers s' from $(e'_p \oplus \rho'_p)$ using (t, n) Shamir's secret-sharing and parses s' as $[r'||K']$. 9. $\mathcal{R}$ outputs $(C, (\mathsf{pw}, \boldsymbol{e}), (\mathsf{pw}, \boldsymbol{e'}), r, r')$.

Figure 5.7: The Reduction to COM's Biding Property.

- $E_{U,NC}$: $C' \neq C$ (we do not care whether $\boldsymbol{e'} = \boldsymbol{e}$). Then we have

  $$C = \mathsf{COM}((\mathsf{pw}, \boldsymbol{e}); r),$$

  $$C' = \mathsf{COM}((\mathsf{pw}, \boldsymbol{e'}); r').$$

We argue

$$\Pr[E_{U,NC} \wedge \neg E_S || \mathbf{G_3}] \leq q_U(1/|D| + \epsilon_{NM}).$$

Note that $E_{U,NC}$ may happen at $\mathcal{A}$'s jth query to $\mathsf{U_{Rec}}$ for any $j = 1, ..., q_U$. Let $E^j_{U,NC}$ be the event that $E_{U,NC}$ happens at $\mathcal{A}$'s jth query to $\mathsf{U_{Rec}}$. Obviously $\Pr[E_{U,NC} \wedge \neg E_S || \mathbf{G_3}] \leq \sum_{j=1}^{q_U} \Pr[E^i_{U,NC} \wedge \neg E_S || \mathbf{G_3}]$. We now argue

$$\Pr[E^j_{U,NC} \wedge \neg E_S || \mathbf{G_3}] \leq 1/|D| + \epsilon_{NM}$$

by constructing a reduction to COM's non-malleability property. To make the proof clear, we first recall the definition of non-malleability in our context. Let $\mathcal{R}$ be any efficient algorithm. Let the message space be $\{(\mathsf{pw}, \boldsymbol{e}) : \mathsf{pw} \in D\}$, the relation be $\{((\mathsf{pw}, \boldsymbol{e}), (\mathsf{pw}', bme')) : \mathsf{pw} = \mathsf{pw}'\}$, and the distribution be $(\mathsf{pw}, \boldsymbol{e})$ where pw is chosen at random from D and $\boldsymbol{e} = (e_1, ..., e_n)$ is a fixed vector of n strings generated by $\mathcal{R}$. Then non-malleability means that the probability that $\mathcal{R}$ wins the following game is no greater than $1/|D| + \epsilon_{NM}$.

NM. 1. $\mathcal{R}$ generates a vector of n strings $\boldsymbol{e} = (e_1, ..., e_n)$.
2. $\mathcal{R}$ receives $C = \mathsf{COM}((\mathsf{pw}, \boldsymbol{e}); r)$, where $\mathsf{pw} \xleftarrow{\$} D$ and $r \xleftarrow{\$} \{0,1\}^l$.
3. $\mathcal{R}$ outputs $C' \neq C$.
4. $\mathcal{R}$ receives $((\mathsf{pw}, \boldsymbol{e}), r)$.
5. $\mathcal{R}$ outputs $((\mathsf{pw}, \boldsymbol{e'}), r')$. If $C' = \mathsf{COM}((\mathsf{pw}, \boldsymbol{e'}); r')$, then $\mathcal{R}$ wins.

Figure 5.8: The Non-Malleability Game **NM**.

The reduction is shown in Figure 5.9.

Combining the two results above, we get

$$\Pr[E_U \wedge \neg E_S || \mathbf{G_3}] \leq \epsilon_B + q_U(1/|D| + \epsilon_{NM}).$$

1. $\mathcal{R}$ picks n random strings $e_1, ..., e_n$, and sets $\boldsymbol{e} := (e_1, ..., e_n)$. (NM. 1.)
2. $\mathcal{R}$ receives C. (NM. 2.)
3. $\mathcal{R}$ sends $\omega = (\boldsymbol{e}, C)$ to $\mathcal{A}$.
4. $\mathcal{A}$ sends $\omega' = (\boldsymbol{e'}, C')$ and P to $\mathcal{R}$, and P' to $\mathcal{F}$.
5. If $C' = C$, $\mathcal{R}$ aborts (note that if $E^j_{U,NC}$ happens, then $C' \neq C$, so aborting when $C' = C$ will not affect our reduction). Otherwise $\mathcal{R}$ outputs C'. (NM. 3.)
6. $\mathcal{R}$ receives $((\mathsf{pw}, \boldsymbol{e}), r)$. (NM. 4.)
7. $\mathcal{R}$ outputs $((\mathsf{pw}, \boldsymbol{e'}), r')$, where r' is the same as in the reduction to the binding property. (NM. 5.)

Figure 5.9: The Reduction to COM's Non-Malleable Property.

1. $\mathcal{R}$ receives $(\mathsf{pw}, \boldsymbol{e})$ and C.
2. $\mathcal{R}$ sends $\omega = (\boldsymbol{e}, C)$ to $\mathcal{A}$.
3. $\mathcal{A}$ sends $\omega' = (\boldsymbol{e'}, C')$ and P to $\mathcal{R}$, and P' to $\mathcal{F}$.
4. If $\omega' \neq \omega$, $\mathcal{R}$ aborts (note that if E^j_F happens, then $\omega' = \omega$, so aborting when $\omega' \neq \omega$ will not affect our reduction). Otherwise $\mathcal{R}$ outputs r', which is the same as in the reduction to the binding property.

Figure 5.10: The Reduction to COM's Property in Lemma 1.

Next we upper bound $\Pr[E_F \wedge \neg E_S || \mathbf{G_3}]$. Again, we break E_F into q_U events E^j_F ($j = 1, ..., q_U$), where E^j_F is the event that E_F happens at $\mathcal{A}$'s jth query to $\mathsf{U_{Rec}}$. If $E^j_F \wedge \neg E_S$ happens, we construct a reduction to COM's property in Lemma 1, as shown in Figure 5.10. Therefore,

$$\Pr[E^j_F \wedge \neg E_S || \mathbf{G_3}] \leq \epsilon_B + \epsilon_H,$$

so

$$\Pr[E_F \wedge \neg E_S || \mathbf{G_3}] \leq q_U(\epsilon_B + \epsilon_H).$$

In sum, we get

$$\Pr[(E_U \vee E_F) \wedge \neg E_S || \mathbf{G_3}] \leq \epsilon_B + \epsilon_{NM} + q_U(1/|D| + \epsilon_B + \epsilon_H).$$

Let $\mathbf{G_4}$ be a modification of $\mathbf{G_3}$, where pw in step 1 of Init is replaced by 0. Since pw does not appear in $\mathbf{G_4}$, we have

$$\Pr[E_S||\mathbf{G_4}] \leq q_S/|D|.$$

Furthermore, an easy reduction shows that

$$\Pr[E_S||\mathbf{G_3}] \leq \Pr[E_S||\mathbf{G_4}] + \epsilon_H.$$

Therefore,

$$\begin{aligned}
\Pr[E||\mathbf{G_3}] &= Pr[E_U \vee E_F \vee E_S||\mathbf{G_3}] \\
&\leq \Pr[(E_U \vee E_F) \wedge \neg E_S||\mathbf{G_3}] + \Pr[E_S||\mathbf{G_3}] \\
&= (q_U + q_S)/|D| + \epsilon.
\end{aligned}$$

Combining all the results we get above, we conclude

$$\begin{aligned}
\mathbf{Adv}_{\mathcal{A}}^{\mathbf{G_0}} &\leq \Pr[E||\mathbf{G_0}] + \Pr[\mathcal{A}\text{wins} \wedge \neg E||\mathbf{G_0}] \\
&= \Pr[E||\mathbf{G_3}] + \Pr[\mathcal{A}\text{wins} \wedge \neg E||\mathbf{G_3}] \\
&= 1/2 + 1/2 \Pr[\neg E||\mathbf{G_3}] \\
&= 1/2[1 + (q_U + q_S)/|D| + \epsilon],
\end{aligned}$$

so the theorem holds. □

Chapter 6

Conclusion

In this work, we define a functionality called $\mathcal{F}_{\text{OPRF}}$ in the UC model, and provide an efficient realization of it, called 2HashDH, in ROM. 2HashDH forms an improvement of 2HashDH-NIZK [7] in terms of computational costs. We then construct a PPSS scheme using $\mathcal{F}_{\text{OPRF}}$, which is based on the scheme in [7], and prove its security.

Bibliography

[1] A. Bagherzandi, S. Jarecki, N. Saxena, and Y. Lu. Password-protected secret sharing. Cryptology ePrint Archive, Report 2010/561, Nov. 2010. `https://eprint.iacr.org/2010/561`

[2] M. Bellare, C. Namprempre, D. Pointcheval, and M. Semanko. The one-more-rsa-inversion problems and the security of Chaum's blind signature scheme. *Journal of Cryptology*, 16(3):185-215, 2003.

[3] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *CRYPTO*, pages 136-145, 2001.

[4] D. Jablon. Passowrd authentication using multiple servers. In *CT-RSA'01:RSA Cryptographer's Track*, pages 344-360. Springer-Verlag, 2001.

[5] M. J. Freedman, Y. Ishai, B. Pinkas, and O. Reingold. Keyword search and oblivious pseudorandom functions. In J. Kilian, editor, *TCC*, volume 3378 of *Lecture Notes in Computer Science*, pages 303-324. Springer, 2005.

[6] C. Hazay and Y. Lindell. Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries. In R. Canetti, editor, *TCC*, volume 4948 of *Lecture Notes in Computer Science*, pages 155-175. Springer, 2008.

[7] S. Jarecki, A. Kiayias, and H. Krawczyk. Round-optimal password-protected secret

sharing and T-PAKE in the password-only model. Cryptology ePrint Archive, Report 2014/650, Aug. 2014. https://eprint.iacr.org/2014/650

[8] S. Jarecki, X. Liu. Efficient oblivious pseudorandom function with applications to adaptive OT and secure computation of set intersection. In O. Reingold, editor, *TCC*, volume 5444 of *Lecture Notes in Computer Science*, pages 577-594. Springer, 2009.