Towards Secure Computation with Optimal Complexity

Peihan Miao

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2019-47
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-47.html

May 16, 2019

Towards Secure Computation with Optimal Complexity

by

Peihan Miao

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Assistant Professor Sanjam Garg, Chair
Assistant Professor Nikhil Srivastava
Professor Luca Trevisan

Spring 2019

Towards Secure Computation with Optimal Complexity

Abstract

Towards Secure Computation with Optimal Complexity

by

Peihan Miao

Doctor of Philosophy in Computer Science

University of California, Berkeley

Assistant Professor Sanjam Garg, Chair

Secure computation enables a set of mutually distrustful parties to collaboratively compute a public function over their private data without leaking anything apart from the output. In this thesis, we present improvements towards obtaining secure computation with optimal computation, communication, and round complexity, in both *theory* and *practice*.

In Theory. We introduce a novel primitive called *laconic oblivious transfer* (or laconic OT for short), which allows an OT receiver to commit to a large input D (of length M) via a short message. Subsequently, a single short message from an OT sender allows the receiver to learn $m_{D[L]}$, where the messages m_0, m_1 and the location $L \in [M]$ are dynamically chosen by the sender. We present a construction of this primitive based on the Decisional Diffie-Hellman (DDH) assumption, which makes is an innovative use of somewhere statistically binding (SSB) hashing in conjunction with hash proof systems.

Since its introduction, laconic OT has proved to be an extremely powerful tool towards achieving optimal computation and communication complexity in secure computation and beyond. In this thesis, we show a few of its very first applications including non-interactive secure computation and multi-hop homomorphic encryption for RAM programs.

In Practice. As a specific application of secure computation, we introduce a new notion called password-based threshold token authentication, which protects password-based authentication against single point of failures. Specifically, we distribute the role of a single server among n servers and allow any t servers to collectively verify clients' passwords and generate tokens, while no $t-1$ servers can forge a valid token or mount offline dictionary attacks. We then introduce PASTA, a general framework wherein clients can sign on using a two-round (optimal) protocol that meets our strong security guarantees.

Our experiments show that the overhead of PASTA, compared to a naïve single-server solution, is extremely low (1-5%) in the most likely setting where parties communicate over the internet. The overhead is higher for certain MAC tokens over a LAN (though still only a few milliseconds) due to inherent public-key operations in PASTA. We show, however, that public-key operations are necessary by proving a symmetric-key-only solution impossible.

To Mom,
for her unconditional love and support, always.

Contents

List of Figures

List of Tables

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my advisor Sanjam Garg. When I came to Berkeley for graduate study, I was lost for a long time and was not sure what to work on until I fortunately met Sanjam and started working on cryptography. I have been given tremendous freedom in research throughout the years, which helps me find my own passion, shape my own research, and prepare myself as an independent researcher. I have gone though a lot of difficult times in my research. Sanjam has always been there providing assistance, guidance, support, and encouragement. Without his continuous patience and trust, I would not have been able to finish this dissertation.

I am also grateful to Christos Papadimitriou for advising me in my first year, for his patience and inspiration. I was deeply impressed by his enthusiasm and energy for research and I will forever benefit from our interaction.

I would like to thank my dissertation committee members Nikhil Srivastava and Luca Trevisan for their valuable comments and guidance on this work. I also thank Alessandro Chiesa for serving on my qualifying exam committee and for his insightful feedback during the exam.

The theory group at Berkeley has provided me with extraordinary research environment from which I greatly benefited. Thanks to my fellow graduate students Lynn Chua, Fotis Iliopoulos, Jingcheng Liu, Siqi Liu, Pasin Manurangsi, Alexandros Psomas, Manuel Sabin, Aaron Schild, Tselil Schramm, Nick Spooner, Akshayaram Srinivasan, Sam Wong, and everyone else, for the many inspiring and fun conversations and for sharing this exciting and rewarding journey with me.

In the summer of 2017, I had a very enjoyable internship at Microsoft Research Redmond. I wish to thank my mentor Melissa Chase for spending so much time working with me during that three months. Also, I would like to thank Hao Chen, Ranjit Kumaresan, Kim Laine, Kristin Lauter, and Yupeng Zhang for many wonderful moments and conversations.

I also enjoyed my internship at VISA Research. I would like to express my special thanks to Payman Mohassel and everyone else in the Security group: Shashank Agrawal, Mihai Christodorescu, Sivanarayana Gaddam, Atul Luykx, Pratyay Mukherjee, Negin Salajegheh, Rohit Sinha, and Mahdi Zamani. It was such a pleasure to interact and collaborate with all of them. I also had a great time interacting with Dean Galland, Hossein Hamooni, Hao Yang, and Yan Zheng. They together made my internship truly amazing.

I am thankful to Kevin Lewi, who was my intern manager at Facebook. From him I learned a lot about engineering for the industrial infrastructure, which was a unique experience for me.

My gratitude also goes to Karn Seth for hosting me as an intern at Google New York, and for his guidance and support during my internship. I also wish to thank the entire cryptography team: Ben Kreuter, Sarvar Patel, Mariana Raykova, Aaron Segal, Kevin Yeo, and Moti Yung, for making my internship an unforgettable experience.

I would also like to thank all of my co-authors and collaborators during my PhD: Shashank Agrawal, Melissa Chase, Alessandro Chiesa, Chongwon Cho, Nico Döttling, Sanjam Garg,

Matthew Green, Divya Gupta, Ranjit Kumaresan, Jingcheng Liu, Ian Miers, Pratyush Mishra, Payman Mohassel, Pratyay Mukherjee, Omkant Pandey, Sarvar Patel, Antigoni Polychroniadou, Mariana Raykova, Karn Seth, Akshayaram Srinivasan, Prashant Nalini Vasudevan, Moti Yung, and Yupeng Zhang. It was a great pleasure to work with everyone of them and I learned immensely from them.

Before coming to Berkeley, I benefited a lot from the program of ACM Honors Class founded by Yong Yu at Shanghai Jiao Tong University. I probably would not have pursued a PhD if it weren't for this program. I would also like to thank Ning Chen for hosting me at Nanyang Technological University in my senior year and helping me take my first steps as a researcher in theoretical computer science.

Outside of research, I wish to thank my friends Chaoran Guo and Qian Zhong for their warmest company during my good and bad days. Special thanks to my close friend from college, Youer Pu, for always being there to support me throughout the years.

I would like to thank Yu Cheng for bringing happiness, light, and hope to my life. Without you none of this really matters.

Finally, I am forever indebted to my mother Fumei Wang, who raised me up on her own after my father passed away when I was three years old. She taught me courage, determination, perseverance, and compassion for others. She is the most amazing mother in the world.

Chapter 1

Introduction

Secure computation allows mutually distrustful parties to jointly perform computation over their private inputs without disclosing anything more than the output. Since its introduction by Yao in the 1980s [Yao82; Yao86], secure computation has found a wide range of practical applications such as electronic voting, electronic auctions, genome data analysis, secure signal processing, secure outsourcing computation, privacy-preserving machine learning, anonymous transactions, and many more.

Starting with the seminal works [Yao82; Yao86; GMW87; BGW88; CCD88; RB89] that established the *feasibility* of secure computation for any function and any number of parties, a central question in the cryptographic community has been whether secure computation protocols can be *efficient* enough to serve for its countless applications. In the past few decades, both theoretical and practical improvements have been pushing the limits of efficiency of those protocols and have demonstrated impressive improvements. Furthermore, for specific secure computation applications, massive research efforts have been devoted towards designing tailored protocols to achieve better efficiency than generic approaches.

Nevertheless, despite its strong security guarantees, wide range of applications, beautiful feasibility results, and substantial research progress, the adoption of secure computation in real industry is very much limited as of today, the biggest challenge of which still boils down to *efficiency*. Compared to a naïve insecure solution, the expensive cost of a secure computation protocol deters most companies from adopting this technology because very few of them are willing to trade-off their product performance for stronger security. This brings us to a natural question that we would like to, and have to answer:

Can we achieve secure computation with optimal *efficiency?*

In examining the efficiency of a protocol, we consider several different measurement aspects. Our eventual goal, of course, is to achieve the optimal efficiency in every aspect. We briefly discuss these measurements below.

Efficiency Measurements. The most commonly considered efficiency measurement for a protocol is the *computational complexity*, and in particular, the computational overhead

of a secure computation protocol, compared to the naïve protocol that does not have any security guarantee. This measurement is especially important in practice because of limited computational resources.

Communication complexity, which refers to the total amount of communication in a protocol, is also one of the major efficiency bottlenecks in most secure computation applications. This is due to the fact that network bandwidth is often times a shared resource for which multiple applications compete and in general the price for expanding network bandwidth tends to be much higher than the price for expanding computational resources. It is especially crucial to minimize communication complexity when secure computation protocols are applied on large datasets, where requiring the communication complexity to scale with the size of the datasets is usually infeasible for even moderate dataset sizes.

Apart from computation and communication complexity, another important metric in the efficiency of a secure computation protocol is its *round complexity*, that is, the number of rounds of interaction among the parties. As the time for one round of communication is lower bounded by the network latency, a high round complexity can easily become a bottleneck of a protocol especially when adopted on a high-latency network.

With explicit awareness of the limitations of available computational resources, network bandwidth and network latency, in this thesis, we aim to achieve secure computation with *optimal* computation, communication, and round complexity. We make substantial progress in both *theory* and *practice*.

1.1 In Theory

Cryptographic protocols for secure computation are typically based on Boolean circuits, where both the computational complexity and communication complexity scale with the size of the input dataset, which makes it generally unsuitable for large datasets. Substantial research effort has been devoted towards overcoming these challenges, including works on fully-homomorphic encryption (FHE) [Gen09; BV11b; BV11a; GSW13] and on the RAM setting of oblivious RAM [Gol87; Ost90] and secure RAM computation [OS97; GKK+12; LO13; GHL+14; GGMP16]. Protocols based on FHE generally have a favorable communication complexity and are basically non-interactive, yet incur a prohibitively large computational overhead that scales with the dataset size. On the other hand, protocols for the RAM model generally have a favorable computational overhead, but lack in terms of communication efficiency, which grows with the program running time, especially in the multi-party setting. Can we achieve the best of both worlds?

In this work [CDG+17] we make positive progress on this question. Specifically, we introduce a new tool called *laconic oblivious transfer* (or laconic OT for short) that helps to strike a balance between the two seemingly opposing goals. Laconic OT is a key tool towards the goal of improving communication complexity while at the same time maintaining low computational complexity for a variety of scenarios of secure computation.

1.1.1 Laconic Oblivious Transfer

Oblivious transfer (or OT for short), since its first introduction by Rabin [Rab81], has been a foundational building block for realizing computationally efficient secure computation protocols [Yao82; Yao86; GMW87; IPS08]. However, typical secure computation protocols involve executions of multiple instances of an oblivious transfer protocol. In fact, the number of needed oblivious transfers grows with the input size of one of the parties, which is the receiver of the oblivious transfer. As a result, the communication complexity of such protocols has to grow with the input size of the receiver.[1]

We introduce a novel technique for optimizing communication complexity of secure computation over large inputs. Specifically, we provide a new oblivious transfer (OT) protocol with a laconic receiver. Laconic OT allows a receiver to commit to a large input D (of length M) via a short message. Subsequently, a single short message by a sender allows the receiver to learn $m_{D[L]}$, where the messages m_0, m_1 and the location $L \in [M]$ are dynamically chosen by the sender. All prior constructions of OT required the receiver's outgoing message to grow with D.

Our key contribution is an instantiation of this primitive based on the Decisional Diffie-Hellman (DDH) assumption in the common reference string (CRS) model. The technical core of this construction is a novel use of somewhere statistically binding (SSB) hashing in conjunction with hash proof systems. Next, we show applications of laconic OT to non-interactive secure computation on large inputs and multi-hop homomorphic encryption for RAM programs.

1.1.2 Application I: Non-Interactive Secure Computation

Circuit Setting. Can a receiver publish a *small* encoding of her *large* confidential database D so that any sender, who holds a secret input x, can reveal the output $f(x, D)$ (where f is a circuit) to the receiver by sending her a single message? For security, we want the receiver's encoding to hide D and the sender's message to hide x. Using laconic OT, we present the first solution to this problem. In our construction, the receiver's published encoding is *independent* of the size of her database, but we do not restrict the size of the sender's message.[2]

RAM Setting. Consider the scenario where f can be computed using a RAM program P of running time t. We use the notation $P^D(x)$ to denote the execution of the program

[1] We remark that related prior works on OT extension [Bea96; IKNP03; KK13; ALSZ13] makes the number of public key operations performed during protocol executions independent of the receiver's input size. However, the communication complexity of receivers in these protocols still grows with the input size of the receiver.

[2] Solutions for this problem based on fully-homomorphic encryption (FHE) [Gen09; LNO13], unlike our result, reduce the communication cost of both the sender's and the receiver's messages to be independent of the size of D, but require additional rounds of interaction.

P on input x with random access to the database D. We provide a construction where as before the size of the receiver's published message is independent of the size of the database D. Moreover, the size of the sender's message (and computational cost of the sender and the receiver) grows only with t and the receiver learns nothing more than the output $P^D(x)$ and the locations in D touched during the computation. Note that in all prior works on general secure RAM computation [OS97; GKK+12; LO13; WHC+14; GHL+14; GLOS15; GLO15] the size of the receiver's message grew at least with its input size.[3]

1.1.3 Application II: Homomorphic Encryption for RAM

Consider a scenario where S (a server), holding an input x, publishes an encryption ct_0 of her private input x under her public key. Now this ciphertext is passed on to a client Q_1 that homomorphically computes a (possibly private) program P_1 accessing (private) memory D_1 on the value encrypted in ct_0, obtaining another ciphertext ct_1. More generally, the computation could be performed by multiple clients. In other words, clients $Q_2, Q_3, \cdots$ could sequentially compute private programs $P_2, P_3, \cdots$ accessing their own private databases $D_2, D_3, \cdots$. Finally, we want S to be able to use her secret key to decrypt the final ciphertext and recover the output of the computation. For security, we require simulation based security for a client Q_i against a collusion of the server and any subset of the clients, and IND-CPA security for the server's ciphertext.

Though we described the simple case above, we are interested in the general case when computation is performed in different sequences of the clients. Examples of two such computation paths are shown in Figure 1.1. Furthermore, we consider the setting of persistent databases, where each client is able to execute dynamically chosen programs on the encrypted ciphertexts while using the same database that gets updated as these programs are executed.

FHE-Based Solution. Gentry's [Gen09] fully homomorphic encryption (FHE) scheme offers a solution to the above problem when circuit representations of the desired programs $P_1, P_2, \ldots$ are considered. Specifically, S could encrypt her input x using an FHE scheme. Now, the clients can publicly compute arbitrary programs on the encrypted value using a public evaluation procedure. This procedure can be adapted to preserve the privacy of the computed circuit [OPP14; DS16; BdMW16] as well. However, this construction only works for circuits. Realizing the scheme for RAM programs involves first converting the RAM program into a circuit of size at least linear in the size of the database. This linear effort can

[3]The communication cost of the receiver's message can be reduced to depend only on the running time of the program by allowing round complexity to grow with the running time of the program (using Merkle Hashing). Analogous to the circuit case, we remark that FHE-based solutions can make the communication of both the sender and the receiver small, but at the cost of extra rounds. Moreover, in the setting of RAM programs FHE-based solutions additionally incur an increased computational cost for the receiver. In particular, the receiver's computational cost grows with the size of its database.

be exponential in the running time of the program for several applications of interest such as binary search.

Our Relaxation. In obtaining homomorphic encryption for RAM programs, we start by relaxing the compactness requirement in FHE.[4] Compactness in FHE requires that the size of the ciphertexts does not grow with computation. In particular, in our scheme, we allow the evaluated ciphertexts to be bigger than the original ciphertext. Gentry, Halevi and Vaikuntanathan [GHV10] considered an analogous setting for the case of circuits. As in Gentry et al. [GHV10], in our setting computation itself will happen at the time of decryption. Therefore, we additionally require that clients $Q_1, Q_2, \cdots$ first ship pre-processed versions of their databases to S for the decryption, and security will additionally require that S does not learn the access pattern of the programs on client databases. This brings us to the following question:

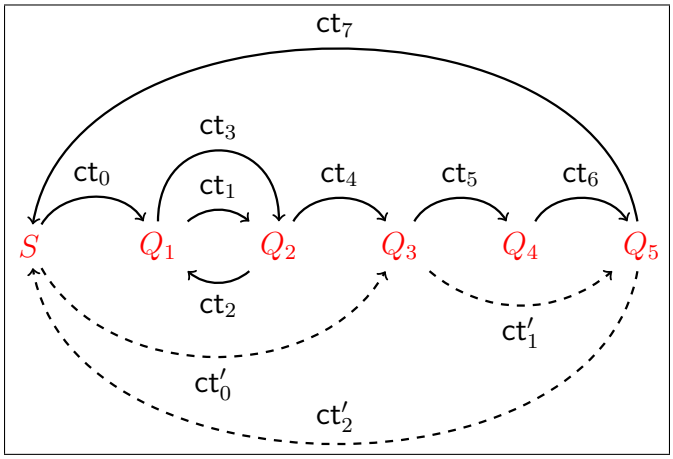


Figure 1.1: Two example paths of computation on server S's ciphertexts.

Can we realize multi-hop encryption schemes for RAM programs where the ciphertext grows linearly only in the running time of the computation performed on it?

We show that laconic OT can be used to realize such a multi-hop homomorphic encryption scheme for RAM programs. Our result bridges the gap between growth in ciphertext size and computational complexity of homomorphic encryption for RAM programs.

Our work also leaves open the problem of realizing (fully or somewhat) homomorphic encryption for RAM programs with (somewhat) compact ciphertexts and for which computational cost grows with the running time of the computation, based on traditional computational assumptions. Our solution for multi-hop RAM homomorphic encryption is for the semi-honest (or, semi-malicious) setting only. We leave open the problem of obtaining a solution in the malicious setting.[5]

[4] One method for realizing homomorphic encryption for RAM programs [GKP+13; GHRW14; CHJV15; BGL+15; KLW15] would be to use obfuscation [GGH+13] based on multilinear maps [GGH13]. However, in this paper we focus on basing homomorphic RAM computation on DDH and defer the work on obfuscation to future work.

[5] Using non-interactive zero-knowledge (NIZK) proofs alone does not solve the problem, because locations accessed during computation are dynamically decided.

1.1.4 Follow-Up Work

Since its introduction, laconic OT has proved to be an extremely powerful tool towards achieving optimal communication complexity as well as computational complexity in various settings of secure computation and beyond. Specifically, it has been utilized as a crucial tool in achieving adaptively secure circuit garbling with nearly optimal online complexity [GS18], adaptive garbled RAM from standard assumptions [GOS18], laconic function evaluation and its applications [QWW18], and so on.

Furthermore, ideas from laconic OT have been useful in making significant progress in areas beyond secure computation, including obtaining Identity-Based Encryption (IBE) from the Computational Diffie-Hellman (CDH) assumption [DG17], anonymous IBE from the CDH assumption [BLSV18], key-dependent message (KDM) secure encryption [BLSV18; DGHM18], reusable designated-verifier NIZKs [LQR+19], etc.

1.2 In Practice

As a specific application of secure computation, we introduce and formalize the notion of password-based threshold token authentication, which protects password-based token authentication against single point of failures.

Token-Based Authentication. Token-based authentication is arguably the most common way we obtain authorized access to resources, services, and applications on the internet and on enterprise networks.

Open standards such as JSON Web Token (JWT) [JWT] and SAML [SAML] are widely used to facilitate single-sign-on authentication by allowing clients to initially sign on using a standard mechanism such as username/password verification to obtain and locally store a token in a cookie or the local storage. The token can then be used for all future accesses to various applications without client involvement, until it expires.

A similar mechanism is used, via open standards such as OAuth [OAuth] and OpenID [OpenID], by many companies including Google, Facebook and Amazon [Google; Facebook; Amazon] to enable their users to share information about their accounts with (or authenticate themselves to) third party applications or websites without revealing their passwords to them.

Finally, network authentication protocols such as Kerberos [Kerberos] are commonly used by enterprises (e.g. Active Directory in Windows Servers) to, periodically but infrequently, authenticate clients with their credentials and issue them a ticket-granting ticket (TGT) that they can use to request access to various services on the enterprise network such as printers, internal web and more.

It is therefore no surprise that most software-based secret management systems provide tokens as a primary method for authenticating clients. For example, consider the following statement from the popular open source solution Vault by Hashicorp [Vault]:

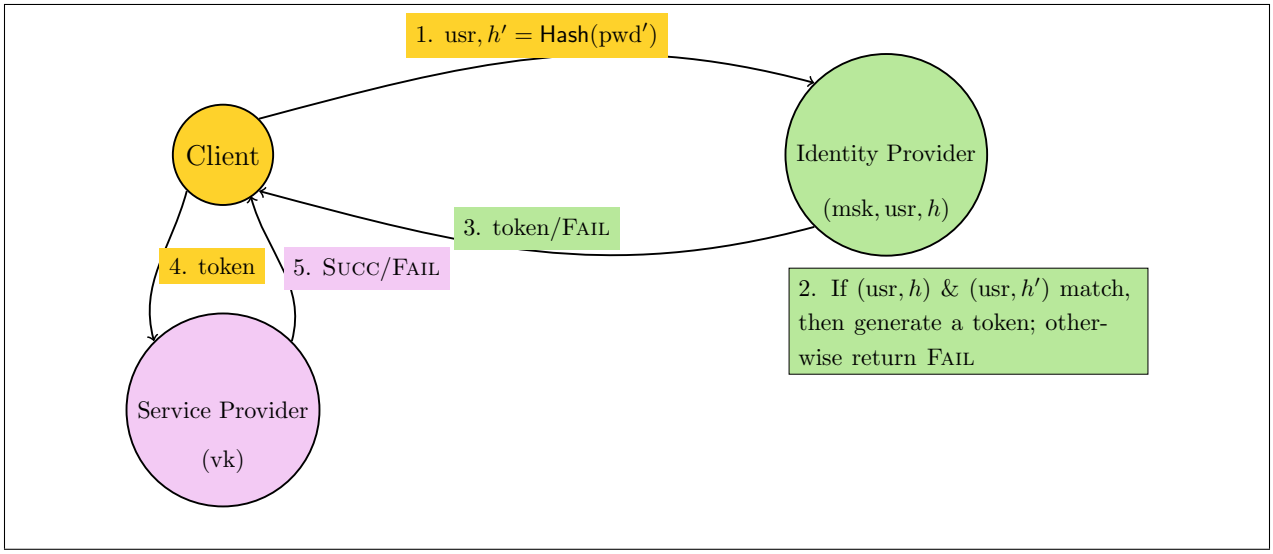


Figure 1.2: Generic flow diagram of commonly used password-based token generation solutions. The figure shows only the sign-on phase which is preceded by a one-time registration phase (not shown) where the client stores its username (usr) and the hash (h) of its password with the identity provider. msk is the secret key used for generating tokens and vk is used for verifying them.

> The token auth method is built-in and is at the core of client authentication. Other auth methods may be used to authenticate a client, but they eventually result in the generation of a client token managed by the token backend.

In all these cases, the authentication flow is effectively the same. A client signs on with its username/password, typically by sending hash of its password to an identity provider. The identity server who stores the username along with its hashed passwords as part of a registration phase, verifies the client's credential by matching the hash during the sign-on process before issuing an authentication token using a master secret key (see Figure 1.2). The token is generated by computing a digital signature or a message authentication code (all the above-mentioned standards support both digital signatures and MACs) on a message that can contain client's information/attributes, expiration time and a policy that would control the nature of access. The token is later verified by an application server which holds the verification key (for MACs this is equal to the master secret key). See Figure 1.3 for a sample JWT authenticated using HMAC [KBC97]. Note that the only secret known to the client is its password, and the device the client uses for access stores the temporary authentication token on its behalf. Besides this temporary (and often restricted) token, client devices do not store any long term secrets that are used to authenticate the client.

However, such an identity provider is a *single point of failure* that if breached, enables an attacker to (i) recover the master secret key and forge arbitrary tokens that enable access to arbitrary resources and information in the system and (ii) obtain hashed passwords to use

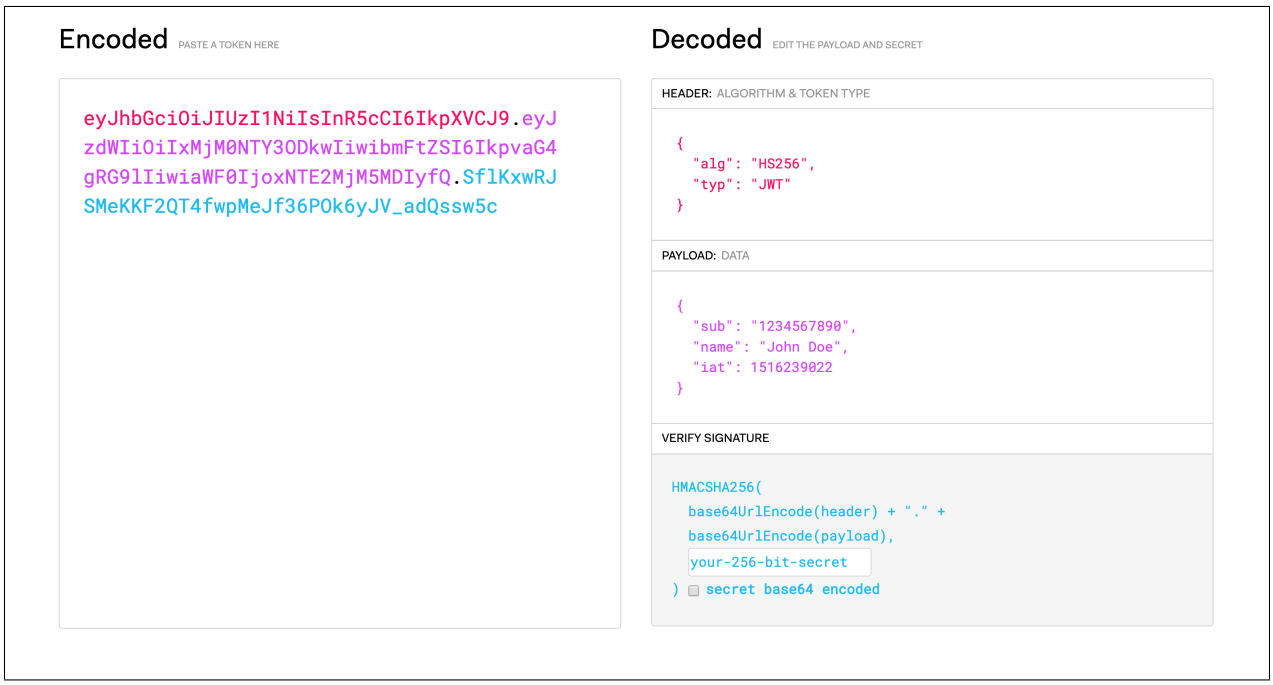


Figure 1.3: A sample JSON Web Token [JWT] that uses HMAC. The base64 encoded token on the left is what is sent and stored. When decoded, it contains a header with algorithm and token type, a payload that includes various attributes and a HMAC of header/payload.

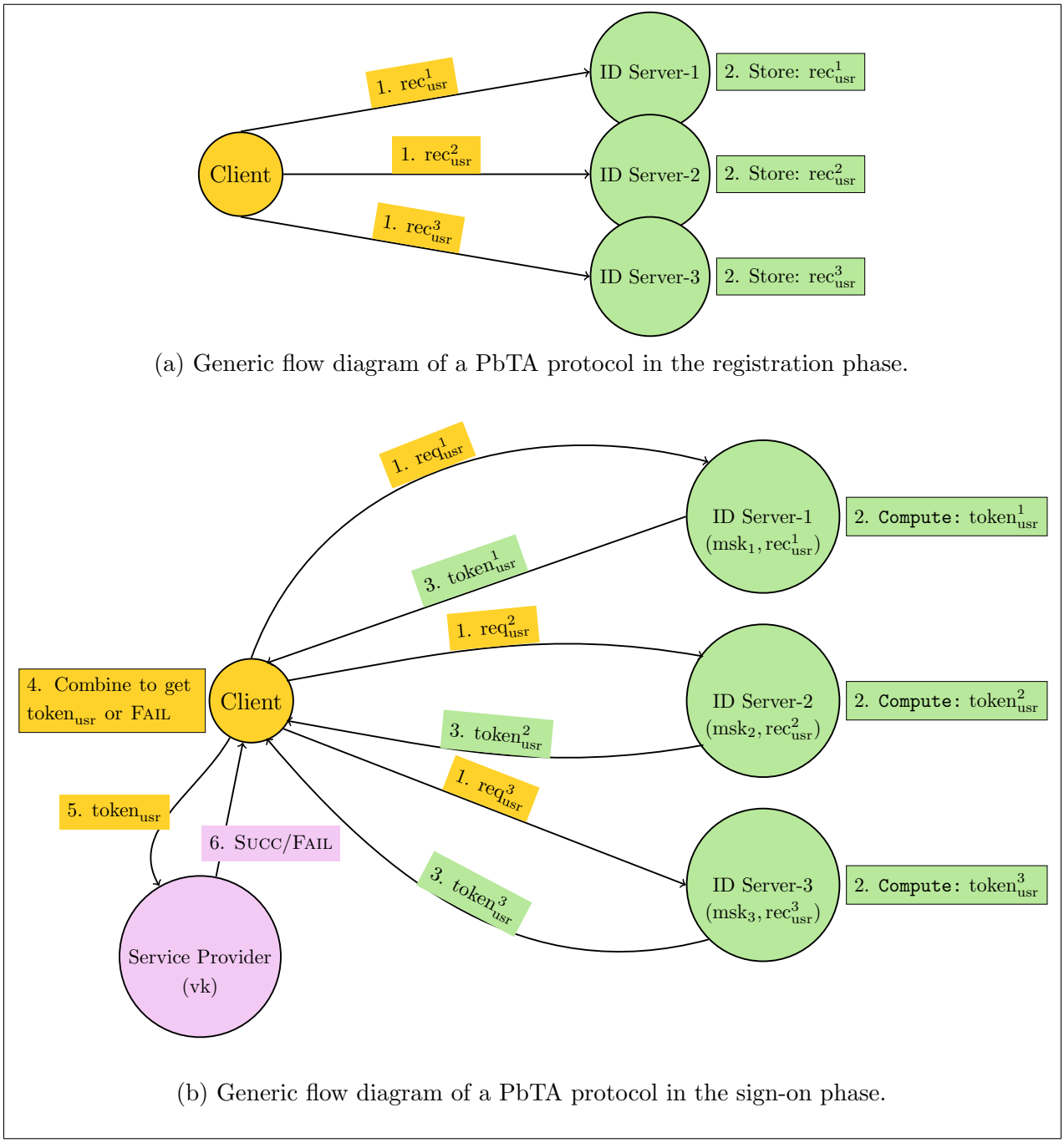
as part of an offline dictionary attack to recover client credentials.

In this work [AMMM18], we propose the notion of Password-based Threshold Authentication (PbTA) for distributing the role of the identity provider among n servers who collectively verify clients' passwords and generate authentication tokens for them (see Figure 1.4 for a generic flow). PbTA enables any t ($2 \leq t \leq n$) servers to authenticate the client and generate valid tokens while any attacker who compromises at most $t-1$ servers cannot forge valid tokens *or* mount offline dictionary attacks, thus providing very strong *unforgeability* and *password-safety* properties.

This functionality is a specific application of secure computation and in the protocol design round complexity is the major bottleneck. In order to achieve the optimal performance we minimize the round of interaction between clients and servers.

Our Contributions. We formally introduce the notion of Password-based Threshold Authentication (PbTA) with the goal of making password-based token generation secure against server breaches that could compromise both long-term keys and user credentials. Our contributions are as follows:

- **Defining Security for PbTA.** We formalize password-based threshold authentication, and establish the necessary security requirements of token unforgeability and password-safety in presence of an adversary who may breach a subset of the identity



(a) Generic flow diagram of a PbTA protocol in the registration phase.

(b) Generic flow diagram of a PbTA protocol in the sign-on phase.

Figure 1.4: Generic flow diagrams of a PbTA protocol.

servers. Our game-based definitions are strong and intuitive, and consider security in a multi-client setting where many clients use the same identity provider. Adversary could corrupt clients in an adaptive fashion during the game. We note that an alternative approach would be to use the Universal Composability framework [Can01] as followed in some prior work involving password-based authentication (e.g. [JKKX17]). We chose to focus on game-based definitions that are much simpler to work with but comprehensive enough to cover a very broad set of attack scenarios.

- **The PASTA Framework with Minimal Interaction.** We propose a general framework called PASTA, that uses as building blocks any threshold oblivious pseudorandom function (TOPRF) and any threshold token generation (TTG) scheme, i.e. a threshold MAC or digital signature. PASTA meets our stringent security requirements for PbTA.

  After a one-time registration phase, a client just needs to remember its password. It can sign-on using a two-round protocol wherein the servers do not talk to each other (assuming only that the servers communicate to the client over an authenticated channel). Therefore, PASTA requires *minimal* interaction.

  The sign-on protocol ensures that if the client's password is correct, he obtains a valid authentication token: client sends a request message to a subset of the servers, and servers respond with messages of their own. If the client password is a match, it can combine server responses to obtain a valid token (see Figure 1.4).[6] Otherwise, it does not learn anything.

  At the first glance, it may seem unnatural to define a general framework that works for both symmetric-key tokens (i.e. MAC) and public-key tokens (i.e. digital signature). Though their verification procedures are different in terms of being private or public, note that their token generation procedures are both private. PASTA focuses on generating tokens, hence it works for both types of tokens.

- **Instantiations and Implementation.** We instantiate and implement our framework in C++ with four different threshold token generation schemes: block-cipher based and DDH-based threshold MACs of Naor et al. [NPR99], threshold RSA-based signature of Shoup [Sho00] and threshold pairing-based signature of Boldyreva [Bol03]. Each instantiation has its own advantages and disadvantages. When instantiated with a threshold MAC, we obtain a more efficient solution but the tokens are not publicly verifiable, i.e. vk in Figure 1.2 and Figure 1.4b stored in the application server would be the same as the master secret key msk, since the verifier needs the secret

[6] Note that in this setting, as opposed to the naïve solution (Figure 1.2), no matching takes place on the identity provider side. In particular, an ID-server does not check against a record stored in the registration phase, because, if it did, one can easily see that offline attacks would be possible even if a single server is breached.

key for verification.[7] PASTA with RSA-based and pairing-based token generation are more expensive but are publicly verifiable. Among the signature-based solutions, the pairing-based one is faster since signing does not require pairings but the RSA-based solution has faster verification and produces signatures that are compatible with legacy applications. To the best of our knowledge, our work is also the first to implement several of the threshold token generation schemes (*not* password-based) and report on their performance.

Our experiments show that the overhead of obtaining security against server breaches using PASTA, in the sign-on stage, is at most 5% compared to the naïve solution of using hashed passwords and a single-server token generation, in the most likely scenario where clients connect to servers over the internet (a WAN network). This is primarily due to the fact that in this case, network latency dominates the total runtime for all token types. The overhead is a bit higher in the LAN setting but the total runtime of sign-on (steps 1-4 in Figure 1.4b) is still very fast, ranging from 1.3 ms for $(n, t) = (3, 2)$ with a symmetric-key MAC token to 23 ms for $(n, t) = (10, 10)$ with an RSA-based token, where n is the number of servers and t is the threshold.

- **Necessity of Computational Overhead.** PASTA has its largest overhead compared to the naïve single-server solution, for symmetric-key based tokens in the LAN setting. This is because public-key operations dominate PASTA's runtime while the naïve solution only involves symmetric-key operations. Nevertheless, we show that this inefficiency is inherent by proving that public-key operations are necessary to achieve our notion of PbTA.

1.3 Organization

In Chapter 2 we introduce the notion of laconic OT formally and present a DDH-based construction. In Chapter 3 we present the first application of laconic OT to non-interactive secure computation on large inputs in the circuit setting as well as RAM setting. In Chapter 4 we formally define the setting of multi-hop homomorphic encryption for RAM programs and present our construction using laconic OT.

In Chapter 5 we formalize password-based threshold authentication and propose the PASTA framework with minimal round complexity as well as present experimental results for various types of tokens.

[7] To achieve better security of the secret key, the verification process can also be made distributed using a standard threshold MAC scheme. We omit the distributed verification in the rest of this paper because it is not our focus.

Chapter 2

Laconic Oblivious Transfer

In this chapter, we introduce the notion of laconic oblivious transfer (or laconic OT for short). Laconic OT allows an OT receiver to commit to a large input $D \in \{0,1\}^M$ via a short message. Subsequently, the sender responds with a single short message to the receiver depending on dynamically chosen two messages m_0, m_1 and a location $L \in [M]$. The sender's response message allows the receiver to recover $m_{D[L]}$ (while $m_{1-D[L]}$ remains computationally hidden). Furthermore, without any additional communication with the receiver, the sender could repeat this process for multiple choices of L.

Our construction of laconic OT is obtained by first realizing a "mildly compressing" laconic OT protocol for which the receiver's message is factor-2 compressing, i.e., half the size of its input. We base this construction on the Decisional Diffie-Hellman (DDH) assumption. We note that, subsequent to our work, the factor-2 compression construction has been simplified by Döttling and Garg [DG17] (another alternative simplification can be obtained using [AIKW13]). Next we show that such a "mildly compressing" laconic OT can be bootstrapped, via the usage of a Merkle Hash Tree and Yao's Garbled Circuits [Yao82; Yao86], to obtain a "fully compressing" laconic OT, where the size of the receiver's message is independent of its input size.

The laconic OT scheme with a Merkle Tree structure allows for good properties like local verification and local updates, which makes it a powerful tool in secure computation with large inputs. We show new applications of laconic OT to non-interactive secure computation in Chapter 3 and homomorphic encryption for RAM programs in Chapter 4.

Organization. In Section 2.1 we give a technical overview of this work. We introduce the notion of laconic OT formally in Section 2.2 and give a construction with factor-2 compression in Section 2.3, which can be bootstrapped to a fully compressing updatable laconic OT in Section 2.4.

2.1 Technical Overview

In this section we provide an overview of laconic OT and our constructions of this new primitive.

Definition at a High Level. Laconic OT consists of two major components: a hash function and an encryption scheme. We call the hash function Hash and the encryption scheme $(\mathsf{Send}, \mathsf{Receive})$. In a nutshell, laconic OT allows a receiver R to compute a *succinct* digest digest of a large database D and a private state $\hat{\mathsf{D}}$ using the hash function Hash. After digest is made public, anyone can non-interactively send OT messages to R w.r.t. a location L of the database such that the receiver's choice bit is $D[L]$. Here, $D[L]$ is the database-entry at location L. In more detail, given digest, a database location L, and two messages m_0 and m_1, the algorithm Send computes a ciphertext e such that R, who owns $\hat{\mathsf{D}}$, can use the decryption algorithm $\mathsf{Receive}$ to decrypt e to obtain the message $m_{D[L]}$.

For security, we require sender privacy against semi-honest receiver. In particular, given an honest receiver's view, which includes the database D, the message $m_{1-D[L]}$ is computationally hidden. We formalize this using a simulation based definition. On the other hand, we do not require receiver privacy as opposed to standard oblivious transfer, namely, no security guarantee is provided against a cheating (semi-honest) sender. This is mostly for ease of exposition. Nevertheless, adding receiver privacy to laconic OT can be done in a straightforward manner via the usage of garbled circuits and two-message OT (see Section 2.2.1 for a detailed discussion).

For efficiency, we have the following requirement: First, the size of digest only depends on the security parameter and is independent of the size of the database D. Moreover, after digest and $\hat{\mathsf{D}}$ are computed by Hash, the workload of *both* the sender and receiver (that is, the runtime of both Send and $\mathsf{Receive}$) becomes essentially independent of the size of the database (i.e., depending at most polynomially on $\log(|D|)$).

Notice that our security definition and efficiency requirement immediately imply that the Hash algorithm used to compute the succinct digest must be collision resistant. Thus, it is clear that the hash function must be keyed and in our case it is keyed by a common reference string.

Construction at a High Level. We first construct a laconic OT scheme with factor-2 compression, which compresses a 2κ-bit database to a κ-bit digest. Next, to get laconic OT for databases of arbitrary size, we bootstrap this construction using an interesting combination of Merkle hashing and garbled circuits. Below, we give an overview of each of these steps.

2.1.1 Laconic OT with Factor-2 Compression

We start with a construction of a laconic OT scheme with factor-2 compression, i.e., a scheme that hashes a 2κ-bit database to a κ-bit digest. This construction is inspired by the notion

of witness encryption [GGSW13]. We first explain the scheme based on witness encryption. Then, we show how this specific witness encryption scheme can be realized with the more standard notion of hash proof systems (HPS) [CS02]. Our overall scheme is based on the security of Decisional Diffie-Hellman (DDH) assumption.

Construction Using Witness Encryption. Recall that a witness encryption scheme is defined for an NP-language $\mathcal{L}$ (with corresponding witness relation $\mathcal{R}$). It consists of two algorithms Enc and Dec. The algorithm Enc takes as input a problem instance x and a message m, and produces a ciphertext. A recipient of the ciphertext can use Dec to decrypt the message if $x \in \mathcal{L}$ and the recipient knows a witness w such that $\mathcal{R}(x, w)$ holds. There are two requirements for a witness encryption scheme, correctness and security. Correctness requires that if $\mathcal{R}(x, w)$ holds, then $\mathsf{Dec}(x, w, \mathsf{Enc}(x, m)) = m$. Security requires that if $x \notin \mathcal{L}$, then $\mathsf{Enc}(x, m)$ computationally hides m.

We now discuss how to construct a laconic OT with factor-2 compression using a two-to-one hash function and witness encryption. Let $\mathsf{H} : \mathcal{K} \times \{0,1\}^{2\kappa} \to \{0,1\}^{\kappa}$ be a keyed hash function, where $\mathcal{K}$ is the key space. Consider the language $\mathcal{L} = \{(K, L, y, b) \in \mathcal{K} \times [2\kappa] \times \{0,1\}^{\kappa} \times \{0,1\} \mid \exists D \in \{0,1\}^{2\kappa}$ such that $\mathsf{H}(K, D) = y$ and $D[L] = b\}$. Let $(\mathsf{Enc}, \mathsf{Dec})$ be a witness encryption scheme for the language $\mathcal{L}$.

The laconic OT scheme is as follows: The Hash algorithm computes $y = \mathsf{H}(K, D)$ where K is the common reference string and $D \in \{0,1\}^{2\kappa}$ is the database. Then y is published as the digest of the database. The Send algorithm takes as input K, y, a location L, and two messages (m_0, m_1) and proceeds as follows. It computes two ciphertexts $\mathsf{e}_0 \leftarrow \mathsf{Enc}((K, L, y, 0), m_0)$ and $\mathsf{e}_1 \leftarrow \mathsf{Enc}((K, L, y, 1), m_1)$ and outputs $\mathsf{e} = (\mathsf{e}_0, \mathsf{e}_1)$. The $\mathsf{Receive}$ algorithm takes as input K, L, y, D, and the ciphertext $\mathsf{e} = (\mathsf{e}_0, \mathsf{e}_1)$ and proceeds as follows. It sets $b = D[L]$, computes $m \leftarrow \mathsf{Dec}((K, L, y, b), D, \mathsf{e}_b)$ and outputs m.

It is easy to check that the above scheme satisfies correctness. However, we run into trouble when trying to prove sender privacy. Since H compresses 2κ bits to κ bits, most hash values have exponentially many pre-images. This implies that for most values of (K, L, y), it holds that both $(K, L, y, 0) \in \mathcal{L}$ and $(K, L, y, 1) \in \mathcal{L}$, that is, most problem instances are yes-instances. However, to reduce sender privacy of our scheme to the security of witness encryption, we ideally want that if $y = \mathsf{H}(K, D)$, then $(K, L, y, D[L]) \in \mathcal{L}$ while $(K, L, y, 1 - D[L]) \notin \mathcal{L}$. To overcome this problem, we use a somewhere statistically binding hash function that allows us to artificially introduce no-instances as described below.

Somewhere Statistically Binding Hash to the Rescue. Somewhere statistically binding (SSB) hash functions [HW15; KLW15; OPWW15] support a special key generation procedure such that the hash value information theoretically fixes certain bit(s) of the pre-image. In particular, the special key generation procedure takes as input a location L and generates a key $K^{(L)}$. Then the hash function keyed by $K^{(L)}$ will bind the L-th bit of the pre-image. That is, $K^{(L)}$ and $y = \mathsf{H}(K^{(L)}, D)$ uniquely determines $D[L]$. The security requirement for

SSB hashing is the *index-hiding* property, i.e., keys $K^{(L)}$ and $K^{(L')}$ should be computationally indistinguishable for any $L \neq L'$.

We can now establish security of the above laconic OT scheme when instantiated with SSB hash functions. To prove security, we will first replace the key K by a key $K^{(L)}$ that statistically binds the L-th bit of the pre-image. The index hiding property guarantees that this change goes unnoticed. Now for every hash value $y = \mathsf{H}(K^{(L)}, D)$, it holds that $(K, L, y, D[L]) \in \mathcal{L}$ while $(K, L, y, 1 - D[L]) \notin \mathcal{L}$. We can now rely on the security of witness encryption to argue that $\mathsf{Enc}((K^{(L)}, L, y, 1 - D[L]), m_{1-D[L]})$ computationally hides the message $m_{1-D[L]}$.

Working with DDH. The above described scheme relies on a witness encryption scheme for the language $\mathcal{L}$. We note that witness encryption for general NP languages is only known under strong assumptions such as graded encodings [GGSW13] or indistinguishability obfuscation [GGH+13]. Nevertheless, the aforementioned laconic OT scheme does not need full power of general witness encryption. In particular, we leverage the fact that hash proof systems [CS02] can be used to construct statistical witness encryption schemes for specific languages [GGSW13]. Towards this end, we will carefully craft an SSB hash function that is hash proof system friendly, that is, allows for a hash proof system (or statistical witness encryption) for the language $\mathcal{L}$ required above. Our construction of the HPS-friendly SSB hash is based on the Decisional Diffie-Hellman assumption and is inspired from a construction by Okamoto et al. [OPWW15].

We briefly outline our HPS-friendly SSB hash below. We strongly encourage the reader to see Section 2.3.2 for the full construction or see [DG17] for a simplified construction.

Let $\mathbb{G}$ be a (multiplicative) cyclic group of order p generated by a generator g. A hashing key is of the form $\hat{\mathbf{H}} = g^{\mathbf{H}}$ (the exponentiation is done component-wisely), where the matrix $\mathbf{H} \in \mathbb{Z}_p^{2\times 2\kappa}$ is chosen uniformly at random. The hash function of $\mathbf{x} \in \mathbb{Z}_p^{2\kappa}$ is computed as $\mathsf{H}(\hat{\mathbf{H}}, \mathbf{x}) = \hat{\mathbf{H}}^{\mathbf{x}} \in \mathbb{G}^2$ (where $(\hat{\mathbf{H}}^{\mathbf{x}})_i = \prod_{k=1}^{2\kappa} \hat{\mathbf{H}}_{i,k}^{x_k}$, hence $\hat{\mathbf{H}}^{\mathbf{x}} = g^{\mathbf{H}\mathbf{x}}$). The binding key $\hat{\mathbf{H}}^{(i)}$ is of the form $\hat{\mathbf{H}}^{(i)} = g^{\mathbf{A}+\mathbf{T}}$, where $\mathbf{A} \in \mathbb{Z}_p^{2\times 2\kappa}$ is a random rank 1 matrix, and $\mathbf{T} \in \mathbb{Z}_p^{2\times 2\kappa}$ is a matrix with zero entries everywhere, except that $\mathbf{T}_{2,i} = 1$.

Now we describe a witness encryption scheme $(\mathsf{Enc}, \mathsf{Dec})$ for the language $\mathcal{L} = \{(\hat{\mathbf{H}}, i, \hat{\mathbf{y}}, b) \mid \exists \mathbf{x} \in \mathbb{Z}_p^{2\kappa} \text{ s.t. } \hat{\mathbf{H}}^{\mathbf{x}} = \hat{\mathbf{y}} \text{ and } x_i = b\}$. $\mathsf{Enc}((\hat{\mathbf{H}}, i, \hat{\mathbf{y}}, b), m)$ first sets

$$\hat{\mathbf{H}}' = \begin{pmatrix} \hat{\mathbf{H}} \\ g^{\mathbf{e}_i^\top} \end{pmatrix} \in \mathbb{G}^{3\times 2\kappa}, \hat{\mathbf{y}}' = \begin{pmatrix} \hat{\mathbf{y}} \\ g^b \end{pmatrix} \in \mathbb{G}^3,$$

where $\mathbf{e}_i \in \mathbb{Z}_p^{2\kappa}$ is the i-th unit vector. It then picks a random $\mathbf{r} \in \mathbb{Z}_p^3$ and computes a ciphertext $c = \left(\left((\hat{\mathbf{H}}')^\top\right)^{\mathbf{r}}, \left((\hat{\mathbf{y}}')^\top\right)^{\mathbf{r}} \oplus m\right)$. To decrypt a ciphertext $c = (\hat{\mathbf{h}}, z)$ given a witness $\mathbf{x} \in \mathbb{Z}_p^{2\kappa}$, we compute $m = z \oplus \hat{\mathbf{h}}^{\mathbf{x}}$. It is easy to check correctness. For the security proof, see Section 2.3.3.

2.1.2 Laconic OT with Arbitrary Compression

We now provide a bootstrapping technique that constructs a laconic OT scheme with arbitrary compression factor from one with factor-2 compression. Let ℓOT_{const} denote a laconic OT scheme with factor-2 compression.

Bootstrapping Hash Function via a Merkle Tree. A binary Merkle tree is a natural way to construct hash functions with an arbitrary compression factor from two-to-one hash functions, and this is exactly the route we pursue. A binary Merkle tree is constructed as follows: The database is split into blocks of κ bits, each of which forms the leaf of the tree. An interior node is computed as the hash value of its two children via a two-to-one hash function. This structure is defined recursively from the leaves to the root. When we reach the root node (of κ bits), its value is defined to be the (succinct) hash value or digest of the entire database. This procedure defines the hash function.

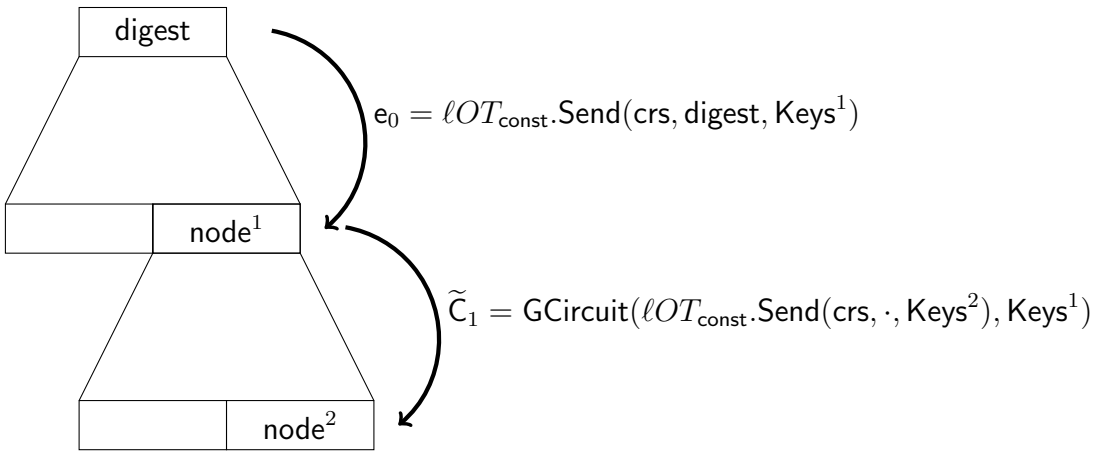
The next step is to define the laconic OT algorithms Send and $\mathsf{Receive}$ for the above hash function. Our first observation is that given the digest, the sender can transfer specific messages corresponding to the values of the left and right children of the root (via 2κ executions of $\ell OT_{\mathsf{const}}.\mathsf{Send}$). Hence, a naive approach for the sender is to output ℓOT_{const} encryptions for the path of nodes from the root to the leaf of interest. This approach runs into an immediate issue because to compute ℓOT_{const} encryptions at any layer other than the root, the sender needs to know the value at that internal node. However, in the scheme a sender only knows the value of the root and nothing else.

Traversing Merkle Tree via Garbled Circuits. Our main idea to make the above naive idea work is via an interesting usage of garbled circuits. At a high level, the sender will output a sequence of garbled circuits (one per layer of the tree) to transfer messages corresponding to the path from the root to the leaf containing the L-th bit, so that the receiver can traverse the Merkle tree from the root to the leaf as illustrated in Figure 2.1.

In more detail, the construction works as follows: The Send algorithm outputs ℓOT_{const} encryptions using the root digest and a collection of garbled circuits, one per layer of the Merkle tree. The i-th circuit has a bit b hardwired in it, which specifies whether the path should go to the left or right child at the i-th layer. It takes as input a pair of sibling nodes $(\mathsf{node}_0, \mathsf{node}_1)$ along the path at layer i and outputs ℓOT_{const} encryptions corresponding to nodes on the path at layer $i+1$ w.r.t. node_b as the hash value. Conceptually, the circuit computes ℓOT_{const} encryptions for the next layer.

The ℓOT_{const} encryptions at the root encrypt the input keys of the first garbled circuit. In the garbled circuit at layer i, the messages being encrypted/sent correspond to the input keys of the garbled circuit at layer $i+1$. The last circuit takes two sibling leaves as input which contains $D[L]$, and outputs ℓOT_{const} encryptions of m_0 and m_1 corresponding to location L (among the 2κ locations).

Given a laconic OT ciphertext, which consists of ℓOT_{const} ciphertexts w.r.t. the root digest and a sequence of garbled circuits, the receiver can traverse the Merkle tree as follows.



Above GCircuit is a circuit garbling procedure, which garbles the circuit $\ell OT_{\mathsf{const}}.\mathsf{Send}(\mathsf{crs}, \cdot, \mathsf{Keys}^2)$ using input keys Keys^1 (see Section 2.4.1.1 for the definition of garbled circuits).

Figure 2.1: Bootstrapping step of laconic OT.

First he runs $\ell OT_{\mathsf{const}}.\mathsf{Receive}$ for the ℓOT_{const} ciphertexts using as witness the children of the root, obtaining the input labels corresponding to these to be fed into the first garbled circuit. Next, he uses the input labels to evaluate the first garbled circuit, obtaining ℓOT_{const} ciphertexts for the second layer. He then runs $\ell OT_{\mathsf{const}}.\mathsf{Receive}$ again for these ciphertexts using as witness the children of the second node on the path. This procedure continues till the last layer.

Security of the construction can be established using the sender security of $\ell OT_{\mathsf{const}}.\mathsf{Receive}$ and simulation based security of the circuit garbling scheme.

Extension. Finally, for our RAM applications we need a slightly stronger primitive which we call *updatable laconic OT* that additionally allows for modifications/writes to the database while ensuring that the digest is updated in a consistent manner. The construction sketched in this paragraph can be modified to support this stronger notion. For a detailed description of this notion refer to Section 2.2.2.

2.2 Definitions of Laconic OT

In this section, we give formal definitions for *Laconic OT* (or, ℓOT for short). We will start by describing laconic OT and then provide an extension of it to the notion of updatable laconic OT.

2.2.1 Laconic OT

Definition 2.2.1 (Laconic OT)**.** *A laconic OT (ℓOT) scheme syntactically consists of four algorithms* crsGen*,* Hash*,* Send *and* $\mathsf{Receive}$*.*

- $\mathsf{crs} \leftarrow \mathsf{crsGen}(1^\kappa)$*. It takes as input the security parameter* 1^κ *and outputs a common reference string* crs*.*
- $(\mathsf{digest}, \hat{\mathsf{D}}) \leftarrow \mathsf{Hash}(\mathsf{crs}, D)$*. It takes as input a common reference string* crs *and a database* $D \in \{0,1\}^*$ *and outputs a digest* digest *of the database and a state* $\hat{\mathsf{D}}$*.*
- $\mathsf{e} \leftarrow \mathsf{Send}(\mathsf{crs}, \mathsf{digest}, L, m_0, m_1)$*. It takes as input a common reference string* crs*, a digest* digest*, a database location* $L \in \mathbb{N}$ *and two messages* m_0 *and* m_1 *of length* κ*, and outputs a ciphertext* e*.*
- $m \leftarrow \mathsf{Receive}^{\hat{\mathsf{D}}}(\mathsf{crs}, \mathsf{e}, L)$*. This is a RAM algorithm with random read access to* $\hat{\mathsf{D}}$*. It takes as input a common reference string* crs*, a ciphertext* e*, and a database location* $L \in \mathbb{N}$*. It outputs a message* m*.*

We require the following properties of an ℓOT scheme $(\mathsf{crsGen}, \mathsf{Hash}, \mathsf{Send}, \mathsf{Receive})$*.*

- ***Correctness.*** *We require that it holds for any database D of size at most $M = \mathsf{poly}(\kappa)$ for any polynomial function $\mathsf{poly}(\cdot)$, any memory location $L \in [M]$, and any pair of messages $(m_0, m_1) \in \{0,1\}^\kappa \times \{0,1\}^\kappa$ that*

$$\Pr\left[m = m_{D[L]} \;\middle|\; \begin{array}{ccc} \mathsf{crs} & \leftarrow & \mathsf{crsGen}(1^\kappa) \\ (\mathsf{digest}, \hat{\mathsf{D}}) & \leftarrow & \mathsf{Hash}(\mathsf{crs}, D) \\ \mathsf{e} & \leftarrow & \mathsf{Send}(\mathsf{crs}, \mathsf{digest}, L, m_0, m_1) \\ m & \leftarrow & \mathsf{Receive}^{\hat{\mathsf{D}}}(\mathsf{crs}, \mathsf{e}, L) \end{array} \right] = 1,$$

  where the probability is taken over the random choices made by crsGen *and* Send*.*

- ***Sender Privacy Against Semi-Honest Receivers.*** *There exists a PPT simulator* $\ell\mathsf{OTSim}$ *such that the following holds. For any database D of size at most $M = \mathsf{poly}(\kappa)$ for any polynomial function $\mathsf{poly}(\cdot)$, any memory location $L \in [M]$, and any pair of messages $(m_0, m_1) \in \{0,1\}^\kappa \times \{0,1\}^\kappa$, let* $\mathsf{crs} \leftarrow \mathsf{crsGen}(1^\kappa)$ *and* $\mathsf{digest} \leftarrow \mathsf{Hash}(\mathsf{crs}, D)$*. Then it holds that*

$$(\mathsf{crs}, \mathsf{Send}(\mathsf{crs}, \mathsf{digest}, L, m_0, m_1)) \stackrel{c}{\approx} \left(\mathsf{crs}, \ell\mathsf{OTSim}(D, L, m_{D[L]})\right).$$

- ***Efficiency.*** *The length of* digest *is a fixed polynomial in κ independent of the size of the database; we will assume for simplicity that* $|\mathsf{digest}| = \kappa$*. Moreover, the algorithm* Hash *runs in time* $|D| \cdot \mathsf{poly}(\log |D|, \kappa)$*,* Send *and* $\mathsf{Receive}$ *run in time* $\mathsf{poly}(\log |D|, \kappa)$*.*

Receiver Privacy. In the above definition, we do not require receiver privacy as opposed to standard oblivious transfer, namely, no security guarantee is provided against a cheating (semi-honest) sender. This is mostly for ease of exposition. We would like to point out that adding receiver privacy (i.e., standard simulation based security against a semi-honest sender) to laconic OT can be done in a straightforward way. Instead of sending digest directly from the receiver to the sender and sending e back to the receiver, the two parties compute Send together via a two-round secure 2PC protocol, where the input of the receiver is digest and the input of the sender is (L, m_0, m_1), and only the receiver obtains the output e. This can be done using standard two-message OT and garbled circuits.

Multiple Executions of Send Sharing the Same digest. Notice that since the common reference string is public (i.e., not chosen by the simulator), the sender can involve Send function multiple times while still ensuring that security can be argued from the above definition (for the case of single execution) via a standard hybrid argument.

It will be convenient to use the following shorthand notations (generalizing the above notions) to run laconic OT for every single element in a database. Let $\mathsf{Keys} = ((\mathsf{Key}_{1,0}, \mathsf{Key}_{1,1}), \ldots, (\mathsf{Key}_{M,0}, \mathsf{Key}_{M,1}))$ be a list of $M = |D|$ key-pairs, where each key is of length κ. Then we define

$$\mathsf{Send}(\mathsf{crs}, \mathsf{digest}, \mathsf{Keys}) = \\ \left(\mathsf{Send}(\mathsf{crs}, \mathsf{digest}, 1, \mathsf{Key}_{1,0}, \mathsf{Key}_{1,1}), \ldots, \mathsf{Send}(\mathsf{crs}, \mathsf{digest}, M, \mathsf{Key}_{M,0}, \mathsf{Key}_{M,1})\right).$$

Likewise, for a vector $\mathsf{e} = (\mathsf{e}_1, \ldots, \mathsf{e}_M)$ of ciphertexts define

$$\mathsf{Receive}^{\hat{\mathsf{D}}}(\mathsf{crs}, \mathsf{e}) = \left(\mathsf{Receive}^{\hat{\mathsf{D}}}(\mathsf{crs}, \mathsf{e}_1, 1), \ldots, \mathsf{Receive}^{\hat{\mathsf{D}}}(\mathsf{crs}, \mathsf{e}_M, M)\right).$$

Similarly, let $\mathsf{Labels} = \mathsf{Keys}_D = (\mathsf{Key}_{1,D[1]}, \ldots, \mathsf{Key}_{M,D[M]})$, and define

$$\ell\mathsf{OTSim}(\mathsf{crs}, D, \mathsf{Labels}) = \left(\ell\mathsf{OTSim}(\mathsf{crs}, D, 1, \mathsf{Key}_{1,D[1]}), \ldots, \ell\mathsf{OTSim}(\mathsf{crs}, D, M, \mathsf{Key}_{M,D[M]}\right).$$

By the sender security for multiple executions, we have that

$$(\mathsf{crs}, \mathsf{Send}(\mathsf{crs}, \mathsf{digest}, \mathsf{Keys})) \overset{c}{\approx} (\mathsf{crs}, \ell\mathsf{OTSim}(\mathsf{crs}, D, \mathsf{Labels})).$$

Security Against Malicious Adversaries. The definition and construction we give is secure against semi-honest adversaries, but it can be upgraded to the malicious setting in a similar way as we will discuss in Chapter 3 for the first application. See Section 3.3 for a detailed discussion.

2.2.2 Updatable Laconic OT

For our applications, we will need a version of laconic OT for which the receiver's short commitment digest to his database can be updated quickly (in time much smaller than the

size of the database) when a bit of the database changes. We call this primitive supporting this functionality updatable laconic OT and define more formally below. At a high level, updatable laconic OT comes with an additional pair of algorithms $\mathsf{SendWrite}$ and $\mathsf{ReceiveWrite}$ which transfer the keys for an updated digest digest^* to the receiver. For convenience, we will define $\mathsf{ReceiveWrite}$ such that it also performs the write in $\hat{\mathsf{D}}$.

Definition 2.2.2 (Updatable Laconic OT)**.** *An updatable laconic OT (updatable ℓOT) scheme consists of algorithms* $\mathsf{crsGen}, \mathsf{Hash}, \mathsf{Send}, \mathsf{Receive}$ *as per Definition 2.2.1 and additionally two algorithms* $\mathsf{SendWrite}$ *and* $\mathsf{ReceiveWrite}$ *with the following syntax.*

- $\mathsf{e_w} \leftarrow \mathsf{SendWrite}\left(\mathsf{crs}, \mathsf{digest}, L, b, \{m_{j,0}, m_{j,1}\}_{j=1}^{|\mathsf{digest}|}\right)$. *It takes as input the common reference string* crs*, a digest* digest*, a location $L \in \mathbb{N}$, a bit $b \in \{0,1\}$ to be written, and* $|\mathsf{digest}|$ *pairs of messages $\{m_{j,0}, m_{j,1}\}_{j=1}^{|\mathsf{digest}|}$, where each $m_{j,c}$ is of length κ. And it outputs a ciphertext* $\mathsf{e_w}$*.*
- $\{m_j\}_{j=1}^{|\mathsf{digest}|} \leftarrow \mathsf{ReceiveWrite}^{\hat{\mathsf{D}}}(\mathsf{crs}, L, b, \mathsf{e_w})$. *This is a RAM algorithm with random read-/write access to $\hat{\mathsf{D}}$. It takes as input the common reference string* crs*, a location L, a bit $b \in \{0,1\}$ and a ciphertext* $\mathsf{e_w}$*. It updates the state $\hat{\mathsf{D}}$ (such that $D[L] = b$) and outputs messages $\{m_j\}_{j=1}^{|\mathsf{digest}|}$.*

We require the following properties on top of properties of a laconic OT scheme.

- ***Correctness with Regard to Writes.*** *For any database D of size at most $M = \mathsf{poly}(\kappa)$ for any polynomial function $\mathsf{poly}(\cdot)$, any memory location $L \in [M]$, any bit $b \in \{0,1\}$, and any messages $\{m_{j,0}, m_{j,1}\}_{j=1}^{|\mathsf{digest}|}$ of length κ, the following holds. Let D^* be identical to D, except that $D^*[L] = b$,*

$$\Pr\left[\begin{array}{l|rl} & \mathsf{crs} & \leftarrow \mathsf{crsGen}(1^\kappa) \\ & (\mathsf{digest}, \hat{\mathsf{D}}) & \leftarrow \mathsf{Hash}(\mathsf{crs}, D) \\ & (\mathsf{digest}^*, \hat{\mathsf{D}}^*) & \leftarrow \mathsf{Hash}(\mathsf{crs}, D^*) \\ & \mathsf{e_w} & \leftarrow \mathsf{SendWrite}\left(\mathsf{crs}, \mathsf{digest}, L, b, \{m_{j,0}, m_{j,1}\}_{j=1}^{|\mathsf{digest}|}\right) \\ m'_j = m_{j,\mathsf{digest}^*_j} & \{m'_j\}_{j=1}^{|\mathsf{digest}|} & \leftarrow \mathsf{ReceiveWrite}^{\hat{\mathsf{D}}}(\mathsf{crs}, L, b, \mathsf{e_w}) \\ \forall j \in [|\mathsf{digest}|] & & \end{array}\right] = 1,$$

  where the probability is taken over the random choices made by crsGen *and* $\mathsf{SendWrite}$*. Furthermore, we require that the execution of* $\mathsf{ReceiveWrite}^{\hat{\mathsf{D}}}$ *above updates $\hat{\mathsf{D}}$ to $\hat{\mathsf{D}}^*$. (Note that* digest *is included in $\hat{\mathsf{D}}$, hence* digest *is also updated to* digest^**.)*

- ***Sender Privacy Against Semi-Honest Receivers with Regard to Writes.*** *There exists a PPT simulator* $\ell\mathsf{OTSimWrite}$ *such that the following holds. For any database D of size at most $M = \mathsf{poly}(\kappa)$ for any polynomial function $\mathsf{poly}(\cdot)$, any memory location $L \in [M]$, any bit $b \in \{0,1\}$, and any messages $\{m_{j,0}, m_{j,1}\}_{j=1}^{|\mathsf{digest}|}$*

of length κ*, let* $\mathsf{crs} \leftarrow \mathsf{crsGen}(1^\kappa)$*,* $(\mathsf{digest}, \hat{\mathsf{D}}) \leftarrow \mathsf{Hash}(\mathsf{crs}, D)$*, and* $(\mathsf{digest}^*, \hat{\mathsf{D}}^*) \leftarrow \mathsf{Hash}(\mathsf{crs}, D^*)$*, where* D^* *is identical to* D *except that* $D^*[L] = b$*. Then it holds that*

$$\begin{pmatrix} \mathsf{crs}, \mathsf{SendWrite}(\mathsf{crs}, \mathsf{digest}, L, b, \{m_{j,0}, m_{j,1}\}_{j=1}^{|\mathsf{digest}|}) \end{pmatrix} \\ \overset{c}{\approx} \begin{pmatrix} \mathsf{crs}, \ell\mathsf{OTSimWrite}\left(\mathsf{crs}, D, L, b, \{m_{j,\mathsf{digest}_j^*}\}_{j\in[|\mathsf{digest}|]}\right) \end{pmatrix}.$$

- ***Efficiency.*** *We require that both* SendWrite *and* ReceiveWrite *run in time* $\mathsf{poly}(\log |D|, \kappa)$.

2.3 Laconic OT with Factor-2 Compression

In this section, based on the DDH assumption we construct a laconic OT scheme for which the hash function Hash compresses a database of length 2κ into a digest of length κ. We refer to this primitive as laconic OT with factor-2 compression. We note that, subsequent to our work, the factor-2 compression construction has been simplified by Döttling and Garg [DG17] (another alternative simplification can be obtained using [AIKW13]). We refer the reader to [DG17] for the simpler construction and preserve the older construction here.

In the following, we give necessary background in Section 2.3.1 and construct two primitives as building blocks: (1) a somewhere statistically binding (SSB) hash function that is friendly to hash proof system (in Section 2.3.2) and (2) a hash proof system that allows for proving knowledge of preimage bits for this SSB hash function (in Section 2.3.3). We present the ℓOT scheme with factor-2 compression in Section 2.3.4 and prove its security in Section 2.3.5.

2.3.1 Background

In this section, we first formalize notations and computational assumptions used in our construction. We then give definitions of somewhere statistically binding (SSB) hash functions [HW15] and hash proof systems [CS98]. For simplicity, we only define SSB hash functions that compress 2κ values in the domain into κ bits. The more general definition works analogously.

2.3.1.1 Notations

We start with some notations. Let $(\mathbb{G}, \cdot)$ be a cyclic group of order p with generator g. Let $\mathbf{M} \in \mathbb{Z}_p^{m\times n}$ be a matrix. We will denote by $\hat{\mathbf{M}} = g^{\mathbf{M}} \in \mathbb{G}^{m\times n}$ the element-wise exponentiation of g with the elements of $\mathbf{M}$. We also define $\hat{\mathbf{L}} = \hat{\mathbf{H}}^{\mathbf{M}} \in \mathbb{G}^{m\times k}$, where $\hat{\mathbf{H}} \in \mathbb{G}^{m\times n}$ and $\mathbf{M} \in \mathbb{Z}_p^{n\times k}$ as follows: Each element $\hat{\mathbf{L}}_{i,j} = \prod_{k=1}^{n} \hat{\mathbf{H}}_{i,k}^{\mathbf{M}_{k,j}}$ (intuitively this operation corresponds to matrix multiplication in the exponent). This is well-defined and efficiently computable.

2.3.1.2 Computational Assumptions

In the following, we define the computational problems on which we base the security of our HPS-friendly SSB hash function.

Definition 2.3.1 (The Decisional Diffie-Hellman (DDH) Problem)**.** *Let $(\mathbb{G}, \cdot)$ be a cyclic group of prime order p and with generator g. Let a, b, c be sampled uniformly at random from $\mathbb{Z}_p$ (i.e., $a, b, c \xleftarrow{\$} \mathbb{Z}_p$). The DDH problem asks to distinguish the distributions (g, g^a, g^b, g^{ab}) and (g, g^a, g^b, g^c).*

Definition 2.3.2 (Matrix Rank Problem)**.** *Let m, n be integers and let $\mathbb{Z}_p^{m\times n;r}$ be the set of all $m \times n$ matrices over $\mathbb{Z}_p$ with rank r. Further, let $1 \leq r_1 < r_2 \leq \min(m, n)$. The goal of the matrix rank problem, denoted as $\mathsf{MatrixRank}(\mathbb{G}, m, n, r_1, r_2)$, is to distinguish the distributions $g^{\mathbf{M}_1}$ and $g^{\mathbf{M}_2}$, where $\mathbf{M}_1 \xleftarrow{\$} \mathbb{Z}_p^{m\times n;r_1}$ and $\mathbf{M}_2 \xleftarrow{\$} \mathbb{Z}_p^{m\times n;r_2}$.*

In a recent result by Villar [Vil12] it was shown that the matrix rank problem can be reduced almost tightly to the DDH problem.

Theorem 2.3.3 ([Vil12] Theorem 1, simplified)**.** *Assume there exists a PPT distinguisher $\mathcal{D}$ that solves $\mathsf{MatrixRank}(\mathbb{G}, m, n, r_1, r_2)$ problem with advantage ε. Then, there exists a PPT distinguisher $\mathcal{D}'$ (running in almost time as $\mathcal{D}$) that solves DDH problem over $\mathbb{G}$ with advantage at least $\frac{\varepsilon}{\lceil \log_2(r_2/r_1)\rceil}$.*

2.3.1.3 Somewhere Statistically Binding (SSB) Hash Functions

Definition 2.3.4 (Somewhere Statistically Binding Hashing)**.** *An SSB hash function SSBH consists of three algorithms crsGen, $\mathsf{bindingCrsGen}$ and Hash with the following syntax.*

- $\mathsf{crs} \leftarrow \mathsf{crsGen}(1^\kappa)$. *It takes the security parameter κ as input and outputs a common reference string crs.*
- $\mathsf{crs} \leftarrow \mathsf{bindingCrsGen}(1^\kappa, i)$. *It takes as input the security parameter κ and an index $i \in [2\kappa]$, and outputs a common reference string crs.*
- $y \leftarrow \mathsf{Hash}(\mathsf{crs}, x)$. *For some domain $\mathfrak{D}$, it takes as input a common reference string crs and a string $x \in \mathfrak{D}^{2\kappa}$, and outputs a string $y \in \{0,1\}^\kappa$.*

We require the following properties of an SSB hash function.

- ***Statistically Binding at Position i.*** *For every $i \in [2\kappa]$ and an overwhelming fraction of crs in the support of $\mathsf{bindingCrsGen}(1^\kappa, i)$ and every $x \in \mathfrak{D}^{2\kappa}$, we have that $(\mathsf{crs}, \mathsf{Hash}(\mathsf{crs}, x))$ uniquely determines x_i. More formally, for all $x' \in \mathfrak{D}^{2\kappa}$ such that $x_i \neq x'_i$ we have that $\mathsf{Hash}(\mathsf{crs}, x') \neq \mathsf{Hash}(\mathsf{crs}, x)$.*
- ***Index Hiding.*** *It holds for all $i \in [2\kappa]$ that $\mathsf{crsGen}(1^\kappa) \overset{c}{\approx} \mathsf{bindingCrsGen}(1^\kappa, i)$, i.e., common reference strings generated by crsGen and $\mathsf{bindingCrsGen}$ are computationally indistinguishable.*

2.3.1.4 Hash Proof Systems

Next, we define hash proof systems [CS98] that are designated verifier proof systems that allow for proving that the given problem instance in some language. We give the formal definition as follows.

Definition 2.3.5 (Hash Proof System)**.** *Let $\mathcal{L}_z \subseteq \mathcal{M}_z$ be an NP-language residing in a universe $\mathcal{M}_z$, both parametrized by some parameter z. Moreover, let $\mathcal{L}_z$ be characterized by an efficiently computable witness-relation $\mathcal{R}$, namely, for all $x \in M_z$ it holds that $x \in \mathcal{L}_z \Leftrightarrow \exists w : \mathcal{R}(x, w) = 1$. A hash proof system HPS for $\mathcal{L}_z$ consists of three algorithms KeyGen, $\mathsf{H}_{\mathsf{public}}$ and $\mathsf{H}_{\mathsf{secret}}$ with the following syntax.*

- $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\kappa, z)$*: Takes as input the security parameter κ and a parameter z, and outputs a public-key and secret key pair $(\mathsf{pk}, \mathsf{sk})$.*
- $y \leftarrow \mathsf{H}_{\mathsf{public}}(\mathsf{pk}, x, w)$*: Takes as input a public key pk, an instance $x \in \mathcal{L}_z$, and a witness w, and outputs a value y.*
- $y \leftarrow \mathsf{H}_{\mathsf{secret}}(\mathsf{sk}, x)$*: Takes as input a secret key sk and an instance $x \in M_z$, and outputs a value y.*

We require the following properties of a hash proof system.

- ***Perfect Completeness.*** *For every z, every $(\mathsf{pk}, \mathsf{sk})$ in the support of $\mathsf{KeyGen}(1^\kappa, z)$, and every $x \in \mathcal{L}_z$ with witness w (i.e., $\mathcal{R}(x, w) = 1$), it holds that*

$$\mathsf{H}_{\mathsf{public}}(\mathsf{pk}, x, w) = \mathsf{H}_{\mathsf{secret}}(\mathsf{sk}, x).$$

- ***Perfect Soundness.*** *For every z and every $x \in M_z \backslash \mathcal{L}_z$, let $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\kappa, z)$, then it holds that*

$$(z, \mathsf{pk}, \mathsf{H}_{\mathsf{secret}}(\mathsf{sk}, x)) \equiv (z, \mathsf{pk}, u),$$

where u is distributed uniformly random in the range of $\mathsf{H}_{\mathsf{secret}}$. Here, $\equiv$ denotes distributional equivalence.

2.3.2 HPS-Friendly SSB Hash Function

In this section, we construct an SSB hash function that supports a hash proof system. In particular, there is a hash proof system that enables proving that a certain bit of the pre-image of a hash-value has a certain fixed value (in our case, either 0 or 1).

Construction. Our construction builds on the scheme of Okamoto et al. [OPWW15]. We will not delve into the details of their scheme and directly jump into our construction.

Let n be an integer such that $n = 2\kappa$, and let $(\mathbb{G}, \cdot)$ be a cyclic group of order p and with generator g. Let $\mathbf{T}_i \in \mathbb{Z}_p^{2\times n}$ be a matrix which is zero everywhere except the i-th column, and the i-th column is equal to $\mathbf{t} = (0,1)^\top$. The three algorithms of the SSB hash function are defined as follows.

- $\mathsf{crsGen}(1^\kappa)$: Pick a uniformly random matrix $\mathbf{H} \xleftarrow{\$} \mathbb{Z}_p^{2\times n}$ and output $\hat{\mathbf{H}} = g^{\mathbf{H}}$.
- $\mathsf{bindingCrsGen}(1^\kappa, i)$: Pick a uniformly random vector $(w_1, w_2)^\top = \mathbf{w} \xleftarrow{\$} \mathbb{Z}_p^2$ with the restriction that $w_1 = 1$, pick a uniformly random vector $\mathbf{a} \xleftarrow{\$} \mathbb{Z}_p^n$ and set $\mathbf{A} \leftarrow \mathbf{w} \cdot \mathbf{a}^\top$. Set $\mathbf{H} \leftarrow \mathbf{T}_i + \mathbf{A}$ and output $\hat{\mathbf{H}} = g^{\mathbf{H}}$.
- $\mathsf{Hash}(\mathsf{crs}, \mathbf{x})$: Parse $\mathbf{x}$ as a vector in $\mathfrak{D}^n$ ($\mathfrak{D} = \mathbb{Z}_p$) and parse $\mathsf{crs} = \hat{\mathbf{H}}$. Compute $\mathbf{y} \in \mathbb{G}^2$ as $\mathbf{y} = \hat{\mathbf{H}}^{\mathbf{x}}$. Parse $\mathbf{y}$ as a binary string and output the result.

Compression. Notice that we can get factor two compression for an input space $\{0,1\}^{2\kappa}$ by restricting the domain to $\mathfrak{D}' = \{0,1\} \subset \mathfrak{D}$. The input length $n = 2\kappa$, where κ is set to be twice the number of bits in the bit representation of a group element in $\mathbb{G}$. In the following we assume that $n = 2\kappa$ and that the bit-representation size of a group element in $\mathbb{G}$ is $\frac{\kappa}{2}$.

Security Proof. We first show that the distributions $\mathsf{crsGen}(1^\kappa)$ and $\mathsf{bindingCrsGen}(1^\kappa, i)$ are computationally indistinguishable for every index $i \in [n]$, given that the DDH problem is computationally hard in the group $\mathbb{G}$.

Lemma 2.3.6 (Index Hiding)**.** *Assume that the* $\mathsf{MatrixRank}(\mathbb{G}, 2, n, 1, 2)$ *problem is hard. Then the distributions* $\mathsf{crsGen}(1^\kappa)$ *and* $\mathsf{bindingCrsGen}(1^\kappa, i)$ *are computationally indistinguishable, for every* $i \in [n]$.

Proof. Assume there exists a PPT distinguisher $\mathcal{D}$ that distinguishes the distributions $\mathsf{crsGen}(1^\kappa)$ and $\mathsf{bindingCrsGen}(1^\kappa, i)$ with non-negligible advantage ε. We will construct a PPT distinguisher $\mathcal{D}'$ that distinguishes $\mathsf{MatrixRank}(\mathbb{G}, 2, n, 1, 2)$ with non-negligible advantage.

The distinguisher $\mathcal{D}'$ does the following on input $\hat{\mathbf{M}} \in \mathbb{G}^{2\times n}$. It computes $\hat{\mathbf{H}} \in \mathbb{G}^{2\times n}$ as element-wise multiplication of $\hat{\mathbf{M}}$ and $g^{\mathbf{T}_i}$ and runs $\mathcal{D}$ on $\hat{\mathbf{H}}$. If $\mathcal{D}$ outputs crsGen, then $\mathcal{D}'$ outputs rank 2, otherwise $\mathcal{D}'$ outputs rank 1.

We now show that $\mathcal{D}'$ also has non-negligible advantage. Write $\mathcal{D}'$'s input as $\hat{\mathbf{M}} = g^{\mathbf{M}}$. If $\mathbf{M}$ is chosen uniformly random with rank 2, then $\mathbf{M}$ is uniform in $\mathbb{Z}_p^{2\times n}$ with overwhelming probability. Hence with overwhelming probability, $\mathbf{M} + \mathbf{T}_i$ is also distributed uniformly random and it follows that $\hat{\mathbf{H}} = g^{\mathbf{M}+\mathbf{T}_i}$ is uniformly random in $\mathbb{G}^{2\times n}$ which is identical to the distribution generated by $\mathsf{crsGen}(1^\kappa)$. On the other hand, if $\mathbf{M}$ is chosen uniformly random with rank 1, then there exists a vector $\mathbf{w} \in \mathbb{Z}_p^2$ such that each column of $\mathbf{M}$ can be written as $a_i \cdot \mathbf{w}$. We can assume that the first element w_1 of $\mathbf{w}$ is 1, since the case

$w_1 = 0$ happens only with probability $1/p = \mathsf{negl}(\kappa)$ and if $w_1 \neq 0$ we can replace all a_i by $a'_i = a_i \cdot w_1$ and replace w_i by $w'_i = \frac{w_i}{w_1}$. Thus, we can write $\mathbf{M}$ as $\mathbf{M} = \mathbf{w} \cdot \mathbf{a}^\top$ and consequently $\hat{\mathbf{H}}$ as $\hat{\mathbf{H}} = g^{\mathbf{w}\cdot\mathbf{a}^\top + \mathbf{T}_i}$. Notice that $\mathbf{a}$ is uniformly distributed, hence $\hat{\mathbf{H}}$ is identical to the distribution generated by $\mathsf{bindingCrsGen}(1^\kappa, i)$. Since $\mathcal{D}$ can distinguish the distributions $\mathsf{crsGen}(1^\kappa)$ and $\mathsf{bindingCrsGen}(1^\kappa, i)$ with non-negligible advantage ε, $\mathcal{D}'$ can distinguish $\mathsf{MatrixRank}(\mathbb{G}, 2, n, 1, 2)$ with advantage $\varepsilon - \mathsf{negl}(\kappa)$, which contradicts the hardness of $\mathsf{MatrixRank}(\mathbb{G}, 2, n, 1, 2)$. □

A corollary of Lemma 2.3.6 is that for all $i, j \in [n]$ the distributions $\mathsf{bindingCrsGen}(1^\kappa, i)$ and $\mathsf{bindingCrsGen}(1^\kappa, j)$ are indistinguishable, stated as follows.

Corollary 2.3.7. *Assume the* $\mathsf{MatrixRank}(\mathbb{G}, 2, n, 1, 2)$ *problem is computationally hard. Then it holds for all* $i, j \in [n]$ *that* $\mathsf{bindingCrsGen}(1^\kappa, i)$ *and* $\mathsf{bindingCrsGen}(1^\kappa, j)$ *are computationally indistinguishable.*

We next show that if the common reference string $\mathsf{crs} = \hat{\mathbf{H}}$ is generated by $\mathsf{bindingCrsGen}(1^\kappa, i)$, then the hash value $\mathsf{Hash}(\mathsf{crs}, \mathbf{x})$ is statistically binded to x_i.

Lemma 2.3.8 (Statistically Binding at Position i)**.** *For every* $i \in [n]$*, every* $\mathbf{x} \in \mathbb{Z}_p^n$*, and all choices of* crs *in the support of* $\mathsf{bindingCrsGen}(1^\kappa, i)$ *we have that for every* $\mathbf{x}' \in \mathbb{Z}_p^n$ *such that* $x'_i \neq x_i$*,* $\mathsf{Hash}(\mathsf{crs}, \mathbf{x}) \neq \mathsf{Hash}(\mathsf{crs}, \mathbf{x}')$*.*

Proof. We first write crs as $\hat{\mathbf{H}} = g^{\mathbf{H}} = g^{\mathbf{w}\cdot\mathbf{a}^\top + \mathbf{T}_i}$ and $\mathsf{Hash}(\mathsf{crs}, \mathbf{x})$ as $\mathsf{Hash}(\hat{\mathbf{H}}, \mathbf{x}) = g^{\mathbf{y}} = g^{\mathbf{H}\cdot\mathbf{x}}$. Thus, by taking the discrete logarithm with basis g our task is to demonstrate that there exists a unique x_i from $\mathbf{H} = \mathbf{w} \cdot \mathbf{a}^\top + \mathbf{T}_i$ and $\mathbf{y} = \mathbf{H} \cdot \mathbf{x}$. Observe that

$$\begin{aligned}\mathbf{y} = \mathbf{H} \cdot \mathbf{x} &= (\mathbf{w} \cdot \mathbf{a}^\top + \mathbf{T}_i) \cdot \mathbf{x} = \mathbf{w} \cdot \langle \mathbf{a}, \mathbf{x} \rangle + \mathbf{T}_i \cdot \mathbf{x} \\ &= \begin{pmatrix} 1 \\ w_2 \end{pmatrix} \cdot \langle \mathbf{a}, \mathbf{x} \rangle + \begin{pmatrix} 0 \\ 1 \end{pmatrix} \cdot x_i,\end{aligned}$$

where $\langle \mathbf{a}, \mathbf{x} \rangle$ is the inner product of $\mathbf{a}$ and $\mathbf{x}$. If $\mathbf{a} \neq \mathbf{0}$, then we can use any non-zero element of $\mathbf{a}$ to compute w_2 from $\mathbf{H}$, and recover x_i by computing $x_i = y_2 - w_2 \cdot y_1$; otherwise $\mathbf{a} = \mathbf{0}$, so $x_i = y_2$. □

2.3.3 A Hash Proof System for Knowledge of Preimage Bits

In this section, we give our desired hash proof systems. In particular, we need a hash proof system for membership in a subspace of a vector space.

Construction. Fix a matrix $\hat{\mathbf{H}} \in \mathbb{G}^{2\times n}$ and an index $i \in [n]$. We will construct a hash proof system $\mathsf{HPS} = (\mathsf{KeyGen}, \mathsf{H}_{\mathsf{public}}, \mathsf{H}_{\mathsf{secret}})$ for the following language $\mathcal{L}_{\hat{\mathbf{H}},i}$:

$$\mathcal{L}_{\hat{\mathbf{H}},i} = \{(\hat{\mathbf{y}}, b) \in \mathbb{G}^2 \times \{0,1\} \mid \exists \mathbf{x} \in \mathbb{Z}_p^n \text{ s.t. } \hat{\mathbf{y}} = \hat{\mathbf{H}}^{\mathbf{x}} \text{ and } x_i = b\}.$$

Note that in our hash proof system we only enforce that a single specified bit is b, where $b \in \{0,1\}$. However, our hash proof system does not place any requirement on the value used at any of the other locations. In fact the values used at the other locations may actually be from the full domain $\mathfrak{D}$ (i.e., $\mathbb{Z}_p$). Observe that the formal definition of the language $\mathcal{L}_{\hat{\mathbf{H}},i}$ above incorporates this difference in how the honest computation of the hash function is performed and what the hash proof system is supposed to prove.

For ease of exposition, it will be convenient to work with a matrix $\hat{\mathbf{H}}' \in \mathbb{G}_p^{3\times n}$:

$$\hat{\mathbf{H}}' = \begin{pmatrix} \hat{\mathbf{H}} \\ g^{\mathbf{e}_i^\top} \end{pmatrix},$$

where $\mathbf{e}_i \in \mathbb{Z}_p^n$ is the i-th unit vector, with all elements equal to zero except the i^{th} one which is equal to one.

- $\mathsf{KeyGen}(1^\kappa, (\hat{\mathbf{H}}, i))$: Choose $\mathbf{r} \xleftarrow{\$} \mathbb{Z}_p^3$ uniformly at random. Compute $\hat{\mathbf{h}} = \left((\hat{\mathbf{H}}')^\top\right)^{\mathbf{r}}$. Set $\mathsf{pk} = \hat{\mathbf{h}}$ and $\mathsf{sk} = \mathbf{r}$. Output $(\mathsf{pk}, \mathsf{sk})$.
- $\mathsf{H}_{\mathsf{public}}(\mathsf{pk}, (\hat{\mathbf{y}}, b), \mathbf{x})$: Parse pk as $\hat{\mathbf{h}}$. Compute $\hat{z} = (\hat{\mathbf{h}}^\top)^{\mathbf{x}}$ and output $\hat{z}$.
- $\mathsf{H}_{\mathsf{secret}}(\mathsf{sk}, (\hat{\mathbf{y}}, b))$: Parse sk as $\mathbf{r}$ and set $\hat{\mathbf{y}}' = \begin{pmatrix} \hat{\mathbf{y}} \\ g^b \end{pmatrix}$. Compute $\hat{z} = ((\hat{\mathbf{y}}')^\top)^{\mathbf{r}}$ and output $\hat{z}$.

Security Proof. In our proof we need the following technical lemma.

Lemma 2.3.9. *Let $\mathbf{M} \in \mathbb{Z}_p^{m\times n}$ be a matrix. Let $\mathsf{colsp}(\mathbf{M}) = \{\mathbf{M}\cdot\mathbf{x} \mid \mathbf{x} \in \mathbb{Z}_p^n\}$ be its column space, and $\mathsf{rowsp}(\mathbf{M}) = \{\mathbf{x}^\top \cdot \mathbf{M} \mid \mathbf{x} \in \mathbb{Z}_p^m\}$ be its row space. Assume that $\mathbf{y} \in \mathbb{Z}_p^m$ and $\mathbf{y} \notin \mathsf{colsp}(\mathbf{M})$. Let $\mathbf{r} \xleftarrow{\$} \mathbb{Z}_p^m$ be chosen uniformly at random. Then it holds that*

$$(\mathbf{M}, \mathbf{y}, \mathbf{r}^\top\mathbf{M}, \mathbf{r}^\top\mathbf{y}) \equiv (\mathbf{M}, \mathbf{y}, \mathbf{r}^\top\mathbf{M}, u),$$

where $u \xleftarrow{\$} \mathbb{Z}_p$ is distributed uniformly and independently of $\mathbf{r}$. Here, $\equiv$ denotes distributional equivalence.

Proof. For any $\mathbf{t} \in \mathsf{rowsp}(\mathbf{M})$ and $s \in \mathbb{Z}_p$, consider following linear equation system

$$\begin{cases} \mathbf{r}^\top\mathbf{M} = \mathbf{t} \\ \mathbf{r}^\top\mathbf{y} = s \end{cases}.$$

Let $\mathcal{N}$ be the left null space of $\mathbf{M}$. We know that $\mathbf{y} \notin \mathsf{colsp}(\mathbf{M})$, hence $\mathbf{M}$ has rank $\leq m-1$, therefore $\mathcal{N}$ has dimension ≥ 1. Let $\mathbf{r}_0$ be an arbitrary solution for $\mathbf{r}^\top\mathbf{M} = \mathbf{t}$, and let $\mathbf{n}$ be a vector in $\mathcal{N}$ such that $\mathbf{n}^\top\mathbf{y} \neq \mathbf{0}$ (there must be such a vector since $\mathbf{y} \notin \mathsf{colsp}(\mathbf{M})$). Then there exists a solution $\mathbf{r}$ for the above linear equation system, that is,

$$\mathbf{r} = \mathbf{r}_0 + (\mathbf{n}^\top\mathbf{y})^{-1} \cdot (s - \mathbf{r}_0^\top\mathbf{y}) \cdot \mathbf{n},$$

where $(\mathbf{n}^\top \mathbf{y})^{-1}$ is the multiplicative inverse of $\mathbf{n}^\top \mathbf{y}$ in $\mathbb{Z}_p$. Then two cases arise: (i) column vectors of $(\mathbf{M}\ \mathbf{y})$ are full-rank, or (ii) not. In this first case, there is a unique solution for $\mathbf{r}$. In the second case the solution space has the same size as the left null space of $(\mathbf{M}\ \mathbf{y})$. Therefore, in both cases, the number of solutions for $\mathbf{r}$ is the same for every $(\mathbf{t}, s)$ pair.

As $\mathbf{r}$ is chosen uniformly at random, all pairs $(\mathbf{t}, s) \in \mathsf{rowsp}(\mathbf{M}) \times \mathbb{Z}_p$ have the same probability of occurrence and the claim follows. □

Now we are ready to prove security.

Theorem 2.3.10. *For every matrix $\hat{\mathbf{H}} \in \mathbb{G}^{2\times n}$ and every $i \in [n]$, HPS is a hash proof system for the language $\mathcal{L}_{\hat{\mathbf{H}},i}$.*

Proof. Let $\hat{\mathbf{H}} = g^{\mathbf{H}}$, $\mathbf{r} = (\mathbf{r}^*, r_3)$ where $\mathbf{r}^* \in \mathbb{Z}_p^2$. Let $\mathbf{y}' := \log_g \hat{\mathbf{y}}'$, $\mathbf{y} := \log_g \hat{\mathbf{y}}$, $\mathbf{H}' := \log_g \hat{\mathbf{H}}'$, $\mathbf{h} := \log_g \hat{\mathbf{h}}$.

For perfect correctness, we need to show that for every $i \in [n]$, every $\hat{\mathbf{H}} \in \mathbb{G}^{2\times n}$, and every $(\mathsf{pk}, \mathsf{sk})$ in the support of $\mathsf{KeyGen}(1^\kappa, (\hat{\mathbf{H}}', i))$, if $(\hat{\mathbf{y}}, b) \in \mathcal{L}_{\hat{\mathbf{H}},i}$ and $\mathbf{x}$ is a witness for membership (i.e., $\hat{\mathbf{y}} = \hat{\mathbf{H}}^{\mathbf{x}}$ and $x_i = b$), then it holds that $\mathsf{H}_{\mathsf{public}}(\mathsf{pk}, (\hat{\mathbf{y}}, b), \mathbf{x}) = \mathsf{H}_{\mathsf{secret}}(\mathsf{sk}, (\hat{\mathbf{y}}, b))$.

To simplify the argument, we again consider the statement under the discrete logarithm with basis g. Then it holds that

$$\begin{aligned}
&\log_g \left(\mathsf{H}_{\mathsf{secret}}(\mathsf{sk}, (\hat{\mathbf{y}}, b))\right) \\
=&\log_g \left(\left((\hat{\mathbf{y}}')^\top\right)^{\mathbf{r}}\right) = \langle \mathbf{y}', \mathbf{r}\rangle = \langle \mathbf{y}, \mathbf{r}^*\rangle + b \cdot r_3 \\
=&\langle \mathbf{H}\cdot\mathbf{x}, \mathbf{r}^*\rangle + x_i \cdot r_3 = \langle \mathbf{H}'\mathbf{x}, \mathbf{r}\rangle = \langle (\mathbf{H}')^\top \mathbf{r}, \mathbf{x}\rangle \\
=&\langle \mathbf{h}, \mathbf{x}\rangle = \log_g \left((\hat{\mathbf{h}}^\top)^{\mathbf{x}}\right) \\
=&\log_g \left(\mathsf{H}_{\mathsf{public}}(\mathsf{pk}, (\hat{\mathbf{y}}, b), \mathbf{x})\right).
\end{aligned}$$

For perfect soundness, let $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\kappa, (\hat{\mathbf{H}}', i))$. We will show that if $(\hat{\mathbf{y}}, b) \notin \mathcal{L}_{\hat{\mathbf{H}},i}$, then $\mathsf{H}_{\mathsf{secret}}(\mathsf{sk}, (\hat{\mathbf{y}}, b))$ is distributed uniformly random in the range of $\mathsf{H}_{\mathsf{secret}}$, even given $\hat{\mathbf{H}}$, i, and pk. Again under the discrete logarithm, this is equivalent to showing that $\langle \mathbf{y}', \mathbf{r}\rangle$ is distributed uniformly random given $\mathbf{H}'$ and $\mathbf{h} = (\mathbf{H}')^\top \mathbf{r}$.

Note that we can re-write the language $\mathcal{L}_{\hat{\mathbf{H}},i} = \{(\hat{\mathbf{y}}, b) \in \mathbb{G}^2 \times \mathbb{Z}_p \mid \exists \mathbf{x} \in \mathbb{Z}_p^n \text{ s.t. } \mathbf{H}'\mathbf{x} = \mathbf{y}'\}$. It follows that if $(\hat{\mathbf{y}}, b) \notin \mathcal{L}_{\hat{\mathbf{H}},i}$, then $\mathbf{y}' \notin \mathsf{span}(\mathbf{H}')$. Now it follows directly from Lemma 2.3.9 that

$$\mathbf{r}^\top \mathbf{y}' \equiv u$$

given $\mathbf{H}'$ and $\mathbf{r}^\top \mathbf{H}'$, where u is distributed uniformly random. This concludes the proof. □

Remark 2.3.11. *While proving the security of our applications based on the above hash-proof system, we would generate $\hat{\mathbf{H}}$ to be the output of $\mathsf{bindingCrsGen}(1^\kappa, i)$ and use the property that if $(\hat{\mathbf{y}}, b) \in \mathcal{L}_{\hat{\mathbf{H}},i}$, then $(\hat{\mathbf{y}}, (1-b)) \notin \mathcal{L}_{\hat{\mathbf{H}},i}$. This follows directly from Lemma 2.3.8 (that is, $\hat{\mathbf{H}}$ and $\hat{\mathbf{y}}$ uniquely fixes x_i).*

2.3.4 Construction of Laconic OT with Factor-2 Compression

We are now ready to put the pieces together and provide our ℓOT scheme with factor-2 compression.

Let $\mathsf{SSBH} = (\mathsf{SSBH.crsGen}, \mathsf{SSBH.bindingCrsGen}, \mathsf{SSBH.Hash})$ be the HPS-friendly SSB hash function constructed in Section 2.3.2 with domain $\mathfrak{D} = \mathbb{Z}_p$. Notice that we achieve factor-2 compression (namely, compressing 2κ bits into κ bits) by restricting the domain from $\mathfrak{D}^n$ to $\{0,1\}^n$ in our laconic OT scheme. Also, abstractly let the associated hash proof system be $\mathsf{HPS} = (\mathsf{HPS.KeyGen}, \mathsf{HPS.H_{public}}, \mathsf{HPS.H_{secret}})$ for the language

$$\mathcal{L}_{\mathsf{crs},i} = \{(\mathsf{digest}, b) \in \{0,1\}^\kappa \times \{0,1\} \mid \exists D \in \mathfrak{D}^{2\kappa} : \mathsf{SSBH.Hash}(\mathsf{crs}, D) = \mathsf{digest} \text{ and } D[i] = b\}.$$

Recall that the bit-representation size of a group element of $\mathbb{G}$ is $\frac{\kappa}{2}$, hence the language defined above is the same as the one defined in Section 2.3.3.

Now we construct the laconic OT scheme $\ell OT = (\mathsf{crsGen}, \mathsf{Hash}, \mathsf{Send}, \mathsf{Receive})$ as follows.

- $\mathsf{crsGen}(1^\kappa)$: Compute $\mathsf{crs} \leftarrow \mathsf{SSBH.crsGen}(1^\kappa)$ and output crs.
- $\mathsf{Hash}(\mathsf{crs}, D \in \{0,1\}^{2\kappa})$:
  $\mathsf{digest} \leftarrow \mathsf{SSBH.Hash}(\mathsf{crs}, D)$
  $\hat{\mathsf{D}} \leftarrow (D, \mathsf{digest})$
  Output $(\mathsf{digest}, \hat{\mathsf{D}})$
- $\mathsf{Send}(\mathsf{crs}, \mathsf{digest}, L, m_0, m_1)$:
  Let HPS be the hash-proof system for the language $\mathcal{L}_{\mathsf{crs},L}$
  $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{HPS.KeyGen}(1^\kappa, (\mathsf{crs}, L))$
  $c_0 \leftarrow m_0 \oplus \mathsf{HPS.H_{secret}}(\mathsf{sk}, (\mathsf{digest}, 0))$
  $c_1 \leftarrow m_1 \oplus \mathsf{HPS.H_{secret}}(\mathsf{sk}, (\mathsf{digest}, 1))$
  Output $\mathsf{e} = (\mathsf{pk}, c_0, c_1)$
- $\mathsf{Receive}^{\hat{\mathsf{D}}}(\mathsf{crs}, \mathsf{e}, L)$:
  Parse $\mathsf{e} = (\mathsf{pk}, c_0, c_1)$
  Parse $\hat{\mathsf{D}} = (D, \mathsf{digest})$, and set $b \leftarrow D[L]$.
  $m \leftarrow c_b \oplus \mathsf{HPS.H_{public}}(\mathsf{pk}, (\mathsf{digest}, b), D)$
  Output m

2.3.5 Security Proof

We now prove that ℓOT is a laconic OT protocol with factor-2 compression, i.e., it has compression factor 2, and satisfies the correctness and sender privacy requirements. First notice that $\mathsf{SSBH.Hash}$ is factor-2 compressing, so Hash also has compression factor 2. We next argue correctness and sender privacy in Lemmas 2.3.12 and 2.3.13, respectively.

Lemma 2.3.12. *Given that* HPS *satisfies the correctness property, the ℓOT scheme also satisfies the correctness property.*

Proof. Fix a common reference string crs in the support of $\mathsf{crsGen}(1^\kappa)$, a database string $D \in \{0,1\}^{2\kappa}$ and an index $L \in [2\kappa]$. For any crs, D, L such that $D[L] = b$, let $\mathsf{digest} = \mathsf{Hash}(\mathsf{crs}, D)$. Then it clearly holds that $(\mathsf{digest}, b) \in \mathcal{L}_{\mathsf{crs},L}$. Thus, by the correctness property of the hash proof system HPS it holds that

$$\mathsf{HPS.H}_{\mathsf{secret}}(\mathsf{sk}, (\mathsf{digest}, b)) = \mathsf{HPS.H}_{\mathsf{public}}(\mathsf{pk}, (\mathsf{digest}, b), D).$$

By the construction of $\mathsf{Send}(\mathsf{crs}, \mathsf{digest}, L, m_0, m_1)$, $c_b = m_b \oplus \mathsf{HPS.H}_{\mathsf{secret}}(\mathsf{sk}, (\mathsf{digest}, b))$. Hence the output m of $\mathsf{Receive}^{\hat{\mathsf{D}}}(\mathsf{crs}, \mathsf{e}, L)$ is

$$\begin{aligned} m =& c_b \oplus \mathsf{HPS.H}_{\mathsf{public}}(\mathsf{pk}, (\mathsf{digest}, b), D) \\ =& m_b \oplus \mathsf{HPS.H}_{\mathsf{secret}}(\mathsf{sk}, (\mathsf{digest}, b)) \oplus \mathsf{HPS.H}_{\mathsf{public}}(\mathsf{pk}, (\mathsf{digest}, b), D) \\ =& m_b. \end{aligned}$$

□

Lemma 2.3.13. *Given that* SSBH *is index-hiding and has the statistically binding property and that* HPS *is sound, then the ℓOT scheme satisfies sender privacy against semi-honest receiver.*

Proof. We first construct the simulator $\ell\mathsf{OTSim}$.

$\ell\mathsf{OTSim}(\mathsf{crs}, D, L, m_{D[L]})$:
$\mathsf{digest} \leftarrow \mathsf{SSBH.Hash}(\mathsf{crs}, D)$
Let HPS be the hash-proof system for the language $\mathcal{L}_{\mathsf{crs},L}$
$(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{HPS.KeyGen}(1^\kappa, (\mathsf{crs}, L))$
$c_0 \leftarrow m_{D[L]} \oplus \mathsf{HPS.H}_{\mathsf{secret}}(\mathsf{sk}, (\mathsf{digest}, 0))$
$c_1 \leftarrow m_{D[L]} \oplus \mathsf{HPS.H}_{\mathsf{secret}}(\mathsf{sk}, (\mathsf{digest}, 1))$
Output (pk, c_0, c_1)

For any database D of size at most $M = \mathsf{poly}(\kappa)$ for any polynomial function $\mathsf{poly}(\cdot)$, any memory location $L \in [M]$, and any pair of messages $(m_0, m_1) \in \{0,1\}^\kappa \times \{0,1\}^\kappa$, let $\mathsf{crs} \leftarrow \mathsf{crsGen}(1^\kappa)$ and $\mathsf{digest} \leftarrow \mathsf{Hash}(\mathsf{crs}, D)$. Then we will prove that the two distributions $(\mathsf{crs}, \mathsf{Send}(\mathsf{crs}, \mathsf{digest}, L, m_0, m_1))$ and $(\mathsf{crs}, \ell\mathsf{OTSim}(\mathsf{crs}, D, L, m_{D[L]}))$ are computationally indistinguishable. Consider the following hybrids.

- Hybrid 0: This is the real experiment, namely $(\mathsf{crs}, \mathsf{Send}(\mathsf{crs}, \mathsf{digest}, L, m_0, m_1))$.
- Hybrid 1: Same as hybrid 0, except that crs is computed by $\mathsf{crs} \leftarrow \mathsf{SSBH.bindingCrsGen}(1^\kappa, L)$.
- Hybrid 2: Same as hybrid 1, except that $c_{1-D[L]}$ is computed by $c_{1-D[L]} \leftarrow m_{D[L]} \oplus \mathsf{HPS.H}_{\mathsf{secret}}(\mathsf{sk}, (\mathsf{digest}, 1 - D[L]))$. That is, both c_0 and c_1 encrypt the same message $m_{D[L]}$.

- Hybrid 3: Same as hybrid 2, except that crs is computed by $\mathsf{crs} \leftarrow \mathsf{SSBH.crsGen}(1^\kappa)$. This is the simulated experiment, namely $(\mathsf{crs}, \ell\mathsf{OTSim}(\mathsf{crs}, D, L, m_{D[L]}))$.

Indistinguishability of hybrid 0 and hybrid 1 follows directly from Lemma 2.3.6, as we replace the distribution of crs from $\mathsf{SSBH.crsGen}(1^\kappa)$ to $\mathsf{SSBH.bindingCrsGen}(1^\kappa, L)$. Indistinguishability of hybrids 2 and 3 also follows from Lemma 2.3.6, as we replace the distribution of crs from $\mathsf{SSBH.bindingCrsGen}(1^\kappa, L)$ back to $\mathsf{SSBH.crsGen}(1^\kappa)$.

We now show that hybrids 1 and 2 are identically distributed. Since crs is in the support of $\mathsf{SSBH.bindingCrsGen}(1^\kappa, i)$ and $\mathsf{digest} = \mathsf{SSBH.Hash}(\mathsf{crs}, D)$, by Lemma 2.3.8 it holds that $(\mathsf{digest}, 1 - D[L]) \notin \mathcal{L}_{\mathsf{crs},L}$. By the soundness property of the hash-proof system HPS, it holds that

$$(\mathsf{crs}, L, \mathsf{pk}, \mathsf{HPS.H}_{\mathsf{secret}}(\mathsf{sk}, (\mathsf{digest}, 1 - D[L]))) \equiv (\mathsf{crs}, L, \mathsf{pk}, u),$$

for a uniformly random u. Furthermore, $c_{D[L]}$ can be computed by $m_{D[L]} \oplus \mathsf{HPS.H}_{\mathsf{public}}(\mathsf{pk}, (\mathsf{digest}, D[L]), D)$. Hence

$$\begin{aligned}
&(\mathsf{crs}, L, \mathsf{pk}, m_{D[L]} \oplus \mathsf{HPS.H}_{\mathsf{secret}}(\mathsf{sk}, (\mathsf{digest}, 1 - D[L])), c_{D[L]}) \\
\equiv &(\mathsf{crs}, L, \mathsf{pk}, u, c_{D[L]}) \\
\equiv &(\mathsf{crs}, L, \mathsf{pk}, m_{1-D[L]} \oplus \mathsf{HPS.H}_{\mathsf{secret}}(\mathsf{sk}, (\mathsf{digest}, 1 - D[L])), c_{D[L]}).
\end{aligned}$$

This concludes the proof. □

2.4 Laconic OT with Arbitrary Compression

In this section, we construct an updatable laconic OT that supports a hash function that allows for compression from an input (database) of size an arbitrary polynomial in κ to κ bits. As every updatable laconic OT protocol is also a (standard) laconic OT protocol, we will only construct the former. Our main technique in this construction, is the use of garbled circuits to bootstrap a laconic OT with factor-2 compression into one with an arbitrary compression factor.

Below in Section 2.4.1 we describe some background on the primitives needed for realizing our laconic OT construction. Then we give the construction of laconic OT along with its correctness and security proofs in Sections 2.4.2 and 2.4.3, respectively.

2.4.1 Background

In this section we recall the needed background of garbled circuits and Merkle trees.

2.4.1.1 Garbled Circuits

Garbled circuits were first introduced by Yao [Yao82; Yao86] (see Lindell and Pinkas [LP09] and Bellare et al. [BHR12] for a detailed proof and further discussion). A circuit garbling

scheme GC is a tuple of PPT algorithms $(\mathsf{GCircuit}, \mathsf{GEval})$. Very roughly $\mathsf{GCircuit}$ is the circuit garbling procedure and GEval the corresponding evaluation procedure. Looking ahead, each individual wire w of the circuit being garbled will be associated with two labels, namely $\mathsf{lab}_{w,0}, \mathsf{lab}_{w,1}$.

- $\widetilde{\mathsf{C}} \leftarrow \mathsf{GCircuit}\left(1^\kappa, \mathsf{C}, \{\mathsf{lab}_{w,b}\}_{w\in\mathsf{inp}(\mathsf{C}),b\in\{0,1\}}\right)$: $\mathsf{GCircuit}$ takes as input a security parameter κ, a circuit C, and a set of labels $\mathsf{lab}_{w,b}$ for all the input wires $w \in \mathsf{inp}(\mathsf{C})$ and $b \in \{0,1\}$. This procedure outputs a *garbled circuit* $\widetilde{\mathsf{C}}$.
- $y \leftarrow \mathsf{GEval}\left(\widetilde{\mathsf{C}}, \{\mathsf{lab}_{w,x_w}\}_{w\in\mathsf{inp}(\mathsf{C})}\right)$: Given a garbled circuit $\widetilde{\mathsf{C}}$ and a garbled input represented as a sequence of input labels $\{\mathsf{lab}_{w,x_w}\}_{w\in\mathsf{inp}(\mathsf{C})}$, GEval outputs y.

Correctness. For correctness, we require that for any circuit C and input $x \in \{0,1\}^m$ (here m is the input length to C) we have that:

$$\Pr\left[\mathsf{C}(x) = \mathsf{GEval}\left(\widetilde{\mathsf{C}}, \{\mathsf{lab}_{w,x_w}\}_{w\in\mathsf{inp}(\mathsf{C})}\right)\right] = 1$$

where $\widetilde{\mathsf{C}} \leftarrow \mathsf{GCircuit}\left(1^\kappa, \mathsf{C}, \{\mathsf{lab}_{w,b}\}_{w\in\mathsf{inp}(\mathsf{C}),b\in\{0,1\}}\right)$.

Security. For security, we require that there is a PPT simulator $\mathsf{CircSim}$ such that for any C, x, and uniformly random keys $\{\mathsf{lab}_{w,b}\}_{w\in\mathsf{inp}(\mathsf{C}),b\in\{0,1\}}$, we have that

$$\left(\widetilde{\mathsf{C}}, \{\mathsf{lab}_{w,x_w}\}_{w\in\mathsf{inp}(\mathsf{C})}\right) \overset{c}{\approx} \mathsf{CircSim}\left(1^\kappa, \mathsf{C}, y\right)$$

where $\widetilde{\mathsf{C}} \leftarrow \mathsf{GCircuit}\left(1^\kappa, \mathsf{C}, \{\mathsf{lab}_{w,b}\}_{w\in\mathsf{inp}(\mathsf{C}),b\in\{0,1\}}\right)$ and $y = \mathsf{C}(x)$.

Terminology of Keys and Labels. We use the notation Keys to refer to both the secret values sampled for wires and the notation Labels to refer to exactly one of them. In other words, generation of garbled circuit involves Keys while computation itself depends just on Labels. Let $\mathsf{Keys} = ((\mathsf{lab}_{1,0}, \mathsf{lab}_{1,1}), \ldots, (\mathsf{lab}_{n,0}, \mathsf{lab}_{n,1}))$ be a list of n key-pairs, we denote Keys_x for a string $x \in \{0,1\}^n$ to be a list of labels $(\mathsf{lab}_{1,x_1}, \ldots, \mathsf{lab}_{n,x_n})$.

2.4.1.2 Merkle Tree

In this section we briefly review Merkle trees. A Merkle tree is a hash based data structure that generically extend the domain of a hash function. The following description will be tailored to the hash function of the laconic OT scheme that we will present in Section 2.4.2. Given a two-to-one hash function $\mathsf{Hash} : \{0,1\}^{2\kappa} \to \{0,1\}^\kappa$, we can use a Merkle tree to construct a hash function that compresses a database of an arbitrary (a priori unbounded polynomial in κ) size to a κ-bit string. Now we briefly illustrate how to compress a database $D \in \{0,1\}^M$ (assume for ease of exposition that $M = 2^d \cdot \kappa$). First, we partition D into

strings of length 2κ; we call each string a *leaf*. Then we use Hash to compress each leaf into a new string of length κ; we call each string a *node*. Next, we bundle the new nodes in pairs of two and call these pairs *siblings*, i.e., each pair of siblings is a string of length 2κ. We then use Hash again to compress each pair of siblings into a new node of size κ. We continue the process till a single node of size κ is obtained. This process forms a binary tree structure, which we refer to as a Merkle tree. Looking ahead, the hash function of the laconic OT scheme has output $(\hat{\mathsf{D}}, \mathsf{digest})$, where $\hat{\mathsf{D}}$ is the entire Merkle tree, and digest is the root of the tree.

A Merkle tree has the following property. In order to verify that a database D with hash root digest has a certain value b at a location L (namely, $D[L] = b$), there is no need to provide the entire Merkle tree. Instead, it is sufficient to provide a path of siblings from the Merkle tree root to the leaf that contains location L. It can then be easily verified if the hash values from the leaf to the root are correct.

Moreover, a Merkle tree can be updated in the same fashion when the value at a certain location of the database is updated. Instead of recomputing the entire tree, we only need to recompute the nodes on the path from the updated leaf to the root. This can be done given the path of siblings from the root to the leaf.

2.4.2 Construction of Updatable Laconic OT with Arbitrary Compression

We now present our construction to bootstrap an ℓOT scheme with factor-2 compression into an updatable ℓOT scheme with an arbitrary compression factor, which can compress a database of an arbitrary (a priori unbounded polynomial in κ) size.

Overview. We first give an overview of the construction. Consider a database $D \in \{0,1\}^M$ such that $M = 2^d \cdot \kappa$. Given a laconic OT scheme with factor-2 compression (denoted as ℓOT_{const}), we will first use a Merkle tree to obtain a hash function with arbitrary (polynomial) compression factor. As described in Section 2.4.1.2, the Hash function of the updatable ℓOT scheme will have an output $(\hat{\mathsf{D}}, \mathsf{digest})$, where $\hat{\mathsf{D}}$ is the entire Merkle tree, and digest is the root of the tree.

In the Send algorithm, suppose we want to send a message depending on a bit $D[L]$, we will follow the natural approach of traversing the Merkle tree layer by layer until reaching the leaf containing L. In particular, L can be represented as $L = (b_1, \ldots, b_{d-1}, t)$, where $b_1, \ldots, b_{d-1}$ are bits representing the path from the root to the leaf containing location L, and $t \in [2\kappa]$ is the position within the leaf. The Send algorithm first takes as input the root digest of the Merkle tree, and it will generate a chain of garbled circuits, which would enable the receiver to traverse the Merkle tree from the root to the leaf. And upon reaching the leaf, the receiver will be able to evaluate the last garbled circuit and retrieve the message corresponding to the t-th bit of the leaf.

We briefly explain the chain of garbled circuits as follows. The chain consists of $d-1$ traversing circuits along with a reading circuit. Every traversing circuit takes as input a pair of siblings $\mathsf{sbl} = (\mathsf{sbl}_0, \mathsf{sbl}_1)$ at a certain layer of the Merkle tree, chooses sbl_b which is the node in the path from root to leaf, and generates a laconic OT ciphertext (using $\ell OT_{\mathsf{const}}.\mathsf{Send}$) which encrypts the input keys of the next traversing garbled circuit and uses sbl_b as the hash value. Looking ahead, when the receiver evaluates the traversing circuit and obtains the laconic OT ciphertext, he can then use the siblings at the next layer to decrypt the ciphertext (by $\ell OT_{\mathsf{const}}.\mathsf{Receive}$) and obtain the corresponding input labels for the next traversing garbled circuit. Using the chain of traversing garbled circuits the receiver can therefore traverse from the first layer to the leaf of the Merkle tree. Furthermore, the correct keys for the first traversing circuit are sent via the ℓOT_{const} with digest (i.e., root of the tree) as the hash value.

Finally, the last traversing circuit will transfer keys for the last reading circuit to the receiver in a similar fashion as above. The reading circuit takes the leaf as input and outputs $m_{\mathsf{leaf}[t]}$, i.e., the message corresponding to the t-th bit of the leaf. Hence, when evaluating the reading circuit, the receiver can obtain the message $m_{\mathsf{leaf}[t]}$.

$\mathsf{SendWrite}$ and $\mathsf{ReceiveWrite}$ are similar as Send and $\mathsf{Receive}$, except that (a) $\mathsf{ReceiveWrite}$ updates the Merkle tree from the leaf to the root, and (b) the last writing circuit recomputes the root of the Merkle tree and outputs messages corresponding to the new root. To enable (b), the writing circuit will take as input the whole path of siblings from the root to the leaf. The input keys for the writing circuit corresponding to the siblings at the $(i+1)$-th layer are transferred via the i-th traversing circuit. That is, the i-th traversing circuit transfers the keys for the $(i+1)$-th transferring circuit as well as partial keys for the writing circuit. In the actual construction, both the reading circuit and writing circuit take as input the entire path of siblings (for the purpose of symmetry).

Construction. Let $\mathsf{GC} = (\mathsf{GCircuit}, \mathsf{GEval})$ be a circuit garbling scheme. Let $\ell OT_{\mathsf{const}} = (\ell OT_{\mathsf{const}}.\mathsf{crsGen}, \ell OT_{\mathsf{const}}.\mathsf{Hash}, \ell OT_{\mathsf{const}}.\mathsf{Send}, \ell OT_{\mathsf{const}}.\mathsf{Receive})$ be a laconic OT protocol with factor-2 compression. Without loss of generality, let $D \in \{0,1\}^M$ be a database such that $|M| = 2^d \cdot \kappa$. A location $L \in [M]$ can be represented as $(b_1, b_2, \ldots, b_{d-1}, t) \in \{0,1\}^{d-1} \times [2\kappa]$, where the bits b_i's define the path from the root to a leaf in the Merkle tree, and $t \in [2\kappa]$ defines a position in that leaf.

Before delving into the construction, we first describe three gadget circuits: the traversing circuit $\mathsf{C}^{\mathsf{trav}}$, the reading circuit $\mathsf{C}^{\mathsf{read}}$, and the writing circuit $\mathsf{C}^{\mathsf{write}}$. These circuits are defined formally in Figures 2.2, 2.3, and 2.4, respectively.

The traversing circuit has hardwired inside it a common reference string crs, a bit b and two vectors of input keys $\mathsf{Keys}, \widetilde{\mathsf{Keys}}$, each containing 2κ key-pairs (a key-pair is a pair of κ-bit strings). It takes as input a pair of siblings $\mathsf{sbl} = (\mathsf{sbl}_0, \mathsf{sbl}_1)$, each of length κ, and generates two laconic OT Send messages with sbl_b as the digest and $\mathsf{Keys}, \widetilde{\mathsf{Keys}}$ as message vectors respectively. Further, it also has the randomness needed for $\ell OT_{\mathsf{const}}.\mathsf{Send}$ hardwired inside it.

The reading circuit $\mathsf{C}^{\mathsf{read}}$ has a location $t \in [2\kappa]$, and messages $m_0, m_1 \in \{0,1\}^\kappa$ hardwired inside it. It takes as input a path of siblings from the root to a leaf, reads the t-th bit of the leaf, and outputs either m_0 or m_1 depending on that bit.

The writing circuit $\mathsf{C}^{\mathsf{write}}$ has hardwired inside it a common reference string crs, a location $L \in [M]$, a bit b and a vector of messages Keys consisting of κ key-pairs. It takes as input a path of siblings from the root to a leaf, changes the t-th bit of the leaf to b (where L corresponds to t-th location in the leaf), recomputes the Merkle tree root along the path, and outputs the corresponding labels for the new root/digest.

Circuit $\mathsf{C}^{\mathsf{trav}}$

Hardwired Values: crs, b, Keys, $\widetilde{\mathsf{Keys}}$, r, $\widetilde{r}$

Input: sbl

Parse sbl as $(\mathsf{sbl}_0, \mathsf{sbl}_1)$

$\mathsf{e} \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Send}(\mathsf{crs}, \mathsf{sbl}_b, \mathsf{Keys}; r)$

$\widetilde{\mathsf{e}} \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Send}(\mathsf{crs}, \mathsf{sbl}_b, \widetilde{\mathsf{Keys}}; \widetilde{r})$

Output $(\mathsf{e}, \widetilde{\mathsf{e}})$

Figure 2.2: Description of the traversing circuit $\mathsf{C}^{\mathsf{trav}}[\mathsf{crs}, b, \mathsf{Keys}, \widetilde{\mathsf{Keys}}, r, \widetilde{r}]$.

Circuit $\mathsf{C}^{\mathsf{read}}$

Hardwired Values: t, m_0, m_1

Input: path

Parse $\mathsf{path} = (\mathsf{sbl}^1, \ldots, \mathsf{sbl}^{d-1}, \mathsf{leaf})$

Output $m_{\mathsf{leaf}[t]}$

Figure 2.3: Description of the reading circuit $\mathsf{C}^{\mathsf{read}}[t, m_0, m_1]$.

Circuit $\mathsf{C}^{\mathsf{write}}$

Hardwired Values: crs, L, b, Keys

Input: path

Parse $L = (b_1, b_2, \ldots, b_{d-1}, t)$

Parse $\mathsf{path} = (\mathsf{sbl}^1, \ldots, \mathsf{sbl}^{d-1}, \mathsf{leaf})$, and parse $\mathsf{sbl}^i = (\mathsf{sbl}^i_0, \mathsf{sbl}^i_1)$ for $i \in [d-1]$

$\mathsf{leaf}[t] \leftarrow b$

$\mathsf{sbl}^d \leftarrow \mathsf{leaf}$

For $i = d-1$ downto 1:

  $\mathsf{sbl}^i_{b_i} \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Hash}(\mathsf{crs}, \mathsf{sbl}^{i+1})$

$\mathsf{digest}^* \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Hash}(\mathsf{crs}, \mathsf{sbl}^1)$.

Output $\mathsf{Keys}_{\mathsf{digest}^*}$

Figure 2.4: Description of the writing circuit $\mathsf{C}^{\mathsf{write}}[\mathsf{crs}, L, b, \mathsf{Keys}]$.

Now we construct the updatable ℓOT, namely ($\mathsf{crsGen}, \mathsf{Hash}, \mathsf{Send}, \mathsf{Receive}, \mathsf{SendWrite}, \mathsf{ReceiveWrite}$) as follows.

- $\mathsf{crsGen}(1^\kappa)$: Sample $\mathsf{crs} \leftarrow \ell OT_{\mathsf{const}}.\mathsf{crsGen}(1^\kappa)$ and output crs.
- $\mathsf{Hash}(\mathsf{crs}, D \in \{0,1\}^M)$:
  Build a Merkle tree $\hat{\mathsf{D}}$ of D using $\ell OT_{\mathsf{const}}.\mathsf{Hash}(\mathsf{crs}, \cdot)$, as in Section 2.4.1.2.
  Let digest be the root of $\hat{\mathsf{D}}$.
  Output $(\mathsf{digest}, \hat{\mathsf{D}})$.
- $\mathsf{Send}(\mathsf{crs}, \mathsf{digest}, L, m_0, m_1)$:
  Parse $L = (b_1, b_2, \ldots, b_{d-1}, t)$.
  Pick $\left(\widetilde{\mathsf{Keys}}^1, \ldots, \widetilde{\mathsf{Keys}}^d\right)$ as input keys for $\mathsf{C}^{\mathsf{read}}$,
    where $\widetilde{\mathsf{Keys}}^i$ corresponds to the input keys of sbl^i for $i \in [d-1]$,
    and $\widetilde{\mathsf{Keys}}^d$ corresponds to the input keys of leaf.
  $\widetilde{\mathsf{C}}^{\mathsf{read}} \leftarrow \mathsf{GCircuit}\left(1^\kappa, \mathsf{C}^{\mathsf{read}}[t, m_0, m_1], \left(\widetilde{\mathsf{Keys}}^1, \ldots, \widetilde{\mathsf{Keys}}^d\right)\right)$
  Let Keys^d be 0^*
  For $i = d-1$ downto 1:
    Pick Keys^i as input keys for $\mathsf{C}^{\mathsf{trav}}$
    Pick $r_i, \widetilde{r}_i$ as random coins for $\ell OT_{\mathsf{const}}.\mathsf{Send}$
    $\widetilde{\mathsf{C}}_i \leftarrow \mathsf{GCircuit}\left(1^\kappa, \mathsf{C}^{\mathsf{trav}}[\mathsf{crs}, b_i, \mathsf{Keys}^{i+1}, \widetilde{\mathsf{Keys}}^{i+1}, r_i, \widetilde{r}_i], \mathsf{Keys}^i\right)$
  $\mathsf{e}_0 \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Send}(\mathsf{crs}, \mathsf{digest}, \mathsf{Keys}^1)$
  $\widetilde{\mathsf{e}}_0 \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Send}(\mathsf{crs}, \mathsf{digest}, \widetilde{\mathsf{Keys}}^1)$
  Output $\mathsf{e} = (\mathsf{e}_0, \widetilde{\mathsf{e}}_0, \widetilde{\mathsf{C}}_1, \ldots, \widetilde{\mathsf{C}}_{d-1}, \widetilde{\mathsf{C}}^{\mathsf{read}})$
- $\mathsf{Receive}^{\hat{\mathsf{D}}}(\mathsf{crs}, L, \mathsf{e})$:
  Parse $\mathsf{e} = (\mathsf{e}_0, \widetilde{\mathsf{e}}_0, \widetilde{\mathsf{C}}_1, \ldots, \widetilde{\mathsf{C}}_{d-1}, \widetilde{\mathsf{C}}^{\mathsf{read}})$
  Parse $L = (b_1, b_2, \ldots, b_{d-1}, t)$
  Parse $\hat{\mathsf{D}}$ as a Merkle tree.
  Denote the end node of path $b_1 b_2 \ldots b_i$ by $\hat{\mathsf{D}}_{b_1 b_2 \ldots b_i}$.
  For $i = 1$ to $d-1$:
    $\mathsf{sbl}^i \leftarrow (\hat{\mathsf{D}}_{b_1 \ldots b_{i-1} 0}, \hat{\mathsf{D}}_{b_1 \ldots b_{i-1} 1})$
    $\mathsf{Labels}^i \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Receive}(\mathsf{crs}, \mathsf{e}_{i-1}, \mathsf{sbl}^i)$
    $\widetilde{\mathsf{Labels}}^i \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Receive}(\mathsf{crs}, \widetilde{\mathsf{e}}_{i-1}, \mathsf{sbl}^i)$
    $(\mathsf{e}_i, \widetilde{\mathsf{e}}_i) \leftarrow \mathsf{GEval}(\widetilde{\mathsf{C}}_i, \mathsf{Labels}^i)$
  $\mathsf{leaf} \leftarrow (\hat{\mathsf{D}}_{b_1 \ldots b_{d-1} 0}, \hat{\mathsf{D}}_{b_1 \ldots b_{d-1} 1})$
  $\widetilde{\mathsf{Labels}}^d \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Receive}(\mathsf{crs}, \widetilde{\mathsf{e}}_{d-1}, \mathsf{leaf})$
  $m \leftarrow \mathsf{GEval}\left(\widetilde{\mathsf{C}}^{\mathsf{read}}, \left(\widetilde{\mathsf{Labels}}^1, \ldots, \widetilde{\mathsf{Labels}}^d\right)\right)$

Output m

- $\mathsf{SendWrite}(\mathsf{crs}, \mathsf{digest}, L, b, \{m_{j,0}, m_{j,1}\}_{j=1}^{\kappa})$:
  Parse $L = (b_1, b_2, \ldots, b_{d-1}, t)$.
  Pick $\left(\widetilde{\mathsf{Keys}}^1, \ldots, \widetilde{\mathsf{Keys}}^d\right)$ as input keys for $\mathsf{C}^{\mathsf{write}}$,
    where $\widetilde{\mathsf{Keys}}^i$ corresponds to the input keys of sbl^i for $i \in [d-1]$,
    and $\widetilde{\mathsf{Keys}}^d$ corresponds to the input keys of leaf.
  $\widetilde{\mathsf{C}}^{\mathsf{write}} \leftarrow \mathsf{GCircuit}\left(1^\kappa, \mathsf{C}^{\mathsf{write}}[\mathsf{crs}, L, b, \{m_{j,0}, m_{j,1}\}_{j=1}^{\kappa}], \left(\widetilde{\mathsf{Keys}}^1, \ldots, \widetilde{\mathsf{Keys}}^d\right)\right)$
  Let Keys^d be 0^*
  For $i = d-1$ downto 1:
    Pick Keys^i as input keys for $\mathsf{C}^{\mathsf{trav}}$
    Pick $r_i, \widetilde{r}_i$ as random coins for $\ell OT_{\mathsf{const}}.\mathsf{Send}$
    $\widetilde{\mathsf{C}}_i \leftarrow \mathsf{GCircuit}\left(1^\kappa, \mathsf{C}^{\mathsf{trav}}[\mathsf{crs}, b_i, \mathsf{Keys}^{i+1}, \widetilde{\mathsf{Keys}}^{i+1}, r_i, \widetilde{r}_i], \mathsf{Keys}^i\right)$
  $\mathsf{e}_0 \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Send}(\mathsf{crs}, \mathsf{digest}, \mathsf{Keys}^1)$
  $\widetilde{\mathsf{e}}_0 \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Send}(\mathsf{crs}, \mathsf{digest}, \widetilde{\mathsf{Keys}}^1)$
  Output $\mathsf{e}_{\mathsf{w}} = (\mathsf{e}_0, \widetilde{\mathsf{e}}_0, \widetilde{\mathsf{C}}_1, \ldots, \widetilde{\mathsf{C}}_{d-1}, \widetilde{\mathsf{C}}^{\mathsf{write}})$

- $\mathsf{ReceiveWrite}^{\hat{\mathsf{D}}}(\mathsf{crs}, L, b, \mathsf{e}_{\mathsf{w}})$:
  Parse $\mathsf{e}_{\mathsf{w}} = (\mathsf{e}_0, \widetilde{\mathsf{e}}_0, \widetilde{\mathsf{C}}_1, \ldots, \widetilde{\mathsf{C}}_{d-1}, \widetilde{\mathsf{C}}^{\mathsf{write}})$
  Parse $L = (b_1, b_2, \ldots, b_{d-1}, t)$
  Parse $\hat{\mathsf{D}}$ as a Merkle tree.
  Denote the end node of path $b_1 b_2 \ldots b_i$ by $\hat{\mathsf{D}}_{b_1 b_2 \ldots b_i}$.

  Computing messages corresponding to the new digest:
  For $i = 1$ to $d-1$:
    $\mathsf{sbl}^i \leftarrow (\hat{\mathsf{D}}_{b_1 \ldots b_{i-1} 0}, \hat{\mathsf{D}}_{b_1 \ldots b_{i-1} 1})$
    $\mathsf{Labels}^i \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Receive}(\mathsf{crs}, \mathsf{e}_{i-1}, \mathsf{sbl}^i)$
    $\widetilde{\mathsf{Labels}}^i \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Receive}(\mathsf{crs}, \widetilde{\mathsf{e}}_{i-1}, \mathsf{sbl}^i)$
    $(\mathsf{e}_i, \widetilde{\mathsf{e}}_i) \leftarrow \mathsf{GEval}(\widetilde{\mathsf{C}}_i, \mathsf{Labels}^i)$
  $\mathsf{leaf} \leftarrow (\hat{\mathsf{D}}_{b_1 \ldots b_{d-1} 0}, \hat{\mathsf{D}}_{b_1 \ldots b_{d-1} 1})$
  $\widetilde{\mathsf{Labels}}^d \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Receive}(\mathsf{crs}, \widetilde{\mathsf{e}}_{d-1}, \mathsf{leaf})$
  $\{m_j\}_{j=1}^{\kappa} \leftarrow \mathsf{GEval}\left(\widetilde{\mathsf{C}}^{\mathsf{write}}, \left(\widetilde{\mathsf{Labels}}^1, \ldots, \widetilde{\mathsf{Labels}}^d\right)\right)$

  Updating the Merkle tree:
  $\left(\hat{\mathsf{D}}_{b_1 \ldots b_{d-1} 0} || \hat{\mathsf{D}}_{b_1 \ldots b_{d-1} 1}\right)[t] \leftarrow b$
  For $i = d-1$ downto 0:
    $\hat{\mathsf{D}}_{b_1 \ldots b_i} \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Hash}(\mathsf{crs}, \hat{\mathsf{D}}_{b_1 \ldots b_i 0} || \hat{\mathsf{D}}_{b_1 \ldots b_i 1})$
  Update digest with the new root of $\hat{\mathsf{D}}$

Output $\{m_j\}_{j=1}^{\kappa}$

Efficiency. It can be seen from the scheme that the length of digest is κ. The algorithm Hash runs in time $|D| \cdot \mathsf{poly}(\log |D|, \kappa)$. Furthermore, $\mathsf{Send}, \mathsf{Receive}, \mathsf{SendWrite}, \mathsf{SendWrite}$ all run in time $\mathsf{poly}(\log |D|, \kappa)$.

Correctness. We briefly argue (perfect) correctness of the updatable laconic OT scheme. Given a ciphertext $\mathsf{e} = (\mathsf{e}_0, \widetilde{\mathsf{e}}_0, \widetilde{\mathsf{C}}_1, \ldots, \widetilde{\mathsf{C}}_{d-1}, \widetilde{\mathsf{C}}^{\mathsf{read}})$ computed by Send, correctness of ℓOT_{const} ensures that $\mathsf{Labels}^1 \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Receive}(\mathsf{crs}, \mathsf{e}_0, \mathsf{sbl}^1)$ outputs the correct labels for $\widetilde{\mathsf{C}}_1$ and that $\widetilde{\mathsf{Labels}}^1 \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Receive}(\mathsf{crs}, \widetilde{\mathsf{e}}_0, \mathsf{sbl}^1)$ outputs the correct labels for $\widetilde{\mathsf{C}}^{\mathsf{read}}$, namely $\mathsf{Labels}^1 = \mathsf{Keys}^1_{\mathsf{sbl}^1}$ and $\widetilde{\mathsf{Labels}}^1 = \widetilde{\mathsf{Keys}}^1_{\mathsf{sbl}^1}$. In turn, correctness of the garbling scheme guarantees that $\widetilde{\mathsf{C}}_1$ outputs the correct $(\mathsf{e}_1, \widetilde{\mathsf{e}}_1)$, namely $\mathsf{e}_1 = \ell OT_{\mathsf{const}}.\mathsf{Send}(\mathsf{crs}, \mathsf{sbl}^1_{b_1}, \mathsf{Keys}^2; r_1)$ and $\widetilde{\mathsf{e}}_1 = \ell OT_{\mathsf{const}}.\mathsf{Send}(\mathsf{crs}, \mathsf{sbl}^1_{b_1}, \widetilde{\mathsf{Keys}}^2; \widetilde{r}_1)$. It follows inductively that for every $i = 1, 2, \ldots, d-1$, $\mathsf{Labels}^i = \mathsf{Keys}^i_{\mathsf{sbl}^i}$, $\widetilde{\mathsf{Labels}}^i = \widetilde{\mathsf{Keys}}^i_{\mathsf{sbl}^i}$, $\mathsf{e}_i = \ell OT_{\mathsf{const}}.\mathsf{Send}(\mathsf{crs}, \mathsf{sbl}^i_{b_i}, \mathsf{Keys}^{i+1}; r_i)$, and $\widetilde{\mathsf{e}}_i = \ell OT_{\mathsf{const}}.\mathsf{Send}(\mathsf{crs}, \mathsf{sbl}^i_{b_i}, \widetilde{\mathsf{Keys}}^{i+1}; \widetilde{r}_i)$. Again by using the correctness of ℓOT_{const}, $\widetilde{\mathsf{Labels}}^d \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Receive}(\mathsf{crs}, \widetilde{\mathsf{e}}_{d-1}, \mathsf{leaf})$ gives $\widetilde{\mathsf{Labels}}^d = \widetilde{\mathsf{Keys}}^d_{\mathsf{leaf}}$. Then by using correctness of the garbling scheme it follows that evaluating $\widetilde{\mathsf{C}}^{\mathsf{read}}$ gives the correct output $m_{D[L]}$. Correctness with regard to writes can be argued analogously.

2.4.3 Security Proof

In this section, we prove the security of the above updatable laconic OT scheme.

Theorem 2.4.1 (Sender Privacy against Semi-honest Receivers)**.** *Given that ℓOT_{const} has sender privacy and that the garbled circuit scheme $\mathsf{GCircuit}$ is secure, the updatable laconic OT scheme ℓOT has sender privacy.*

Proof. Let $\ell\mathsf{OTSim}_{\mathsf{const}}$ be the simulator for ℓOT_{const} and $\mathsf{CircSim}$ be the simulator for the garbling scheme $\mathsf{GCircuit}$. Below, we provide the two simulators $\ell\mathsf{OTSim}$ for a read and $\ell\mathsf{OTSimWrite}$ for the write.

- $\ell\mathsf{OTSim}(\mathsf{crs}, D, L, m)$:
  $(\mathsf{digest}, \hat{\mathsf{D}}) \leftarrow \mathsf{Hash}(\mathsf{crs}, D)$
  Parse $L = (b_1, b_2, \ldots, b_{d-1}, t)$
  $\left(\widetilde{\mathsf{C}}^{\mathsf{read}}, \left(\widetilde{\mathsf{Labels}}^1, \ldots, \widetilde{\mathsf{Labels}}^d\right)\right) \leftarrow \mathsf{CircSim}(1^\kappa, \mathsf{C}^{\mathsf{read}}, m)$
  $\mathsf{leaf} \leftarrow (\hat{\mathsf{D}}_{b_1 \ldots b_{d-1} 0}, \hat{\mathsf{D}}_{b_1 \ldots b_{d-1} 1})$
  $\mathsf{e}_{d-1} \leftarrow \ell\mathsf{OTSim}_{\mathsf{const}}(\mathsf{crs}, \mathsf{leaf}, 0^*)$

$\widetilde{\mathsf{e}}_{d-1} \leftarrow \ell\mathsf{OTSim}_{\mathsf{const}}\left(\mathsf{crs}, \mathsf{leaf}, \widetilde{\mathsf{Labels}}^{d}\right)$
For $i = d-1$ downto 1:
$(\widetilde{\mathsf{C}}_i, \mathsf{Labels}^i) \leftarrow \mathsf{CircSim}(1^\kappa, \mathsf{C}^{\mathsf{trav}}, (\mathsf{e}_i, \widetilde{\mathsf{e}}_i))$
$\mathsf{sbl}^i \leftarrow (\hat{\mathsf{D}}_{b_1 \ldots b_{i-1}0}, \hat{\mathsf{D}}_{b_1 \ldots b_{i-1}1})$
$\mathsf{e}_{i-1} \leftarrow \ell\mathsf{OTSim}_{\mathsf{const}}(\mathsf{crs}, \mathsf{sbl}^i, \mathsf{Labels}^i)$
$\widetilde{\mathsf{e}}_{i-1} \leftarrow \ell\mathsf{OTSim}_{\mathsf{const}}(\mathsf{crs}, \mathsf{sbl}^i, \widetilde{\mathsf{Labels}}^i)$
Output $\mathsf{e} = (\mathsf{e}_0, \widetilde{\mathsf{e}}_0, \widetilde{\mathsf{C}}_1, \ldots, \widetilde{\mathsf{C}}_{d-1}, \widetilde{\mathsf{C}}^{\mathsf{read}})$

- $\ell\mathsf{OTSimWrite}(\mathsf{crs}, D, L, b, \{m_j\}_{j=1}^\kappa)$:
$(\mathsf{digest}, \hat{\mathsf{D}}) \leftarrow \mathsf{Hash}(\mathsf{crs}, D)$
Parse $L = (b_1, b_2, \ldots, b_{d-1}, t)$
$\left(\widetilde{\mathsf{C}}^{\mathsf{write}}, \left(\widetilde{\mathsf{Labels}}^1, \ldots, \widetilde{\mathsf{Labels}}^d\right)\right) \leftarrow \mathsf{CircSim}(1^\kappa, \mathsf{C}^{\mathsf{write}}, \{m_j\}_{j=1}^\kappa)$
$\mathsf{leaf} \leftarrow (\hat{\mathsf{D}}_{b_1 \ldots b_{d-1}0}, \hat{\mathsf{D}}_{b_1 \ldots b_{d-1}1})$
$\mathsf{e}_{d-1} \leftarrow \ell\mathsf{OTSim}_{\mathsf{const}}(\mathsf{crs}, \mathsf{leaf}, 0^*)$
$\widetilde{\mathsf{e}}_{d-1} \leftarrow \ell\mathsf{OTSim}_{\mathsf{const}}\left(\mathsf{crs}, \mathsf{leaf}, \widetilde{\mathsf{Labels}}^d\right)$
For $i = d-1$ downto 1:
$(\widetilde{\mathsf{C}}_i, \mathsf{Labels}^i) \leftarrow \mathsf{CircSim}(1^\kappa, \mathsf{C}^{\mathsf{trav}}, (\mathsf{e}_i, \widetilde{\mathsf{e}}_i))$
$\mathsf{sbl}^i \leftarrow (\hat{\mathsf{D}}_{b_1 \ldots b_{i-1}0}, \hat{\mathsf{D}}_{b_1 \ldots b_{i-1}1})$
$\mathsf{e}_{i-1} \leftarrow \ell\mathsf{OTSim}_{\mathsf{const}}(\mathsf{crs}, \mathsf{sbl}^i, \mathsf{Labels}^i)$
$\widetilde{\mathsf{e}}_{i-1} \leftarrow \ell\mathsf{OTSim}_{\mathsf{const}}(\mathsf{crs}, \mathsf{sbl}^i, \widetilde{\mathsf{Labels}}^i)$
Output $\mathsf{e}_{\mathsf{w}} = (\mathsf{e}_0, \widetilde{\mathsf{e}}_0, \widetilde{\mathsf{C}}_1, \ldots, \widetilde{\mathsf{C}}_{d-1}, \widetilde{\mathsf{C}}^{\mathsf{write}})$

In the following we only prove sender security with regard to reads. Since $(\mathsf{Send}, \ell\mathsf{OTSim})$ and $(\mathsf{SendWrite}, \ell\mathsf{OTSimWrite})$ are very similar, sender security with regard to writes can be argued analogously.

We prove security via a hybrid argument. In the first hybrid, we replace the ciphertexts e_0 and $\widetilde{\mathsf{e}}_0$ computed by $\ell OT_{\mathsf{const}}.\mathsf{Send}$ with ciphertexts computed by $\ell\mathsf{OTSim}_{\mathsf{const}}$.

Afterwards, we can use security of the garbling scheme to replace the honestly generated $\widetilde{\mathsf{C}}_1$ with a simulated one, and run $\ell\mathsf{OTSim}_{\mathsf{const}}$ using the simulated input labels of $\widetilde{\mathsf{C}}_1$. As the output of $\widetilde{\mathsf{C}}_1$ is again a pair of ciphertexts $(\mathsf{e}_1, \widetilde{\mathsf{e}}_1)$, we will simulate it using $\ell\mathsf{OTSim}_{\mathsf{const}}$ in the next hybrid. We continue alternating between simulating the garbled circuits and simulating the ciphertexts, until reaching the reading circuit. Once we reach the reading circuit, it holds that all $\widetilde{\mathsf{Labels}}^i$ are information theoretically fixed to the path from the root to the leaf containing L. We will then invoke the garbled circuit security of the reading circuit, and conclude the hybrid argument.

The formal proof is as follows. For every PPT machine $\mathcal{A}$, let $\mathsf{crs} \leftarrow \mathsf{crsGen}(1^\kappa)$, and let $(D, L, m_0, m_1) \leftarrow \mathcal{A}(\mathsf{crs})$. Further let $\mathsf{digest} \leftarrow \mathsf{Hash}(\mathsf{crs}, D)$. Then we will prove that

the two distributions $(\mathsf{crs}, \mathsf{Send}(\mathsf{crs}, \mathsf{digest}, L, m_0, m_1))$ and $(\mathsf{crs}, \ell\mathsf{OTSim}(\mathsf{crs}, D, L, m_{D[L]}))$ are computationally indistinguishable. Consider the following hybrids.

- **Hybrid 0**: This is the real experiment, i.e., $(\mathsf{crs}, \mathsf{Send}(\mathsf{crs}, \mathsf{digest}, L, m_0, m_1))$.
- **Hybrid 1**: Same has hybrid 0, except that e_0 and $\widetilde{\mathsf{e}}_0$ are computed as follows.

  $\boxed{(\mathsf{digest}, \hat{\mathsf{D}}) \leftarrow \mathsf{Hash}(\mathsf{crs}, D)}$
  Parse $L = (b_1, b_2, \ldots, b_{d-1}, t)$.
  Pick $\left(\widetilde{\mathsf{Keys}}^1, \ldots, \widetilde{\mathsf{Keys}}^d\right)$ as input keys for $\mathsf{C}^{\mathsf{read}}$
  $\widetilde{\mathsf{C}}^{\mathsf{read}} \leftarrow \mathsf{GCircuit}\left(1^\kappa, \mathsf{C}^{\mathsf{read}}[t, m_0, m_1], \left(\widetilde{\mathsf{Keys}}^1, \ldots, \widetilde{\mathsf{Keys}}^d\right)\right)$
  Let Keys^d be 0^*
  For $i = d-1$ downto 1:
    Pick Keys^i as input keys for $\mathsf{C}^{\mathsf{trav}}$
    Pick $r_i, \widetilde{r}_i$ as random coins for $\ell OT_{\mathsf{const}}.\mathsf{Send}$
    $\widetilde{\mathsf{C}}_i \leftarrow \mathsf{GCircuit}\left(1^\kappa, \mathsf{C}^{\mathsf{trav}}[\mathsf{crs}, b_i, \mathsf{Keys}^{i+1}, \widetilde{\mathsf{Keys}}^{i+1}, r_i, \widetilde{r}_i], \mathsf{Keys}^i\right)$
  $\boxed{\mathsf{sbl}^1 \leftarrow (\hat{\mathsf{D}}_0, \hat{\mathsf{D}}_1)}$
  $\boxed{\mathsf{Labels}^1 \leftarrow \mathsf{Keys}^1_{\mathsf{sbl}^1}}$
  $\boxed{\widetilde{\mathsf{Labels}}^1 \leftarrow \widetilde{\mathsf{Keys}}^1_{\mathsf{sbl}^1}}$
  $\boxed{\mathsf{e}_0 \leftarrow \ell\mathsf{OTSim}_{\mathsf{const}}(\mathsf{crs}, \mathsf{sbl}^1, \mathsf{Labels}^1)}$
  $\boxed{\widetilde{\mathsf{e}}_0 \leftarrow \ell\mathsf{OTSim}_{\mathsf{const}}(\mathsf{crs}, \mathsf{sbl}^1, \widetilde{\mathsf{Labels}}^1)}$
  Output $\mathsf{e} = (\mathsf{e}_0, \widetilde{\mathsf{e}}_0, \widetilde{\mathsf{C}}_1, \ldots, \widetilde{\mathsf{C}}_{d-1}, \widetilde{\mathsf{C}}^{\mathsf{read}})$

  The differences between hybrid 0 and hybrid 1 have been marked with boxes. Indistinguishability between hybrid 0 and hybrid 1 can be argued from the multi-execution sender security of ℓOT_{const} via the following reduction. Given crs by the experiment and the adversarial input D, compute hybrid 1 until e_0 and $\widetilde{\mathsf{e}}_0$ are computed. In particular, compute $\hat{\mathsf{D}}, \mathsf{Keys}^1, \widetilde{\mathsf{Keys}}^1, \mathsf{sbl}^1, \mathsf{Labels}^1 = \mathsf{Keys}^1_{\mathsf{sbl}^1}, \widetilde{\mathsf{Labels}}^1 = \widetilde{\mathsf{Keys}}^1_{\mathsf{sbl}^1}$. Then choose sbl^1 as the database and $(\mathsf{Keys}^1, \widetilde{\mathsf{Keys}}^1)$ as the messages for ℓOT_{const}, and obtain the challenge $(\mathsf{e}_0^*, \widetilde{\mathsf{e}}_0^*)$, which is from one of the following two distributions:

$$\left(\ell OT_{\mathsf{const}}.\mathsf{Send}(\mathsf{crs}, \mathsf{sbl}^1, \mathsf{Keys}^1), \ell OT_{\mathsf{const}}.\mathsf{Send}(\mathsf{crs}, \mathsf{sbl}^1, \widetilde{\mathsf{Keys}}^1)\right);$$

$$\left(\ell\mathsf{OTSim}_{\mathsf{const}}(\mathsf{crs}, \mathsf{sbl}^1, \mathsf{Labels}^1), \ell\mathsf{OTSim}_{\mathsf{const}}(\mathsf{crs}, \mathsf{sbl}^1, \widetilde{\mathsf{Labels}}^1)\right).$$

  If $(\mathsf{e}_0^*, \widetilde{\mathsf{e}}_0^*)$ is from the first distribution, then it results in hybrid 0; otherwise it results in hybrid 1. Hence the indistinguishability of the two distributions implies indistinguishability of the two hybrids.

- **Hybrid** $2k$ ($k = 1, 2, \ldots, d-1$): Same has hybrid $2k-1$, except that $\widetilde{\mathsf{C}}_k$ is computed as follows.

  $(\mathsf{digest}, \hat{\mathsf{D}}) \leftarrow \mathsf{Hash}(\mathsf{crs}, D)$
  Parse $L = (b_1, b_2, \ldots, b_{d-1}, t)$.
  Pick $\left(\widetilde{\mathsf{Keys}}^1, \ldots, \widetilde{\mathsf{Keys}}^d\right)$ as input keys for $\mathsf{C}^{\mathsf{read}}$
  $\widetilde{\mathsf{C}}^{\mathsf{read}} \leftarrow \mathsf{GCircuit}\left(1^\kappa, \mathsf{C}^{\mathsf{read}}[t, m_0, m_1], \left(\widetilde{\mathsf{Keys}}^1, \ldots, \widetilde{\mathsf{Keys}}^d\right)\right)$
  Let Keys^d be 0^*
  For $i = d-1$ downto $\boxed{k+1}$:
    Pick Keys^i as input keys for $\mathsf{C}^{\mathsf{trav}}$
    Pick $r_i, \widetilde{r}_i$ as random coins for $\ell OT_{\mathsf{const}}.\mathsf{Send}$
    $\widetilde{\mathsf{C}}_i \leftarrow \mathsf{GCircuit}\left(1^\kappa, \mathsf{C}^{\mathsf{trav}}[\mathsf{crs}, b_i, \mathsf{Keys}^{i+1}, \widetilde{\mathsf{Keys}}^{i+1}, r_i, \widetilde{r}_i], \mathsf{Keys}^i\right)$
  $\boxed{\mathsf{sbl}^k \leftarrow (\hat{\mathsf{D}}_{b_1 \ldots b_{k-1} 0}, \hat{\mathsf{D}}_{b_1 \ldots b_{k-1} 1})}$
  $\boxed{\mathsf{e}_k \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Send}(\mathsf{crs}, \mathsf{sbl}^k_{b_k}, \mathsf{Keys}^{k+1})}$
  $\boxed{\widetilde{\mathsf{e}}_k \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Send}(\mathsf{crs}, \mathsf{sbl}^k_{b_k}, \widetilde{\mathsf{Keys}}^{k+1})}$
  For $i = \boxed{k}$ downto 1:
    $\boxed{(\widetilde{\mathsf{C}}_i, \mathsf{Labels}^i) \leftarrow \mathsf{CircSim}(1^\kappa, \mathsf{C}^{\mathsf{trav}}, (\mathsf{e}_i, \widetilde{\mathsf{e}}_i))}$
    $\mathsf{sbl}^i \leftarrow (\hat{\mathsf{D}}_{b_1 \ldots b_{i-1} 0}, \hat{\mathsf{D}}_{b_1 \ldots b_{i-1} 1})$
    $\widetilde{\mathsf{Labels}}^i \leftarrow \widetilde{\mathsf{Keys}}^i_{\mathsf{sbl}^i}$
    $\mathsf{e}_{i-1} \leftarrow \ell\mathsf{OTSim}_{\mathsf{const}}(\mathsf{crs}, \mathsf{sbl}^i, \mathsf{Labels}^i)$
    $\widetilde{\mathsf{e}}_{i-1} \leftarrow \ell\mathsf{OTSim}_{\mathsf{const}}(\mathsf{crs}, \mathsf{sbl}^i, \widetilde{\mathsf{Labels}}^i)$
  Output $\mathsf{e} = (\mathsf{e}_0, \widetilde{\mathsf{e}}_0, \widetilde{\mathsf{C}}_1, \ldots, \widetilde{\mathsf{C}}_{d-1}, \widetilde{\mathsf{C}}^{\mathsf{read}})$

- **Hybrid** $2k+1$ ($k = 1, 2, \ldots, d-1$): Same has hybrid $2k$, except that e_k and $\widetilde{\mathsf{e}}_k$ are computed as follows.

  $(\mathsf{digest}, \hat{\mathsf{D}}) \leftarrow \mathsf{Hash}(\mathsf{crs}, D)$
  Parse $L = (b_1, b_2, \ldots, b_{d-1}, t)$.
  Pick $\left(\widetilde{\mathsf{Keys}}^1, \ldots, \widetilde{\mathsf{Keys}}^d\right)$ as input keys for $\mathsf{C}^{\mathsf{read}}$
  $\widetilde{\mathsf{C}}^{\mathsf{read}} \leftarrow \mathsf{GCircuit}\left(1^\kappa, \mathsf{C}^{\mathsf{read}}[t, m_0, m_1], \left(\widetilde{\mathsf{Keys}}^1, \ldots, \widetilde{\mathsf{Keys}}^d\right)\right)$
  Let Keys^d be 0^*
  For $i = d-1$ downto $k+1$:
    Pick Keys^i as input keys for $\mathsf{C}^{\mathsf{trav}}$
    Pick $r_i, \widetilde{r}_i$ as random coins for $\ell OT_{\mathsf{const}}.\mathsf{Send}$
    $\widetilde{\mathsf{C}}_i \leftarrow \mathsf{GCircuit}\left(1^\kappa, \mathsf{C}^{\mathsf{trav}}[\mathsf{crs}, b_i, \mathsf{Keys}^{i+1}, \widetilde{\mathsf{Keys}}^{i+1}, r_i, \widetilde{r}_i], \mathsf{Keys}^i\right)$

$\boxed{\mathsf{sbl}^{k+1} \leftarrow (\hat{\mathsf{D}}_{b_1 \ldots b_k 0}, \hat{\mathsf{D}}_{b_1 \ldots b_k 1})}$

$\boxed{\mathsf{Labels}^{k+1} \leftarrow \mathsf{Keys}^{k+1}_{\mathsf{sbl}^{k+1}}}$

$\boxed{\widetilde{\mathsf{Labels}}^{k+1} \leftarrow \widetilde{\mathsf{Keys}}^{k+1}_{\mathsf{sbl}^{k+1}}}$

$\boxed{\mathsf{e}_k \leftarrow \ell\mathsf{OTSim}_{\mathsf{const}}(\mathsf{crs}, \mathsf{sbl}^{k+1}, \mathsf{Labels}^{k+1})}$

$\boxed{\widetilde{\mathsf{e}}_k \leftarrow \ell\mathsf{OTSim}_{\mathsf{const}}(\mathsf{crs}, \mathsf{sbl}^{k+1}, \widetilde{\mathsf{Labels}}^{k+1})}$

For $i = k$ downto 1:

$(\widetilde{\mathsf{C}}_i, \mathsf{Labels}^i) \leftarrow \mathsf{CircSim}(1^\kappa, \mathsf{C}^{\mathsf{trav}}, (\mathsf{e}_i, \widetilde{\mathsf{e}}_i))$

$\mathsf{sbl}^i \leftarrow (\hat{\mathsf{D}}_{b_1 \ldots b_{i-1} 0}, \hat{\mathsf{D}}_{b_1 \ldots b_{i-1} 1})$

$\widetilde{\mathsf{Labels}}^i \leftarrow \widetilde{\mathsf{Keys}}^i_{\mathsf{sbl}^i}$

$\mathsf{e}_{i-1} \leftarrow \ell\mathsf{OTSim}_{\mathsf{const}}(\mathsf{crs}, \mathsf{sbl}^i, \mathsf{Labels}^i)$

$\widetilde{\mathsf{e}}_{i-1} \leftarrow \ell\mathsf{OTSim}_{\mathsf{const}}(\mathsf{crs}, \mathsf{sbl}^i, \widetilde{\mathsf{Labels}}^i)$

Output $\mathsf{e} = (\mathsf{e}_0, \widetilde{\mathsf{e}}_0, \widetilde{\mathsf{C}}_1, \ldots, \widetilde{\mathsf{C}}_{d-1}, \widetilde{\mathsf{C}}^{\mathsf{read}})$

We first show that hybrids $2k-1$ and $2k$ are indistinguishable via a reduction to the security of the garbling scheme $\mathsf{GCircuit}$. Notice that the only difference between hybrids $2k-1$ and $2k$ is $(\widetilde{\mathsf{C}}_k, \mathsf{e}_{k-1})$. Consider the following two distributions:

$$(\widetilde{\mathsf{C}}_k, \mathsf{Labels}^k) \leftarrow \left(\mathsf{GCircuit}\left(1^\kappa, \mathsf{C}^{\mathsf{trav}}[\mathsf{crs}, b_k, \mathsf{Keys}^{k+1}, \widetilde{\mathsf{Keys}}^{k+1}, r_i, \widetilde{r}_k], \mathsf{Keys}^k\right), \mathsf{Keys}^k_{\mathsf{sbl}^k}\right);$$

$$(\widetilde{\mathsf{C}}_k, \mathsf{Labels}^k) \leftarrow \mathsf{CircSim}\left(1^\kappa, \mathsf{C}^{\mathsf{trav}}, (\mathsf{e}_k, \widetilde{\mathsf{e}}_k)\right),$$

where $\mathsf{e}_k \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Send}(\mathsf{crs}, \mathsf{sbl}^k_{b_k}, \mathsf{Keys}^{k+1})$ and $\widetilde{\mathsf{e}}_k \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Send}(\mathsf{crs}, \mathsf{sbl}^k_{b_k}, \widetilde{\mathsf{Keys}}^{k+1})$. Notice that $(\mathsf{e}_k, \widetilde{\mathsf{e}}_k)$ is the output of $(\widetilde{\mathsf{C}}_k, \mathsf{Labels}^k)$ from the first distribution. By security of the garbled circuit scheme, the above two distributions are computationally indistinguishable. Furthermore, if $\widetilde{\mathsf{C}}_k$ is generated using the first distribution and e_{k-1} is computed using Labels^k from the first distribution, then it results in hybrid $2k-1$; otherwise it results in hybrid $2k$. Hence the two hybrids are computationally indistinguishable.

Indistinguishability of hybrids $2k$ and $2k+1$ follows again from sender security of ℓOT_{const}, in the same fashion as the indistinguishability between hybrids 0 and 1.

- **Hybrid** $2d$: This is the simulated experiment, namely $(\mathsf{crs}, \ell\mathsf{OTSim}(\mathsf{crs}, D, L, m_{D[L]}))$.

  The difference between hybrids $2d-1$ and $2d$ is $\left(\widetilde{\mathsf{C}}^{\mathsf{read}}, \widetilde{\mathsf{e}}_0, \ldots, \widetilde{\mathsf{e}}_{d-1}\right)$. The indistinguishability would follow from the security of garbled circuit scheme, similarly as when we argue indistinguishability hybrids $2k-1$ and $2k$.

□

Chapter 3

Non-Interactive Secure Computation

In this chapter, we apply laconic OT to reduce communication complexity in non-interactive secure computation on large inputs in the circuit setting as well as RAM setting. Our construction for the circuit setting is presented in Section 3.1 and for the RAM setting is presented in Section 3.2. Finally we discuss how to enhance the security guarantee from semi-honest security to malicious security in Section 3.3.

3.1 Circuit Setting

In the circuit setting, consider a receiver R, holding a large database D, publishes a *short* encoding of it such that any sender S, with private input x, can send a single message to reveal $C(x, D)$ to R. Here, C is the circuit being evaluated. For security, we want the receiver's encoding to hide D and the sender's message to hide x. For communication complexity, we restrict the receiver's published encoding to be independent of the size of her database.

This is a straightforward application of laconic OT. Recall the garbled circuit based approach to non-interactive secure computation, where R can publish the first message of a two-message oblivious transfer (OT) for his input D, and the sender responds with a garbled circuit for $C[x, \cdot]$ (with hardcoded input x) and sends the input labels corresponding to D via the second OT message. The downside of this protocol is that R's public message grows with the size of D, which could be substantially large.

We resolve this issue via our new primitive laconic OT. In our protocol, R's first message is the digest digest of his large database D. Next, the sender generates the garbled circuit for $C[x, \cdot]$ as before. It also transfers the labels for each location of D via laconic OT Send messages. Hence, by efficiency requirements of laconic OT, the length of R's public message is independent of the size of D. Moreover, sender privacy against a semi-honest receiver follows directly from the sender privacy of laconic OT and security of garbled circuits. To achieve receiver privacy, we can enhance the laconic OT with receiver privacy (discussed in Section 2.2.1).

3.2 RAM Setting

In this section, we consider the application of non-interactive secure computation in the RAM (random access machine) setting. It is the RAM version of the above application where S holds a RAM program P and R holds a large database D. As before, we want that (1) the length of R's first message is independent of $|D|$, (2) R's first message can be published and used by multiple senders, (3) the database is persistent for a sequence of programs for every sender, and (4) the computational complexity of both S and R per program execution grows only with running time of the corresponding program. For this application, we only achieve unprotected memory access (UMA) security against a corrupt receiver, i.e., the memory access pattern in the execution of $P^D(x)$ is leaked to the receiver. We achieve full security against a corrupt sender.

Organization. In Section 3.2.1 we give a technical overview of this application. In Section 3.2.2 we describe necessary background needed for our construction. Then we formalize our model in Section 3.2.3 and present our construction for a single sender executing a single program with the receiver in Sections 3.2.4. Correctness and security proofs are provided in Sections 3.2.5 and 3.2.6, respectively. Finally we discuss how to extend our construction to executing multiple programs on a persistent database in Section 3.2.7.

3.2.1 Technical Overview

For simplicity, consider a read-only program such that each CPU step outputs the next location to be read based on the value read from last location. At a high level, since we want the sender's complexity to grow only with the running time t of the program, we cannot create a garbled circuit that takes D as input. Instead, we would go via the garbled RAM based approaches where we have a sequence of t garbled circuits where each circuit executes one CPU step. A CPU step circuit takes the current CPU state and the last bit read from the database D as input and outputs an updated state and a new location to be read. The new location would be read from the database and fed into the next CPU step. The most non-trivial part in all garbled RAM constructions is being able to compute the correct labels for the next circuit based on the value of $D[L]$, where L is the location being read. Since we are working with garbled circuits, it is crucial for security that the receiver does not learn two labels for any input wire. We solve this issue via laconic OT as follows.

For the simpler case of sender security, R publishes the short digest of D, which is fed into the first garbled circuit and this digest is passed along the sequence of garbled circuits. When a circuit wants to read a location L, it outputs the laconic OT ciphertexts which encrypt the input keys for the next circuit and use digest of D as the hash value.[1] Security

[1]We note that the above idea of using laconic OT also gives a conceptually very simple solution for UMA secure garbled RAM scheme [LO13]. Moreover, there is a general transformation [GHL+14] that converts any UMA secure garbled RAM into one with full security via the usage of symmetric key encryption and oblivious

against a corrupt receiver follows from the sender security of laconic OT and security of garbled circuits. To achieve security against a corrupt sender, R does not publishes digest in the clear. Instead, the labels for digest for the first circuit are transferred to R via regular OT.

Note that the garbling time of the sender as well as execution time of the receiver will grow only with the running time of the program. This follows from the efficiency requirements of laconic OT.

Above, we did not describe how we deal with general programs that also write to the database or memory. We achieve this via updatable laconic OT (for definition see Section 2.2.2), This allows for transferring the labels for updated digest (corresponding to the updated database) to the next circuit. For a formal description of our scheme for general RAM programs, see Section 3.2.4.

Other Related Work. Prior works consider secure computation which hides the input size of one [MRK03; IP07; ADT11; LNO13] or both parties [LNO13]. Our notion only requires the receiver's communication cost to be independent of the its input size, and is therefore weaker. However, these results are largely restricted to special functionalities, such as zero-knowledge sets and computing certain branching programs (which imply input-size hiding private set intersection). The general result of [LNO13] uses FHE and as mentioned earlier needs more rounds of interaction.[2]

3.2.2 Background

We recall the needed background of RAM computation model and two-message oblivious transfer in this section. We also use garbled circuits (see Section 2.4.1.1) as building blocks.

3.2.2.1 Random Access Machine (RAM) Model of Computation

Now we define the RAM model of computation. Parts of this subsection have been taken verbatim from [GLO15].

Notations. The RAM model consists of a CPU and a memory storage of size M. The CPU executes a program that can access the memory by using read/write operations. In particular, for a program P with memory of size M we denote the initial contents of the memory data by $D \in \{0,1\}^M$. Additionally, the program gets a "short" input $x \in \{0,1\}^m$, which we alternatively think of as the initial state of the program. We use the notation $P^D(x)$ to denote the execution of program P with initial memory contents D and input x.

RAM. This would give a simplified construction of fully secure garbled RAM under DDH assumption.

[2]In an orthogonal work of Hubacek and Wichs [HW15] obtain constructions where the communication cost is independent of the length of the output of the computation using indistinguishability obfuscation [GGH+13].

The program P can read from and write to various locations in memory D throughout its execution.[3]

We will also consider the case where several different programs are executed sequentially and the memory persists between executions. We denote this process as $(y_1, \ldots, y_\ell) = (P_1(x_1), \ldots, P_\ell(x_\ell))^D$ to indicate that first $P_1^D(x_1)$ is executed, resulting in some memory contents D_1 and output y_1, then $P_2^{D_1}(x_2)$ is executed resulting in some memory contents D_2 and output y_2 etc. As an example, imagine that D is a huge database and the programs P_i are database queries that can read and possibly write to the database and are parameterized by some values x_i.

CPU-Step Circuit. Consider an execution of a RAM program which involves at most t CPU steps. We represent a RAM program P via t small *CPU-Step Circuits* each of which executes one CPU step. In this work we will denote one CPU step by:

$$C^P_{\mathsf{CPU}}(\mathsf{state}, \mathsf{rData}) = (\mathsf{state}', \mathsf{R/W}, L, \mathsf{wData})$$

This circuit takes as input the current CPU state state and a bit rData. Looking ahead the bit rData will be read from the memory location that was requested by the previous CPU step. The circuit outputs an updated state state', a read or write bit $\mathsf{R/W}$, the next location to read/write from $L \in [M]$, and a bit wData to write into that location ($\mathsf{wData} = \perp$ when reading). The sequence of locations and read/write values collectively form what is known as the *access pattern*, namely $\mathsf{MemAccess} = \{(\mathsf{R/W}^\tau, L^\tau, \mathsf{wData}^\tau) : \tau = 1, \ldots, t\}$.

Note that in the description above without loss of generality we have made some simplifying assumptions. We assume that each CPU-step circuit always reads from or write some location in memory. This is easy to implement via a dummy read and write step. Moreover, we assume that the instructions of the program itself are hardwired into the CPU-step circuits.

Representing RAM Computation by CPU-Step Circuits. The computation $P^D(x)$ starts with the initial state set as $\mathsf{state}_1 = x$. In each step $\tau \in \{1, \ldots t\}$, the computation proceeds as follows: If $\tau = 1$ or $\mathsf{R/W}^{\tau-1} = \mathsf{write}$, then $\mathsf{rData}^\tau := 0$; otherwise $\mathsf{rData}^\tau := D[L^{\tau-1}]$. Next it executes the CPU-Step Circuit $C^P_{\mathsf{CPU}}(\mathsf{state}^\tau, \mathsf{rData}^\tau) = (\mathsf{state}^{\tau+1}, \mathsf{R/W}^\tau, L^\tau, \mathsf{wData}^\tau)$. If $\mathsf{R/W}^\tau = \mathsf{write}$, then set $D[L^\tau] = \mathsf{wData}^\tau$. Finally, when $\tau = t$, then $\mathsf{state}^{\tau+1}$ is the output of the program.

3.2.2.2 Oblivious Transfer

[AIR01; NP01; HK12] gave two-message oblivious transfer (OT) protocols. We describe the definition below and refer the reader to [AIR01; NP01; HK12] for details.

[3] In general, the distinction between what to include in the program P, the memory data D and the short input x can be somewhat arbitrary. However as motivated by our applications we will typically be interested in a setting where the data D is large while the size of the program $|P|$ and input length m is small.

Definition 3.2.1 (Two-Message Oblivious Transfer)**.** *A two-message oblivious transfer protocol* $\mathsf{OT} = (\mathsf{OT_1}, \mathsf{OT_2}, \mathsf{OT_3})$ *is a protocol between a sender* S *and a receiver* R *where* S *gets as input two strings* s_1, s_2 *of equal length and* R *gets as input a choice bit* $x \in \{0,1\}$*. The algorithms have the following syntax:*

- $(m_1, \mathit{secret}) \leftarrow \mathsf{OT_1}(1^\kappa, x)$*: It takes as input the security parameter* 1^κ *and receiver's choice bit* $x \in \{0,1\}$ *and outputs the first OT message* m_1 *(sent by the receiver) and receiver's secret state secret.*
- $m_2 \leftarrow \mathsf{OT_2}(m_1, s_0, s_1)$*: It takes as input the first OT message and the sender's input* (s_0, s_1)*, and outputs the second OT message* m_2 *(sent back to the receiver).*
- $s \leftarrow \mathsf{OT_3}(m_2, \mathit{secret})$*: It takes* m_2 *and secret as input, and outputs a string* s*.*

The following conditions are satisfied:

- ***Perfect Correctness.*** *For all security parameter* κ*, sender input strings* (s_1, s_2) *of equal length, and receiver's choice bit* x*, let* $(m_1, \mathit{secret}) \leftarrow \mathsf{OT_1}(1^\kappa, x)$*,* $m_2 \leftarrow \mathsf{OT_2}(m_1, s_0, s_1)$*, and* $s \leftarrow \mathsf{OT_3}(m_2, \mathit{secret})$*, then it holds that*

$$\Pr[s = s_x] = 1.$$

- ***Receiver Security.*** *The following two distributions are computationally indistinguishable:*

$$\mathsf{OT_1}(1^\kappa, 0) \overset{c}{\approx} \mathsf{OT_1}(1^\kappa, 1).$$

- ***Sender Security.*** *There exists a PPT simulator* OTSim *such that for all sender input strings* (s_1, s_2) *of equal length and receiver's choice bit* x*, and any first message* m_1 *in the support of* $\mathsf{OT_1}(1^\kappa, x)$*, the following two distributions are statistically close:*

$$\mathsf{OT_2}(m_1, s_0, s_1) \overset{s}{\approx} \mathsf{OTSim}(1^\kappa, x, s_x, m_1).$$

We described the above definition with respect to one OT, but the same formalism naturally extends to support multiple parallel executions of OT. We use the following shorthand notations (generalizing the above notions) to run multiple parallel executions. Let $\mathsf{Keys} = ((\mathsf{Key}_{1,0}, \mathsf{Key}_{1,1}), \ldots, (\mathsf{Key}_{n,0}, \mathsf{Key}_{n,1}))$ be a list of n string-pairs, and $x \in \{0,1\}^n$ be an n-bit choice string. Then we define

- $(m_1, \mathrm{secret}) \leftarrow \mathsf{OT_1}(1^\kappa, x) = (\mathsf{OT_1}(1^\kappa, x_1), \ldots, \mathsf{OT_1}(1^\kappa, x_n))$.
- $m_2 \leftarrow \mathsf{OT_2}(m_1, \mathsf{Keys}) = \left(\mathsf{OT_2}(m_{1,1}, \mathsf{Key}_{1,0}, \mathsf{Key}_{1,1}), \cdots, \mathsf{OT_2}(m_{1,n}, \mathsf{Key}_{n,0}, \mathsf{Key}_{n,1})\right)$.
- $\mathsf{Labels} \leftarrow \mathsf{OT_3}(m_2, \mathrm{secret}) = (\mathsf{OT_3}(m_{2,1}, \mathrm{secret}_1), \ldots, \mathsf{OT_3}(m_{2,n}, \mathrm{secret}_n))$.

In the above $m_1 = (m_{1,1}, \ldots, m_{1,n})$, $m_2 = (m_{2,1}, \ldots, m_{2,n})$, $\text{secret} = (\text{secret}_1, \ldots, \text{secret}_n)$. Correctness guarantees that $\mathsf{Labels} = \mathsf{Keys}_x = \left(\mathsf{Key}_{1,x_1}, \ldots, \mathsf{Key}_{n,x_n}\right)$.

Moreover, we will use two important properties of the oblivious transfer [NP01] for our applications: (1) Security holds for multiple second OT messages with regard to the same first OT message. This will be crucial for extending NISC for RAM to support multiple senders with the same receiver. (2) The second OT message is re-randomizable. This will be crucial for the application of multi-hop homomorphic encryption for RAM.

3.2.3 Our Model

Suppose the receiver owns a large confidential database $D \in \{0,1\}^M$. It first publishes a *short* message, denoted by m_1, which hides D. Afterwards, if a sender wants to run a RAM program P (with input x) on D, it can send a single message m_2 to the receiver. For security we require that m_2 only reveals the output $P^D(x)$ and the memory access pattern $\mathsf{MemAccess}$ of the execution to the receiver. We require that once m_1 is published, the computational cost of both the sender (in computing m_2) and the receiver (in evaluation), as well as the size of m_2, should grow only with the running time of the RAM computation and the size of m_1, and is independent of the size of D.

Moreover, the sender can run a sequence of programs on a persistent database by sending one message per program to the receiver. Finally, the receiver can run the protocol in parallel with multiple senders, where the same m_1 is used. For ease of exposition, below we will describe the setting of one single sender executing one program with the receiver. We provide details on above extensions in Section 3.2.7.

Definition 3.2.2 (Non-Interactive Secure RAM Computation)**.** *A non-interactive secure RAM computation scheme* $\mathsf{NISC\text{-}RAM} = (\mathsf{Setup}, \mathsf{EncData}, \mathsf{EncProg}, \mathsf{Dec})$ *has the following syntax. It is a two-party protocol between a receiver holding a large secret database D and a sender holding secret program P of running time t and a short input x.*

- ***Setup.*** $\mathsf{crs} \leftarrow \mathsf{Setup}(1^\kappa)$.
  On input the security parameter 1^κ, it outputs a common reference string.

- ***Database Encryption.*** $(m_1, \widetilde{D}) \leftarrow \mathsf{EncData}(\mathsf{crs}, D)$.
  On input the common reference string crs *and a database $D \in \{0,1\}^M$, it outputs a message m_1 and a secret state $\widetilde{D}$. The receiver publishes m_1 as the short message corresponding to D.*

- ***Program Encryption.*** $m_2 \leftarrow \mathsf{EncProg}(\mathsf{crs}, m_1, (P, x, t))$.
  It takes as input the crs*, a message m_1, a RAM program P with input x and maximum run-time t. It then outputs another message m_2. The sender sends the message m_2.*

- ***Decryption.*** $y \leftarrow \mathsf{Dec}^{\widetilde{D}}(\mathsf{crs}, m_2)$.
  The procedure Dec *is modeled as a RAM program that can read and write to arbitrary*

locations of its database initially containing $\widetilde{D}$. This procedure is run by the receiver. On input the crs *and m_2, it outputs y.*

The following conditions are satisfied:

- ***Correctness.*** *For every database $D \in \{0,1\}^M$ where $M = \mathsf{poly}(\kappa)$ for any polynomial function $\mathsf{poly}(\cdot)$, for every RAM program (P, x, t), it holds that*

$$\Pr\left[\mathsf{Dec}^{\widetilde{D}}(\mathsf{crs}, m_2) = P^D(x)\right] = 1,$$

where $\mathsf{crs} \leftarrow \mathsf{Setup}(1^\kappa)$, $(m_1, \widetilde{D}) \leftarrow \mathsf{EncData}(\mathsf{crs}, D)$, $m_2 \leftarrow \mathsf{EncProg}(\mathsf{crs}, m_1, (P, x, t))$.

- ***Receiver Privacy.*** *For every pair of databases $D_0 \in \{0,1\}^M, D_1 \in \{0,1\}^M$ where M is polynomial in κ, for every crs in the support of $\mathsf{Setup}(1^\kappa)$, let $(m_0, \widetilde{D}_0) \leftarrow \mathsf{EncData}(\mathsf{crs}, D_0)$, $(m_1, \widetilde{D}_1) \leftarrow \mathsf{EncData}(\mathsf{crs}, D_1)$. Then it holds that*

$$(\mathsf{crs}, m_0) \overset{c}{\approx} (\mathsf{crs}, m_1).$$

- ***Sender Privacy.*** *There exists a PPT simulator $\mathsf{niscSim}$ such that for every database $D \in \{0,1\}^M$ where $M = \mathsf{poly}(\kappa)$ for any polynomial function $\mathsf{poly}(\cdot)$, and for every RAM program (P, x, t), let $y = P^D(x)$ be the output of the program, and $\mathsf{MemAccess}$ be the memory access pattern, then it holds that*

$$(\mathsf{crs}, D, (m_1, \widetilde{D}), m_2) \overset{c}{\approx} \mathsf{niscSim}(1^\kappa, D, (y, \mathsf{MemAccess}))$$

where $\mathsf{crs} \leftarrow \mathsf{Setup}(1^\kappa)$, $(m_1, \widetilde{D}) \leftarrow \mathsf{EncData}(\mathsf{crs}, D)$ and $m_2 \leftarrow \mathsf{EncProg}(\mathsf{crs}, m_1, (P, x, t))$.

- ***Efficiency.*** *The length of m_1 is a fixed polynomial in κ independent of the size of the database. Moreover, the algorithm $\mathsf{EncData}$ runs in time $M \cdot \mathsf{poly}(\kappa, \log M)$, $\mathsf{EncProg}$ and Dec run in time $t \cdot \mathsf{poly}(\kappa, \log M)$.*

3.2.4 Construction of Non-Interactive Secure RAM Computation

Overview. We first give an overview of the construction. For ease of exposition, consider a read-only program where each CPU step outputs the next location to be read based on the value read from last location.

We first describe the $\mathsf{EncProg}$ procedure. As already mentioned in technical overview (see Section 3.2.1), our construction is based on high level ideas of garbled RAM (introduced by Lu and Ostrovsky [LO13]) to make sender and receiver complexity grow only with the running time of the program. In particular, the sender would generate a garbled RAM program consisting of a sequence of t garbled *step circuits*. Similar to the RAM computation model described in Section 3.2.2.1, every step circuit takes as input the current CPU state

and the last read bit and outputs the updated state and the next read location, say L. Note that the next step circuit would take the new value read from database as input.

The main challenge in program garbling is revealing the correct labels for the next circuit based on the value of $D[L]$. Moreover, it is crucial for garble circuit security that the receiver does not learn the label corresponding to $1 - D[L]$. Prior works [LO13; GHL+14; GLOS15; GLO15] proposed several different solutions to the above problem. Here we present a new and arguably simpler solution for achieving this using laconic oblivious transfer.

Let digest be the hash value of D that would be fed into the first step circuit and passed along the sequence of circuits. That is, each circuit would take this digest as input and also output the correct input labels corresponding to the digest for the next circuit. Now, to transfer the correct label corresponding to the value in the database, a step circuit would output a laconic OT ciphertext (using algorithm Send) that encrypts the input keys of the next step circuit and uses digest as the hash value. Looking ahead, when the receiver evaluates the step circuit which outputs the laconic OT ciphertext, he can use D to decrypt it to obtain the correct labels (using the procedure $\mathsf{Receive}$ of laconic OT).

We show that the sender privacy follows from the sender privacy of laconic OT and security of circuit garbling. In order to achieve receiver privacy, the receiver does not publish digest in the clear, but instead, the labels for digest of the first step circuit are transferred from the sender to the receiver via a two-message OT. In particular, the $\mathsf{EncData}$ procedure outputs the first OT message of digest, and $\mathsf{EncProg}$ will output the garbled step circuits along with the second OT message for digest's labels.

Finally, note that a general program can also write to the database, in which case we need to update the database as well as the step circuits need to know the updated digest for the correctness of laconic OT and future reads/writes. This is achieved via the updatability property of the laconic OT which allows a sender to generate a ciphertext that allows the receiver to learn messages corresponding to the updated digest. In our case, the messages encrypted would be the input digest keys of the next step circuit.

Next, we give a more formal construction of our scheme.

Construction. Let $\ell OT = (\mathsf{crsGen}, \mathsf{Hash}, \mathsf{Send}, \mathsf{Receive}, \mathsf{SendWrite}, \mathsf{ReceiveWrite})$ be an updatable laconic OT protocol as per Definition 2.2.2. Let $\mathsf{OT} = (\mathsf{OT}_1, \mathsf{OT}_2, \mathsf{OT}_3)$ be a two-message secure oblivious transfer, and let $\mathsf{GC} = (\mathsf{GCircuit}, \mathsf{GEval})$ be a circuit garbling scheme. The non-interactive secure RAM computation scheme $\mathsf{NISC\text{-}RAM} = (\mathsf{Setup}, \mathsf{EncData}, \mathsf{EncProg}, \mathsf{Dec})$ is constructed as follows.

- **Setup.** $\mathsf{crs} \leftarrow \mathsf{Setup}(1^\kappa)$.
  The set up algorithm is described in Figure 3.1. It generates the common reference string for the updatable laconic OT scheme.

- **Database Encryption.** $(m_1, \widetilde{D}) \leftarrow \mathsf{EncData}(\mathsf{crs}, D)$.
  The algorithm is formally described in Figure 3.2. It hashes the database D using

Setup. $\mathsf{crs} \leftarrow \mathsf{Setup}(1^\kappa)$.
1. $\mathsf{crs} \leftarrow \mathsf{crsGen}(1^\kappa)$.
2. Output crs.

Figure 3.1: Setup procedure of NISC-RAM.

laconic OT Hash function and obtains digest. Then digest is encrypted using the OT_1 procedure of two-message OT protocol.

Database Encryption. $(m_1, \widetilde{D}) \leftarrow \mathsf{EncData}(\mathsf{crs}, D)$.
1. $(\mathsf{digest}, \hat{\mathsf{D}}) \leftarrow \mathsf{Hash}(\mathsf{crs}, D)$.
2. $(m_1, \text{secret}) \leftarrow \mathsf{OT}_1(1^\kappa, \mathsf{digest})$.
3. Output $\Big(m_1, \widetilde{D} = (\mathsf{digest}, \hat{\mathsf{D}}, \text{secret})\Big)$.

Figure 3.2: Database encryption procedure of NISC-RAM.

- **Program Encryption.** $m_2 \leftarrow \mathsf{EncProg}(\mathsf{crs}, m_1, (P, x, t))$.
  The program encryption procedure is formally described in Figure 3.3. As mentioned above, it generates t garbled *step circuits* $\{\widetilde{\mathsf{C}}_\tau^{\mathsf{step}}\}_{\tau=1}^t$, where every step circuit implements the functionality of a CPU-step circuit. We describe the structure of a step circuit C^{step} below. The program encryption also consists of the second OT message corresponding to the short message m_1 of the receiver (for digest) where the sender's messages consist of the input keys for the first garbled circuit. Finally, it also outputs the keys for decrypting the output of the last step circuit.

Program Encryption. $m_2 \leftarrow \mathsf{EncProg}(\mathsf{crs}, m_1, (P, x, t))$.
1. **Generate the garbled program for** P: Generate garbled circuits $\{\widetilde{\mathsf{C}}_\tau^{\mathsf{step}}\}_{\tau=1}^t$.
   a) Sample $\mathsf{stateKeys}^\tau, \mathsf{dataKeys}^\tau, \mathsf{digestKeys}^\tau$ for each $\tau \in \{1, \ldots, t+1\}$.
   b) For each $\tau \in \{1, \ldots, t\}$
   $$\widetilde{\mathsf{C}}_\tau^{\mathsf{step}} \leftarrow \mathsf{GCircuit}\left(1^\kappa, \mathsf{C}^{\mathsf{step}}[\mathsf{crs}, P, \mathsf{Keys}^{\tau+1}], \mathsf{Keys}^\tau\right),$$
   where $\mathsf{Keys}^\tau = (\mathsf{stateKeys}^\tau, \mathsf{dataKeys}^\tau, \mathsf{digestKeys}^\tau)$.
   c) For $\tau = 1$, embed labels $\mathsf{dataKeys}_0^1$ and $\mathsf{stateKeys}_x^1$ in $\widetilde{\mathsf{C}}_1^{\mathsf{step}}$.
2. Compute $\mathsf{L} \leftarrow \mathsf{OT}_2\left(m_1, \mathsf{digestKeys}^1\right)$.
3. Output $m_2 = \Big(\mathsf{L}, \{\widetilde{\mathsf{C}}_\tau^{\mathsf{step}}\}_{\tau=1}^t, \mathsf{stateKeys}^{t+1}\Big)$.

Figure 3.3: Program encryption procedure of NISC-RAM.

Now we elaborate on the logic of a step circuit. The pseudocode of a step circuit $\mathsf{C}^{\mathsf{step}}$ is formally described in Figure 3.4, and the structure is illustrated in Figure 3.5.

Hardwired Parameters: $[\mathsf{crs}, P, \mathsf{nextKeys} = (\mathsf{stateKeys}, \mathsf{dataKeys}, \mathsf{digestKeys})]$.
Input: $(\mathsf{state}, \mathsf{rData}, \mathsf{digest})$.

$(\mathsf{state}', \mathsf{R/W}, L, \mathsf{wData}) := C^P_{\mathsf{CPU}}(\mathsf{state}, \mathsf{rData})$.

if $\mathsf{R/W} = \mathsf{read}$ **then**
  $\mathsf{e}_{\mathsf{data}} \leftarrow \mathsf{Send}(\mathsf{crs}, \mathsf{digest}, L, \mathsf{dataKeys})$.
  return $\big((\mathsf{stateKeys}_{\mathsf{state}'}, \mathsf{e}_{\mathsf{data}}, \mathsf{digestKeys}_{\mathsf{digest}}), \mathsf{R/W}, L\big)$.
else
  $\mathsf{e}_{\mathsf{digest}} \leftarrow \mathsf{SendWrite}(\mathsf{crs}, \mathsf{digest}, L, \mathsf{wData}, \mathsf{digestKeys})$.
  return $((\mathsf{stateKeys}_{\mathsf{state}'}, \mathsf{dataKeys}_0, \mathsf{e}_{\mathsf{digest}}, \mathsf{wData}), \mathsf{R/W}, L)$.

Figure 3.4: Pseudocode of a step circuit $\mathsf{C}^{\mathsf{step}}[\mathsf{crs}, P, \mathsf{nextKeys}]$.

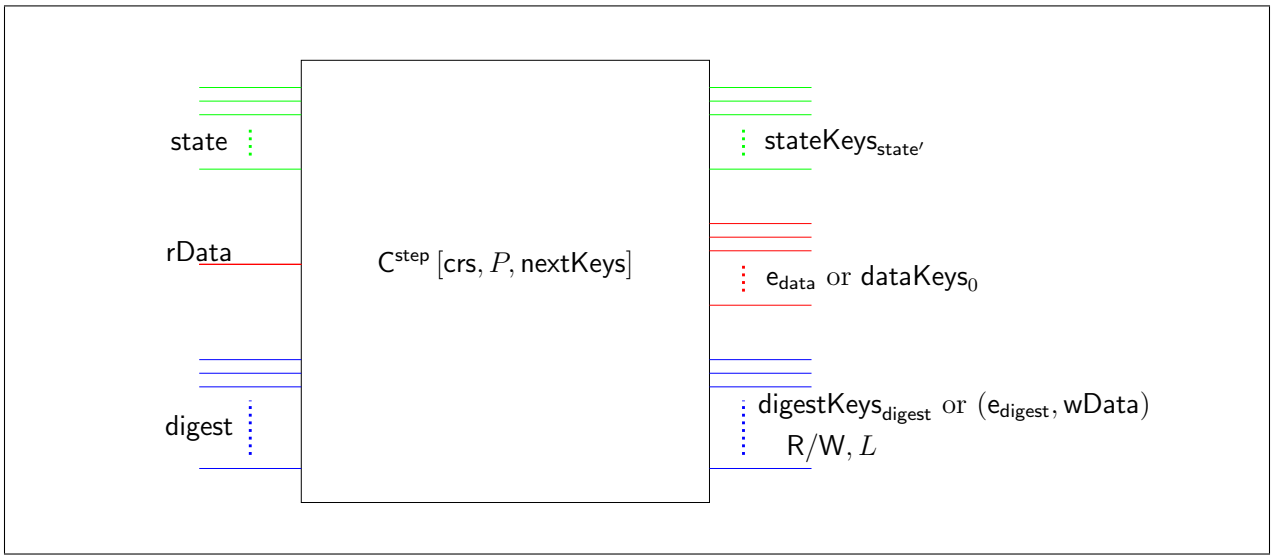


Figure 3.5: Input-output behavior of a step circuit $\mathsf{C}^{\mathsf{step}}[\mathsf{crs}, P, \mathsf{nextKeys}]$.

The input of a step circuit can be partitioned into $(\mathsf{state}, \mathsf{rData}, \mathsf{digest})$, where state is the current CPU state, rData is the bit read from the database, and digest is the up-to-date digest of the database. If the previous step is a write, then $\mathsf{rData} = 0$. The program encryption outputs garbled circuits for these step circuits, hence, the first step of $\mathsf{EncProg}$ is to pick the input keys for all the circuits. The τ-th step circuit $\mathsf{C}^{\mathsf{step}}_\tau$ has hardwired in it the input keys $\mathsf{nextKeys} = (\mathsf{stateKeys}, \mathsf{dataKeys}, \mathsf{digestKeys})$ for the next step circuit $\mathsf{C}^{\mathsf{step}}_{\tau+1}$.

The logic of the step circuit is as follows: It first computes the new $(\mathsf{state}', \mathsf{R/W}, L, \mathsf{wData})$. Then, in the case of a "read" it outputs $\mathsf{stateKeys}$ corresponding to state', labels for rData via laconic OT procedure $\mathsf{Send}(\cdot)$, and $\mathsf{digestKeys}$ corresponding to digest. The case of a write is similar, but now the labels of new updated digest are transferred via laconic OT procedure $\mathsf{SendWrite}(\cdot)$.

- **Decryption:** $y \leftarrow \mathsf{Dec}^{\widetilde{D}}(\mathsf{crs}, m_2)$.
  The decryption procedure is described in Figure 3.6. At a high level the receiver evaluates the garbled step circuits one by one from $\widetilde{\mathsf{C}}_1^{\mathsf{step}}$ to $\widetilde{\mathsf{C}}_t^{\mathsf{step}}$, and uses the database to decrypt ℓOT ciphertexts between two consecutive circuits. The output of the last step circuit can be decrypted using $\mathsf{stateKeys}^{t+1}$ and hence y is obtained.

Decryption. $y \leftarrow \mathsf{Dec}^{\widetilde{D}}(\mathsf{crs}, m_2)$.

1. Parse $\widetilde{D} = (\mathsf{digest}, \hat{\mathsf{D}}, \text{secret})$.
2. Parse $m_2 = \left(\mathsf{L}, \{\widetilde{\mathsf{C}}_\tau^{\mathsf{step}}\}_{\tau=1}^t, \mathsf{stateKeys}^{t+1}\right)$.
3. Compute $\mathsf{digestLabels}^1 \leftarrow \mathsf{OT}_3(\mathsf{L}, \text{secret})$.
4. Parse $\widetilde{\mathsf{C}}_1^{\mathsf{step}} = (\widetilde{\mathsf{C}}_1^{\mathsf{step}}, \mathsf{dataLabels}^1, \mathsf{stateLabels}^1)$.
5. For $\tau = 1$ to t do the following:

   $(X, \mathsf{R/W}, L) := \mathsf{GEval}\left(\widetilde{\mathsf{C}}_\tau^{\mathsf{step}}, (\mathsf{stateLabels}^\tau, \mathsf{dataLabels}^\tau, \mathsf{digestLabels}^\tau)\right)$.

   if $\mathsf{R/W} = \mathsf{read}$ **then**
   Parse $X = (\mathsf{stateLabels}^{\tau+1}, \mathsf{e}_{\mathsf{data}}, \mathsf{digestLabels}^{\tau+1})$
   $\mathsf{dataLabels}^{\tau+1} = \mathsf{Receive}^{\hat{\mathsf{D}}}(\mathsf{crs}, \mathsf{e}_{\mathsf{data}}, L)$
   else
   Parse $X = (\mathsf{stateLabels}^{\tau+1}, \mathsf{dataLabels}^{\tau+1}, \mathsf{e}_{\mathsf{digest}}, \mathsf{wData})$
   $\mathsf{digestLabels}^{\tau+1} = \mathsf{ReceiveWrite}^{\hat{\mathsf{D}}}(\mathsf{crs}, L, \mathsf{wData}, \mathsf{e}_{\mathsf{digest}})$
6. Use $\mathsf{stateKeys}^{t+1}$ to decode $\mathsf{stateLabels}^{t+1}$ and obtain y.

Figure 3.6: Decryption procedure of NISC-RAM.

More precisely, the receiver first obtains the digestLabels for the first step circuit by running OT_3. Note that the first garbled step circuit already has labels for the rData and state embedded. Hence the receiver can obtain all the labels for the first step circuit and evaluate it. Then the receiver executes the circuits $\{\widetilde{\mathsf{C}}_\tau^{\mathsf{step}}\}_{\tau=1}^t$ one by one, and learns the labels for the next circuit by running the receiver algorithms of laconic OT on its database.

3.2.5 Correctness

For correctness, we require that for every database $D \in \{0,1\}^M$, for every RAM program (P, x, t), it holds that

$$\Pr\left[\mathsf{Dec}^{\widetilde{D}}(\mathsf{crs}, m_2) = P^D(x)\right] = 1,$$

where $\mathsf{crs} \leftarrow \mathsf{Setup}(1^\kappa)$, $(m_1, \widetilde{D}) \leftarrow \mathsf{EncData}(\mathsf{crs}, D)$, $m_2 \leftarrow \mathsf{EncProg}(\mathsf{crs}, m_1, (P, x, t))$. Correctness follows from Lemma 3.2.4 that we will prove below.

Claim 3.2.3. *The first garbled step circuit* $\widetilde{\mathsf{C}}_1^{\mathsf{step}}$ *gets evaluated on* $(x, 0, \mathsf{digest})$*, where* $(\mathsf{digest}, \hat{\mathsf{D}}) = \mathsf{Hash}(\mathsf{crs}, D)$*.*

Proof. Since $(m_1, \mathrm{secret}) \leftarrow \mathsf{OT}_1(1^\kappa, \mathsf{digest})$, $\mathsf{L} \leftarrow \mathsf{OT}_2\left(m_1, \mathsf{digestKeys}^1\right)$, and $\mathsf{digestLabels}^1 \leftarrow \mathsf{OT}_3(\mathsf{L}, \mathrm{secret})$, by correctness of OT, $\mathsf{digestLabels}^1 = \mathsf{digestKeys}^1_{\mathsf{digest}}$. Moreover, $\widetilde{\mathsf{C}}_1^{\mathsf{step}}$ already has labels $\mathsf{stateKeys}^1_x$ and $\mathsf{dataKeys}^1_0$ embedded in it, by correctness of the circuit garbling scheme, $\widetilde{\mathsf{C}}_1^{\mathsf{step}}$ gets evaluated on $(x, 0, \mathsf{digest})$. □

Lemma 3.2.4. *Consider the execution of* $P^D(x)$*. Let* $(\mathsf{state}^\tau, \mathsf{rData}^\tau)$ *be the input to the* τ*-th CPU step. Let* D^τ *be the database at the beginning of step* τ*, and let* $(\mathsf{digest}^\tau, \hat{\mathsf{D}}^\tau) = \mathsf{Hash}(\mathsf{crs}, D^\tau)$*. During the* Dec *procedure, for every* $\tau \in [t]$*,* $\widetilde{\mathsf{C}}_\tau^{\mathsf{step}}$ *is evaluated on inputs* $(\mathsf{state}^\tau, \mathsf{rData}^\tau, \mathsf{digest}^\tau)$*. Moreover, the state of the database held by the receiver at the beginning of evaluating* $\widetilde{\mathsf{C}}_\tau^{\mathsf{step}}$ *is* $\hat{\mathsf{D}}^\tau$*.*

Proof. We will prove this lemma by induction on τ. The base case follows from Claim 3.2.3. Assume that the lemma holds for $\tau = \rho$, then we prove that the lemma holds for $\rho + 1$ in the following. We know that $(\hat{\mathsf{D}}^\rho, \mathsf{digest}^\rho) = \mathsf{Hash}(\mathsf{crs}, D^\rho)$, and that $\widetilde{\mathsf{C}}_\rho^{\mathsf{step}}$ is executed on $(\mathsf{state}^\rho, \mathsf{rData}^\rho, \mathsf{digest}^\rho)$. By correctness of GC, $\widetilde{\mathsf{C}}_\rho^{\mathsf{step}}$ implements its code of a CPU step, namely $(\mathsf{state}', \mathsf{R/W}, L, \mathsf{wData}) = C^P_{\mathsf{CPU}}(\mathsf{state}^\rho, \mathsf{rData}^\rho)$. Also notice that $\mathsf{nextKeys} = (\mathsf{stateKeys}, \mathsf{dataKeys}, \mathsf{digestKeys})$ hardwired in $\widetilde{\mathsf{C}}_\rho^{\mathsf{step}}$ are the input keys for $\widetilde{\mathsf{C}}_{\rho+1}^{\mathsf{step}}$. There are two cases:

- $\mathsf{R/W} = \mathsf{read}$: In this case, it follows directly from the Dec procedure that $\mathsf{stateLabels}^{\rho+1} = \mathsf{stateKeys}_{\mathsf{state}'}$ and $\mathsf{digestLabels}^{\rho+1} = \mathsf{digestKeys}_{\mathsf{digest}}$. Since $\mathsf{e}_{\mathsf{data}} \leftarrow \mathsf{Send}(\mathsf{crs}, \mathsf{digest}^\rho, L, \mathsf{dataKeys})$ and $\mathsf{dataLabels}^{\rho+1} = \mathsf{Receive}^{\hat{\mathsf{D}}^\rho}(\mathsf{crs}, \mathsf{e}_{\mathsf{data}}, L)$, by correctness of the ℓOT scheme, $\mathsf{dataLabels}^{\rho+1} = \mathsf{dataKeys}_{D^\rho[L]}$. Hence $\widetilde{\mathsf{C}}_{\rho+1}^{\mathsf{step}}$ is evaluated on inputs $(\mathsf{state}', D^\rho[L], \mathsf{digest})$, which is exactly $(\mathsf{state}^{\rho+1}, \mathsf{rData}^{\rho+1}, \mathsf{digest}^{\rho+1})$. And $(\hat{\mathsf{D}}^\rho, \mathsf{digest}^\rho)$ remains unchanged.
- $\mathsf{R/W} = \mathsf{write}$: In this case, it follows from the Dec procedure that $\mathsf{stateLabels}^{\rho+1} = \mathsf{stateKeys}_{\mathsf{state}'}$ and $\mathsf{dataLabels}^{\rho+1} = \mathsf{dataKeys}_0$. Since $\mathsf{e}_{\mathsf{digest}} \leftarrow \mathsf{SendWrite}(\mathsf{crs}, \mathsf{digest}^\rho, L, \mathsf{wData}, \mathsf{digestKeys})$ and $\mathsf{digestLabels}^{\rho+1} = \mathsf{ReceiveWrite}^{\hat{\mathsf{D}}^\rho}(\mathsf{crs}, L, \mathsf{wData}, \mathsf{e}_{\mathsf{digest}})$, by correctness of the ℓOT scheme, $\mathsf{digestLabels}^{\rho+1} = \mathsf{digestKeys}_{\mathsf{digest}'}$ where $(\hat{\mathsf{D}}', \mathsf{digest}') = \mathsf{Hash}(\mathsf{crs}, D')$ for an updated database D' (D' is identical to D^ρ except that $D'[L] = \mathsf{wData}$). Hence $\widetilde{\mathsf{C}}_{\rho+1}^{\mathsf{step}}$ is evaluated on inputs $(\mathsf{state}', 0, \mathsf{digest}')$, which is exactly $(\mathsf{state}^{\rho+1}, \mathsf{rData}^{\rho+1}, \mathsf{digest}^{\rho+1})$. In addition, $(\hat{\mathsf{D}}^\rho, \mathsf{digest}^\rho)$ gets updated to $(\hat{\mathsf{D}}', \mathsf{digest}')$, which is exactly $(\hat{\mathsf{D}}^{\rho+1}, \mathsf{digest}^{\rho+1})$.

□

3.2.6 Security Proof

In this section we prove sender privacy and receiver privacy as defined in Section 3.2.3 under the decisional Diffie-Hellman (DDH) assumption. The receiver privacy follows directly from the receiver security of OT. Below we prove sender privacy by describing a PPT simulator $\mathsf{niscSim}$ such that for every database $D \in \{0,1\}^M$ where M is polynomial in κ, and for every RAM program (P, x, t), let $y = P^D(x)$ be the output of the program, and $\mathsf{MemAccess}$ be the memory access pattern, then it holds that

$$\left(\mathsf{crs}, (m_1, \widetilde{D}), \mathsf{EncProg}(\mathsf{crs}, m_1, (P, x, t))\right) \overset{c}{\approx} \left(\mathsf{crs}, (m_1, \widetilde{D}), \mathsf{niscSim}(\mathsf{crs}, m_1, D, y, \mathsf{MemAccess})\right),$$

where $\mathsf{crs} \leftarrow \mathsf{Setup}(1^\kappa)$, $(m_1, \widetilde{D}) \leftarrow \mathsf{EncData}(\mathsf{crs}, D)$. Notice that this definition is slightly different from the definition in Section 3.2.3, but in the semi-honest case it implies a simulator as defined in Section 3.2.3

1. Sample input keys $(\mathsf{stateKeys}^{t+1}, \mathsf{dataKeys}^{t+1}, \mathsf{digestKeys}^{t+1})$ for $\mathsf{C}^{\mathsf{step}}$.
2. Parse $\mathsf{MemAccess}$ as $\{(\mathsf{R/W}^\tau, L^\tau, \mathsf{wData}^\tau) : \tau \in [t]\}$, where $(\mathsf{R/W}^\tau, L^\tau, \mathsf{wData}^\tau)$ is partial output of the τ-th CPU step circuit. Compute $(\mathsf{rData}^\tau, D^\tau, \mathsf{digest}^\tau)$ at the beginning of step τ for every $\tau \in [t+1]$.
3. Compute $(\mathsf{stateLabels}^{t+1}, \mathsf{dataLabels}^{t+1}, \mathsf{digestLabels}^{t+1})$:
   $\mathsf{stateLabels}^{t+1} \leftarrow \mathsf{stateKeys}^{t+1}_{y}$.
   $\mathsf{digestLabels}^{t+1} \leftarrow \mathsf{digestKeys}^{t+1}_{\mathsf{digest}^{t+1}}$.
   $\mathsf{dataLabels}^{t+1} \leftarrow \mathsf{dataKeys}^{t+1}_{\mathsf{rData}^{t+1}}$.
4. For $\tau = t$ downto 1, proceed as follows:
   if $\mathsf{R/W}^\tau = \mathsf{read}$ **then**
   $\mathsf{e}_{\mathsf{data}} \leftarrow \ell\mathsf{OTSim}\left(\mathsf{crs}, D^\tau, L^\tau, \mathsf{dataLabels}^{\tau+1}\right)$.
   $X \leftarrow (\mathsf{stateLabels}^{\tau+1}, \mathsf{e}_{\mathsf{data}}, \mathsf{digestLabels}^{\tau+1})$.
   else
   $\mathsf{e}_{\mathsf{digest}} \leftarrow \ell\mathsf{OTSimWrite}\left(\mathsf{crs}, D^\tau, L^\tau, \mathsf{wData}^\tau, \mathsf{digestLabels}^{\tau+1}\right)$.
   $X \leftarrow (\mathsf{stateLabels}^{\tau+1}, \mathsf{dataLabels}^{\tau+1}, \mathsf{e}_{\mathsf{digest}}, \mathsf{wData}^\tau)$.

   $\left(\widetilde{\mathsf{C}}^{\mathsf{step}}_\tau, \mathsf{stateLabels}^\tau, \mathsf{dataLabels}^\tau, \mathsf{digestLabels}^\tau\right) \leftarrow \mathsf{CircSim}\left(1^\kappa, \mathsf{C}^{\mathsf{step}}, (X, \mathsf{R/W}^\tau, L^\tau)\right)$.
5. $\mathsf{L} \leftarrow \mathsf{OT}_2\left(m_1, (\mathsf{digestLabels}^1, \mathsf{digestLabels}^1)\right)$.
6. Output $\left(\mathsf{L}, \{\widetilde{\mathsf{C}}^{\mathsf{step}}_\tau\}_{\tau=1}^t, \mathsf{stateKeys}^{t+1}\right)$.

We show that the above simulation is indistinguishable from the real execution through a sequence of hybrids where the first hybrid outputs the real execution and the last hybrid outputs the simulated one.

- H_{2i} for $i \in \{0, 1, \ldots, t\}$: Notice that in the output, there are t garbled step circuits $\{\widetilde{\mathsf{C}}^{\mathsf{step}}_\tau\}^t_{\tau=1}$. In hybrid H_{2i}, the garbled step circuits from 1 to i are simulated while the remaining step circuits ($i+1$ to t) are generated honestly. Given the program, all the intermediate outputs of every step circuit can all be computed. Given the correct output of circuit $\mathsf{C}^{\mathsf{step}}_i$, the step circuits from 1 to i can be simulated one by one from the i-th to the first similarly as $\mathsf{niscSim}$. More formally, it proceeds as follows.

  1. Execute $P^D(x)$ to obtain $(\mathsf{R/W}^\tau, L^\tau, \mathsf{wData}^\tau)$ for every $\tau \in [t]$ and $\mathsf{state}^{t+1} = y$. Compute $(\mathsf{rData}^\tau, D^\tau, \mathsf{digest}^\tau)$ at the beginning of step τ for every $\tau \in [t+1]$.
  2. Generate the garble circuits $\{\widetilde{\mathsf{C}}^{\mathsf{step}}_\tau\}^t_{\tau=i+1}$ honestly (same as Step 1 in $\mathsf{EncProg}$).
  3. Let $\left(\mathsf{stateKeys}^{i+1}, \mathsf{dataKeys}^{i+1}, \mathsf{digestKeys}^{i+1}\right)$ be the input keys of $\widetilde{\mathsf{C}}^{\mathsf{step}}_{i+1}$.
  4. Compute $(\mathsf{stateLabels}^{i+1}, \mathsf{dataLabels}^{i+1}, \mathsf{digestLabels}^{i+1})$:
     $\mathsf{stateLabels}^{i+1} \leftarrow \mathsf{stateKeys}^{i+1}_{\mathsf{state}^{i+1}}$.
     $\mathsf{digestLabels}^{i+1} \leftarrow \mathsf{digestKeys}^{i+1}_{\mathsf{digest}^{i+1}}$.
     $\mathsf{dataLabels}^{i+1} \leftarrow \mathsf{dataKeys}^{i+1}_{\mathsf{rData}^{i+1}}$.
  5. For $\tau = i$ downto 1, proceed as in Step 4 of the simulator $\mathsf{niscSim}$.
  6. $\mathsf{L} \leftarrow \mathsf{OT}_2\left(m_1, (\mathsf{digestLabels}^1, \mathsf{digestLabels}^1)\right)$.
  7. Output $\left(\mathsf{L}, \{\widetilde{\mathsf{C}}^{\mathsf{step}}_\tau\}^t_{\tau=1}, \mathsf{stateKeys}^{t+1}\right)$.

- H_{2i+1} for $i \in \{0, \ldots, t-1\}$: Hybrid H_{2i+1} is identical to H_{2i} except that H_{2i+1} simulates $\widetilde{\mathsf{C}}^{\mathsf{step}}_{i+1}$ based on the real output of $\mathsf{C}^{\mathsf{step}}_{i+1}$. In particular, H_{2i+1} is the same as H_{2i} except that Steps 2, 3, 4 proceed as follows:

  2. Generate the garble circuits $\{\widetilde{\mathsf{C}}^{\mathsf{step}}_\tau\}^t_{\tau=i+2}$ honestly (same as Step 1 in $\mathsf{EncProg}$).
  3. Let $\left(\mathsf{stateKeys}^{i+2}, \mathsf{dataKeys}^{i+2}, \mathsf{digestKeys}^{i+2}\right)$ be the input keys of $\widetilde{\mathsf{C}}^{\mathsf{step}}_{i+2}$.
  4. **if** $\mathsf{R/W}^{i+1} = \mathsf{read}$ **then**
     $\mathsf{e}_{\mathsf{data}} \leftarrow \mathsf{Send}(\mathsf{crs}, \mathsf{digest}^{i+1}, L^{i+1}, \mathsf{dataKeys}^{i+2})$.
     $X \leftarrow (\mathsf{stateKeys}^{i+2}_{\mathsf{state}^{i+2}}, \mathsf{e}_{\mathsf{data}}, \mathsf{digestKeys}^{i+2}_{\mathsf{digest}})$.
     else
     $\mathsf{e}_{\mathsf{digest}} \leftarrow \mathsf{SendWrite}\left(\mathsf{crs}, \mathsf{digest}^{i+1}, L^{i+1}, \mathsf{wData}^{i+1}, \mathsf{digestKeys}^{i+2}\right)$.
     $X \leftarrow (\mathsf{stateKeys}^{i+1}_{\mathsf{state}^{i+1}}, \mathsf{dataKeys}^{i+1}_0, \mathsf{e}_{\mathsf{digest}}, \mathsf{wData}^{i+1})$.
     $\left(\widetilde{\mathsf{C}}^{\mathsf{step}}_{i+1}, \mathsf{stateLabels}^{i+1}, \mathsf{dataLabels}^{i+1}, \mathsf{digestLabels}^{i+1}\right) \leftarrow$
     $\mathsf{CircSim}\left(1^\kappa, \mathsf{C}^{\mathsf{step}}, (X, \mathsf{R/W}^{i+1}, L^{i+1})\right)$.

It is easy to see that H_0 is the output of the real execution, and H_{2t} is the simulated output. Now we prove that the consecutive hybrids are computationally indistinguishable. Below we prove that $\mathsf{H}_{2i} \overset{c}{\approx} \mathsf{H}_{2i+1} \overset{c}{\approx} \mathsf{H}_{2(i+1)}$ for every $i \in \{0, \ldots, t-1\}$. Since hybrid H_{2i+1} simulates $\widetilde{\mathsf{C}}^{\mathsf{step}}_{i+1}$ based on the real output of $\mathsf{C}^{\mathsf{step}}_{i+1}$, the output of $\widetilde{\mathsf{C}}^{\mathsf{step}}_{i+1}$ is identical for hybrids

H_{2i} and H_{2i+1}. That said, indistinguishibility of hybrids H_{2i} and H_{2i+1} follows from the garbled circuit security. Next, indistinguishability between H_{2i+1} and $\mathsf{H}_{2(i+1)}$ follows from the sender's privacy property of the updatable laconic OT since the laconic OT responses are simulated in $\mathsf{H}_{2(i+1)}$. This concludes the proof.

3.2.7 Extension

For simplicity of exposition, the protocol we described so far is for a single sender executing a single program with the receiver. It can be extended to the setting where a sender can execute a sequence of programs on a persistent database. Moreover, the message m_1 published by the receiver can be used by multiple senders, in which case the receiver maintains a different copy of the database for every sender.

Executing Multiple Programs on a Persistent Database. After receiving the first message m_1 from the receiver, a sender can run multiple programs on a persistent database (with initial content D) by sending one message per program to the receiver. For security we require only the output and the memory access pattern of every program execution are revealed to the receiver. We also require that once m_1 is published, the computational cost of both the sender and the receiver for every program should grow only with the running time of the RAM computation, and is independent of the size of D. The NISC-RAM scheme we constructed in Section 3.2.4 can be naturally extended to the multi-program setting. We explain the extension by describing the changes of EncProg and Dec procedures for the second program. Encryption and evaluation of more programs would follow analogously.

- EncProg: When encrypting the first program, the sender should store locally $\mathsf{digestKeys}^* = \mathsf{digestKeys}^{t+1}$. Then, when encrypting the second program, there are two changes in EncProg compared to encrypting the first program: (1) $\mathsf{digestKeys}^*$ is used as the digest keys of the first step circuit, (2) L is not generated.
- Dec: When evaluating the first program, the receiver should store locally $\mathsf{digestLabels}^* = \mathsf{digestLabels}^{t+1}$. Then, when evaluating the second program, the sender should use $\mathsf{digestLabels}^*$ as the digest labels for the first step circuit.

Multiple Senders with a Single Receiver. The above protocol also works for multiple parallel senders. That is, after the receiver publishes the first message m_1, every sender S can send a message m_S to the receiver enabling the execution of $P_S^D(x_S)$, where D is the initial database of the receiver, and (P_S, x_S) is the program of S. Security follows from the security of OT which supports multiple second OT messages with the same first OT message. Moreover, every sender can execute a sequence of programs on a persistent database. In this case, the receiver keeps a different copy of her initial database for every sender.

3.3 Security Against Malicious Adversaries

The aforementioned results are obtained in the semi-honest setting. We can upgrade to security against a malicious sender by use of (i) non-interactive zero knowledge proofs (NIZKs) [FLS90] at the cost of additionally assuming doubly enhanced trapdoor permutations or bilinear maps [CHK04; GOS06], (ii) the techniques of Ishai et al. [IKO+11] while obtaining slightly weaker security,[4] or (iii) interactive zero-knowledge proofs but at the cost of additional interaction.

Upgrading to security against a malicious receiver is tricky. This is because the receiver's public encoding is short and hence, it is not possible to recover the receiver's entire database just given the encoding. Standard simulation-based security can be obtained by using (i) universal arguments as done by [CV12; COV15] at the cost of additional interaction, or (ii) using SNARKs at the cost of making extractability assumptions [BCCT12; BCG+13].[5]

[4]The receiver is required to keep the output of the computation private.

[5]We finally note that relaxing to the weaker notion of indistinguishability-based security we can expect to obtain the best of both worlds, i.e. a non-interactive solution while making only a black-box use of the adversary (a.k.a. avoiding the use of extractability assumptions). We leave this open for future work.

Chapter 4

Homomorphic Encryption for RAM

In this chapter, we present another application of laconic OT, that is multi-hop homomorphic encryption for RAM programs. In Section 4.1 we give a technical overview of this application and then we formalize our model in Section 4.2. We present our construction with unprotected memory access (UMA) security in Section 4.3 and enhance it to obtain full security in Section 4.4.

4.1 Technical Overview

Our Model. We consider a scenario where a server S, holding an input x, publishes a public key pk and an encryption ct of x under pk. Now this ciphertext is passed on to a client Q that will compute a (possibly private) program P accessing memory D on the value encrypted in ct, obtaining another ciphertext ct'. Finally, we want that the server can use its secret key to recover $P^D(x)$ from the ciphertext ct' and $\widetilde{D}$, where $\widetilde{D}$ is an encrypted form of D that has been previously provided to S in a one-time setup phase. More generally, the computation could be performed by multiple clients $Q_1, \ldots, Q_n$. In this case, each client is required to place a pre-processed version of its database $\widetilde{D}_i$ with the server during setup. The computation itself could be performed in different sequences of the clients (for different extensions of the model, see Section 4.2). Examples of two such computation paths are shown in Figure 1.1.

For security, we want IND-CPA security for server's input x. For honest clients, we want *program-privacy* as well as *data-privacy*, i.e., the evaluation does not leak anything beyond the output of the computation even when the adversary corrupts the server and any subset of the clients. We note that data-privacy is rather easy to achieve via encryption and ORAM. Hence we focus on the challenges of achieving UMA security for honest clients, i.e., the adversary is allowed to learn the database D as well as memory access pattern of P on D.

UMA Secure Multi-Hop Scheme. We first build on the ideas from non-interactive secure computation for RAM programs. Every client first passes its database to the server.

Then in every round, the server sends an OT message for input x. We assume for simplicity that every client has an up-to-date digest of its own database. Next, the first client Q_1 generates a garbled program for P_1, say ct_1 and sends it to Q_2. Here, the garbled program consists of t_1 (t_1 is the running time of P_1) garbled circuits accessing D_1 via laconic OT as described in the previous application. Now, Q_2 appends its garbled program for P_2 to the end of ct_1 and generates ct_2 consisting of ct_1 and new garbled program. Note that P_2 takes the output of P_1 as input and hence, the output keys of the last garbled circuit of P_1 have to be compatible with the input keys of the first garbled circuit of P_2 and so on. If we continue this procedure, after the last client Q_n, we get a sequence of garbled circuits where the first t_1 circuits access D_1, the next set accesses from D_2 and so on. Finally, the server S can evaluate the sequence of garbled circuits given $D_1, \ldots, D_n$. It is easy to see that correctness holds. But we have no security for clients.

The issue is similar to the issue pointed out by [GHV10] for the case of multi-hop garbled circuits. If the client Q_{i-1} colludes with the server, then they can learn both input labels for the garbled program of Q_i. To resolve this issue it is crucial that Q_i re-randomizes the garbled circuits provided by Q_{i-1}. For this we rely on re-randomizable garbled circuits provided by [GHV10], where given a garbled circuit anyone can re-garble it such that functionality of the original circuit is preserved while the re-randomized garbled circuit is unrecognizable even to the party who generated it. In our protocol we use re-randomizable garbled circuits but we stumble upon the following issue.

Recall that in the RAM application above, a garbled circuit outputs the laconic OT ciphertexts corresponding to the input keys of the next circuit. Hence, the input keys of the $(\tau+1)$-th circuit have to be hardwired inside the τ-th circuit. Since all of these circuits will be re-randomized for security, for correctness we require that we transform the hardwired keys in a manner consistent with the future re-randomization. But for security, Q_{i-1} does not know the randomness that will be used by Q_i.

Our first idea to resolve this issue is as follows: The circuits generated by Q_{i-1} will take additional inputs $s_i, \ldots s_n$ which are the randomness used by future parties for their re-randomization procedure. Since we are in the non-interactive setting, we cannot run an OT protocol between clients Q_{i-1} and later clients. We resolve this issue by putting the first message of OT for s_j in the public key of client Q_j and client Q_{i-1} will send the OT second messages along with ct_{i-1}. We do not want the clients' public keys to grow with the running time of the programs, hence, we think of s_j as PRF keys and each circuit re-randomization will invoke the PRF on a unique input.

The above approach causes a subtle issue in the security proof. Suppose, for simplicity, that client Q_i is the only honest client. When arguing security, we want to simulate all the garbled circuits in ct_i. To rely on the security of re-randomization, we need to replace the output of the PRF with key s_i with uniform random values but this key is fed as input to the circuits of the previous clients. We note that this is not a circularity issue but makes arguing security hard. We solve this issue as follows: Instead of feeding in PRF keys directly to the garbled circuits, we feed in corresponding outputs of the PRF. We generate the PRF output via a bunch of PRF circuits that take the PRF keys as input (see Figure 4.1). Now

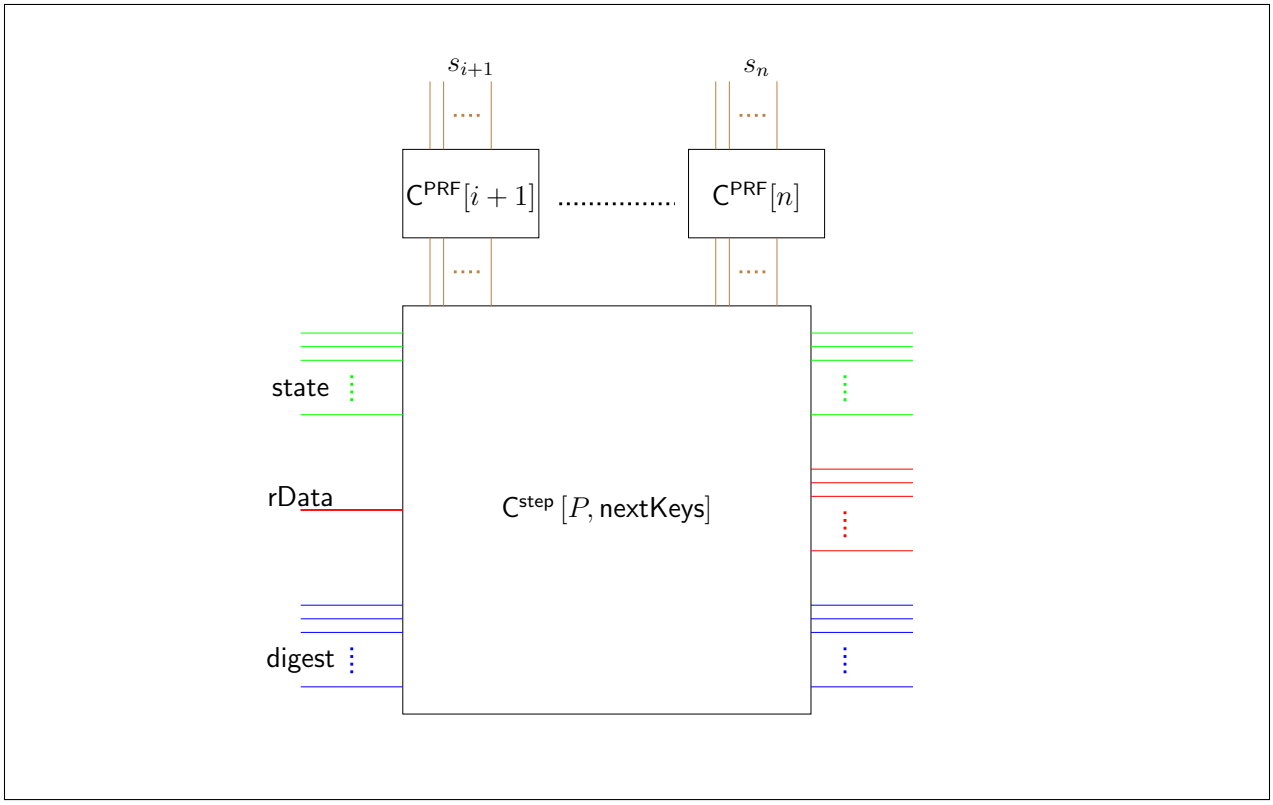


Figure 4.1: One step circuit for P_i along with the attached PRF circuits generated by Q_i.

during simulation, we first simulate these PRF circuits, followed by the simulation of the main circuits.

4.2 Our Model

Consider a server S and a collection of clients $Q_1, Q_2, \ldots$ with private databases $D_1, D_2, \ldots$, respectively. The clients ship their encrypted databases to S to be computed on later in multiple executions in a persistent manner. At the beginning of any execution, the server S encrypts his private input x as ct_0, chooses a subset of clients $Q_{i_1}, \ldots, Q_{i_n}$ and sends the ct_0 to client Q_{i_1}. Next, for all $j \in [n]$, client Q_{i_j} homomorphically evaluates an arbitrary program P_j of his choice on ct_{j-1} to obtain ct_j. Finally, client Q_{i_n} sends ct_n to the server S. The server decrypts this ciphertext using his secret key of encryption as well as encrypted databases sent earlier to learn $P_n^{D_{i_n}}\left(\ldots P_1^{D_{i_1}}(x)\ldots\right)$. During this execution, the databases get updated and future execution of any client happens on respective updated databases.

We require that the size of the ciphertext only grows with the cumulative running time of all programs in an execution and is independent of the size of the databases. For security, we require program and data privacy for all honest clients against an adversary that corrupts

the server and any subset of the clients. Next, we describe the model formally.

Syntax. We say that an ordered sequence of RAM programs $P_1, \cdots, P_n$ are *compatible* if the output length of P_i is the same as the input length of P_{i+1} for every $i \in [n-1]$. A multi-hop RAM homomorphic encryption scheme $\mathsf{mhop\text{-}RAM} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{InpEnc}, \mathsf{EncData}, \mathsf{Eval}, \mathsf{Dec})$ has the following syntax. We define the algorithms with regard to clients $Q_1, \ldots, Q_n$.

- **Setup.** $\mathsf{crs} \leftarrow \mathsf{Setup}(1^\kappa)$.
  On input the security parameter 1^κ, it outputs a common reference string.

- **Key Generation.** $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\kappa)$.
  On input the security parameter 1^κ, it outputs a public/secret key pair $(\mathsf{pk}, \mathsf{sk})$.

- **Database Encryption.** $(\widetilde{D} = (\hat{D}, \mathsf{digest})) \leftarrow \mathsf{EncData}(\mathsf{crs}, D)$.
  On input the common reference string crs and database $D \in \{0,1\}^M$, it outputs an encrypted database $\widetilde{D} = (\hat{D}, \mathsf{digest})$, where digest is a short digest of the database.

- **Input Encryption.** $(\mathsf{ct}, \mathrm{x\_secret}) \leftarrow \mathsf{InpEnc}(x)$.
  On input S's input x, it outputs a ciphertext ct and secret state for S denoted by x_secret.

- **Homomorphic Evaluation.** $\mathsf{ct}' \leftarrow \mathsf{Eval}(\mathsf{crs}, i, \{\mathsf{pk}_j\}_{j=i+1}^n, \mathsf{ct}, \mathsf{sk}, (P,t), \mathsf{digest})$.
  It takes as input the crs, the client number i, the public keys of the clients later in the sequence, i.e., $Q_{i+1}, \cdots, Q_n$, a ciphertext from the previous client, Q's secret key sk, Q's RAM program P with maximum run-time t and the digest digest of the database D of Q. It then outputs a new ciphertext ct'.

- **Decryption.** $y = \mathsf{Dec}^{\widetilde{D}_1, \cdots, \widetilde{D}_n}(\mathsf{crs}, \mathrm{x\_secret}, \mathsf{ct})$.
  On input the crs, server's state x_secret, the final ciphertext ct from client Q_n, and RAM access to encrypted databases $\widetilde{D}_1, \cdots, \widetilde{D}_n$, it outputs y. The procedure Dec is itself modeled as a RAM program that can read and write to arbitrary locations of its database initially containing $\widetilde{D}_1, \cdots, \widetilde{D}_n$.

Next, we describe how these algorithms are used in a real execution.

Real Scenario. In our multi-hop scheme for RAM programs, after the initialization phase that generates the common-random string crs, each cleint runs key generation to generate the public key and the secret key, followed by the database encryption. The encrypted database is sent to the server, and the cleint stores the digest of the database locally. After this initialization phase, the server S can initiate various executions of RAM computations with different subsets of the clients. After each execution, the database of a client gets updated by the server during the decryption phase. It is ensured that the server also learns

the updated digest of the database that is communicated to the clients during the start of the next execution.

At the onset of any execution, the server S encrypts his input and sends the ciphertext ct_0 to the first client Q_1 and maintains x_secret to be used later. The client Q_1 generates the ciphertext ct_1 using his program P_1 and digest digest_1 and sends it to Q_2. Similarly, when a client Q_i receives ct_{i-1} from Q_{i-1}, it uses program P_i and digest_i to generate ct_i and sends it to Q_{i+1}. This continues and finally, Q_n sends ct_n back to the server S. Then, the server runs the decryption procedure on ct_n using all the encrypted databases and secret state x_secret to obtain output y.

For the case of multiple executions, each of the above procedures take the session identifier sid as additional input. We denote by $\widetilde{D}_1^{(\mathsf{sid})}, \cdots, \widetilde{D}_n^{(\mathsf{sid})}$ the encrypted databases before the execution with session identifier sid. Initially $\widetilde{D}_1^{(1)} = \widetilde{D}_1, \cdots, \widetilde{D}_n^{(1)} = \widetilde{D}_n$.

We require the algorithms above to satisfy the correctness, sender privacy, client privacy and efficiency properties described below.

Correctness. We require that in a sequence of executions, each output of homomorphic evaluation equals the output of the corresponding computation in the clear. We formalize this as follows: For every set of keys $\{(\mathsf{pk}_i, \mathsf{sk}_i)\}_{i=1}^n$ in the support of KeyGen, and any collection of initial databases $D_1, \cdots, D_n$, for any unbounded polynomial N number of executions the following holds: For $\mathsf{sid} \in \mathsf{N}$, let $P_1^{(\mathsf{sid})}, \ldots, P_n^{(\mathsf{sid})}$ be the sequence of programs, $x^{(\mathsf{sid})}$ be the server's input and $D_i^{(\mathsf{sid})}$ be the resulting database after executing the session sid-1 in the clear, then

$$\mathsf{Pr}\left[\mathsf{Dec}^{\widetilde{D}_1^{(\mathsf{sid})},\cdots,\widetilde{D}_n^{(\mathsf{sid})}}\left(\mathsf{crs}, \text{x\_secret}^{(\mathsf{sid})}, \mathsf{ct}_n^{(\mathsf{sid})}\right) = {P_n^{(\mathsf{sid})}}^{D_n^{(\mathsf{sid})}}\left(\cdots\left({P_1^{(\mathsf{sid})}}^{D_1^{(\mathsf{sid})}}(x^{(\mathsf{sid})})\right)\cdots\right)\right] = 1,$$

where $\widetilde{D}_i^{(\mathsf{sid})}$ is the resulting garbled database after executing sid-1 homomorphic evaluations, $(\mathsf{ct}_0^{(\mathsf{sid})}, \text{x\_secret}^{(\mathsf{sid})}) \leftarrow \mathsf{InpEnc}(x^{(\mathsf{sid})}), \mathsf{ct}_i^{(\mathsf{sid})} \leftarrow \mathsf{Eval}(\mathsf{crs}, i, \{\mathsf{pk}_j\}_{j=i+1}^n, \mathsf{ct}_{i-1}^{(\mathsf{sid})}, \mathsf{sk}, P_i^{(\mathsf{sid})}, t_i^{(\mathsf{sid})}, \mathsf{digest}_i^{(\mathsf{sid})})$.

Server Privacy (Semantic Security). For server privacy, we require that for every pair of inputs (x_0, x_1), let $(\mathsf{CT}_b, \text{x\_secret}^b) \leftarrow \mathsf{InpEnc}(x_b)$ for $b \in \{0, 1\}$, then

$$\mathsf{CT}_0 \overset{c}{\approx} \mathsf{CT}_1.$$

Client Privacy (Program Privacy) with Unprotected Memory Access (UMA). We define client privacy against a semi-honest adversary that corrupts the server S as well as an arbitrary subset of clients $\mathcal{I} \subset [n]$. Intuitively, we require *program-privacy* for the honest clients such that the adversary cannot learn anything beyond the output of the honest client's program on one input. We formalize this as follows:

There exists a PPT simulator $\mathsf{ihopSim}$ such that the following holds. Let $\mathsf{crs} \leftarrow \mathsf{Setup}(1^\kappa)$, for every set of keys $\{(\mathsf{pk}_i, \mathsf{sk}_i)\}_{i=1}^n$ in the support of KeyGen, and any collection of initial

databases $D_1, \cdots, D_n$, for any unbounded polynomial N number of executions: For $\mathsf{sid} \in \mathsf{N}$, let $P_1^{(\mathsf{sid})}, \ldots, P_n^{(\mathsf{sid})}$ be the sequence of programs, $x^{(\mathsf{sid})}$ be the server's input, then

$$\left(\mathsf{crs}, (\widetilde{D}_1, \ldots, \widetilde{D}_n), \left\{\mathsf{ct}_0^{(\mathsf{sid})}, \mathsf{ct}_1^{(\mathsf{sid})}, \ldots, \mathsf{ct}_n^{(\mathsf{sid})}\right\}_{\mathsf{sid}\in[\mathsf{N}]}\right) \overset{c}{\approx}$$
$$\left(\mathsf{crs}, \mathsf{ihopSim}\left(\mathsf{crs}, \left\{\{\mathsf{pk}_i, \mathsf{sk}_i\}_{i\in[n]}, \left(\{D_j, P_j^{(\mathsf{sid})}\}_{j\in\mathcal{I}}, x^{(\mathsf{sid})}\right), \left\{D_j, \mathsf{MemAccess}_j^{(\mathsf{sid})}, y_j^{(\mathsf{sid})}\right\}_{j\in[n]\setminus\mathcal{I}}\right\}_{\mathsf{sid}\in[\mathsf{N}]}\right)\right)$$

where $\widetilde{D}_i, \mathsf{ct}_i^{(\mathsf{sid})}$ corresponds to outputs in the real execution given all the programs and databases and $y_j^{(\mathsf{sid})} = {P_j^{(\mathsf{sid})}}^{D_j^{(\mathsf{sid})}}\left(\cdots\left({P_1^{(\mathsf{sid})}}^{D_1^{(\mathsf{sid})}}(x^{(\mathsf{sid})})\right)\cdots\right)$.

Remark 4.2.1. *We note that the above definition also captures the security against a semi-malicious adversary who may choose his randomness for* KeyGen *maliciously but behaves honestly in the protocol.*

Client Privacy (Program Privacy) with Full Security. For full client privacy, the simulator does not get the database or access pattern of the honest clients. That is, the simulator $\mathsf{ihopSim}$ takes as input $\left\{\{\mathsf{pk}_i, \mathsf{sk}_i\}_{i\in[n]}, (\{D_j, P_j^{(\mathsf{sid})}\}_{j\in\mathcal{I}}, x^{(\mathsf{sid})}), \left\{1^{M_j}, 1^{t_j^{(\mathsf{sid})}}, y_j^{(\mathsf{sid})}\right\}_{j\in[n]\setminus\mathcal{I}}\right\}_{\mathsf{sid}\in[\mathsf{N}]}$, where M_j is the size of D_j and $t_j^{(\mathsf{sid})}$ is the running time of $P_j^{(\mathsf{sid})}$.

Efficiency. We require the following efficiency guarantees from $\mathsf{mhop\text{-}RAM}$. Let $M_i = |D_i|$.

- $|\widetilde{D}_i| = M_i \cdot \mathsf{poly}(\kappa, \log M_i)$ for all $i \in [n]$.
- $|\mathsf{ct}_0| = |x| \cdot \mathsf{poly}(\kappa)$, where ct_0 is the output of $\mathsf{InpEnc}(x)$.
- $|\mathsf{ct}_i| = \sum_{j=1}^{i} n \cdot t_j \cdot \mathsf{poly}(\kappa, \log M_j, \log t_j)$ for all $i \in [n]$.

Extension. This definition (and our construction) can be extended to the setting where in each execution all the clients do not necessarily join the homomorphic evaluation. We allow for different set of clients to participate in different executions. In particular, before the first execution, the initial database of every client is encrypted. Later before each execution, a sequence of distinct clients $\langle i_1, \cdots, i_m\rangle$ can be specified.

The input encryption is the same as above, while the homomorphic evaluation is executed in the specified order (as specified by the server) as $\mathsf{ct}_j \leftarrow \mathsf{Eval}(\mathsf{crs}, j, \{\mathsf{pk}_{i_u}\}_{u=j+1}^{n}, \mathsf{sk}_{i_j}, \mathsf{ct}_{j-1}, P_{i_j}, t_{i_j}, \mathsf{digest}_{i_j})$ for every $j \in [m]$. And the decryption is executed as $y = \mathsf{Dec}^{\widetilde{D}_{i_1}, \cdots, \widetilde{D}_{i_m}}(\mathsf{crs}, \mathrm{x\_secret}, \mathsf{ct}_m)$, where $\widetilde{D}_{i_1}, \cdots, \widetilde{D}_{i_m}$ are the up-to-date garbled databases of clients $Q_{i_1}, \ldots, Q_{i_m}$. The correctness and privacy properties can be naturally extended to this setting.

4.3 UMA-Secure Construction

In this section, we first describe the UMA-secure scheme for a single execution in Section 4.3.2 and then explain how this scheme can be extended naturally for multiple executions in Section 4.3.5. Also, as we shall see our scheme can easily be extended to the setting where different subset of parties participate in each session. The correctness and security proofs are presented in Sections 4.3.3 and 4.3.4, respectively. Necessary background for our construction is given in Section 4.3.1.

4.3.1 Background

In this section we introduce building blocks needed in our construction. The two-message secure function evaluation (SFE) and re-randomizable secure function evaluation that we consider are both based on garbled circuits (see Section 2.4.1.1 for the definition of garbled circuits). In addition to these building blocks, we will also need RAM computation model (see Section 3.2.2.1) and two-message oblivious transfer (see Section 3.2.2.2). We use $[n]$ to denote the set $\{1, \ldots, n\}$.

4.3.1.1 Two-Message Secure Function Evaluation

A two-message secure function evaluation (SFE) based on garbled circuits is as follows: Let $\mathcal{U}(\cdot, \cdot)$ be a particular "universal circuit evaluator" that takes as input the description of a circuit C and an argument x, and returns $\mathcal{U}(C, x)$. We write $C(x)$ as a shorthand for $\mathcal{U}(C, x)$. Let Alice be the client with private input x and Bob have private input a circuit C. The protocol is as follows:

1. $(m_1, \text{x\_secret}) \leftarrow \mathsf{SFE}_1(x)$: Alice computes $(m_1, \text{x\_secret}) \leftarrow \mathsf{OT}_1(x)$ and sends m_1 to Bob.
2. $m_2 \leftarrow \mathsf{SFE}_2(C, m_1)$: Bob computes $\widetilde{\mathsf{C}} \leftarrow \mathsf{GCircuit}\left(\mathsf{C}, \{\mathsf{lab}_b^w\}_{w \in \mathsf{inp}(\mathsf{C}), b \in \{0,1\}}\right)$ and $\mathsf{L} \leftarrow \mathsf{OT}_2\left(m_1, \{\mathsf{lab}_b^w\}_{w \in \mathsf{inp}(\mathsf{C}), b \in \{0,1\}}\right)$. Sends $m_2 := (\widetilde{\mathsf{C}}, \mathsf{L})$.
3. $y = \mathsf{SFE}_{\mathsf{out}}(\text{x\_secret}, m_2)$: Alice locally computes the output: $\{\mathsf{lab}_{x_w}^w\}_{w \in \mathsf{inp}(\mathsf{C})} = \mathsf{OT}_3(\mathsf{L}, \text{s\_secret})$, and $y = \mathsf{GEval}\left(\widetilde{\mathsf{C}}, \{\mathsf{lab}_{x_w}^w\}_{w \in \mathsf{inp}(\mathsf{C})}\right)$.

The correctness of the above protocol follows from the correctness of Yao garbled circuits. It can be shown that the above protocol is a secure function evaluation protocol satisfying both semi-honest client privacy and semi-honest server privacy.

4.3.1.2 Re-Randomizable Secure Function Evaluation

[GHV10] defined the tool of "re-randomizable secure function evaluation" that was used to realize multi-hop homomorphic computation for circuits. This tool was constructed under the DDH assumption by instantiating Yao's garbled circuits with a special encryption scheme (BHHO [BHHO08]) and using re-randomizable two-message oblivious transfer [NP01].

Definition 4.3.1. *A secure function evaluation protocol is said to be re-randomizable if there exists an efficient procedure* $\mathsf{Re\text{-}rand}$ *such that for every input* x *and function* f *and every* $(m_1, x\_secret) \in \mathsf{SFE}_1(x)$ *and* $m_2 \in \mathsf{SFE}_2(C, m_1)$*, the distributions* $\{x, C, m_1, x\_secret, m_2, \mathsf{Re\text{-}rand}(m_1, m_2)\}$ *and* $\{x, C, m_1, x\_secret, m_2, \mathsf{SFE}_2(C, m_1)\}$ *are computationally indistinguishable.*

[GHV10] proved the following:

Theorem 4.3.2 ([GHV10])**.** *Under the DDH assumption, there exists a re-randomizable secure function evaluation protocol satisfying Definition 4.3.1.*

Below, we abstract out the scheme of [GHV10] by stating some of the procedures implicitly provided by [GHV10] that will be needed for this paper.

Definition 4.3.3 (Re-randomizable Yao garbled circuits.)**.** *The scheme in [GHV10] provides the following algorithms (implicitly) for their re-randomizable Yao scheme.*

1. $\mathsf{Keys} = \mathsf{SampleKeys}(1^\kappa, \mathsf{W}; r)$*: Takes as input a set of input wires* W *as well randomness* r *and outputs the input-keys for set of wires* W *for re-randomizable Yao. Note that it is a deterministic function given the randomness* r*. When clear from context, we will skip mentioning the inputs in the calls to this function.*

2. $\widetilde{\mathsf{C}} \leftarrow \mathsf{ReGCircuit}(C, \mathsf{InpKeys})$*: Takes as input a circuit* C *and* $\mathsf{InpKeys}$ *for the input wires of* C *and outputs a re-randomizable garbled circuit* $\widetilde{\mathsf{C}}$ *where input wires have keys as* $\mathsf{InpKeys}$*.*

3. $\widetilde{\mathsf{C}}' \leftarrow \mathsf{ReGCircuit}'(C, \mathsf{InpKeys}, \mathsf{OutKeys})$*: Takes as input a circuit* C*,* $\mathsf{InpKeys}$ *for input wires of* C *and* $\mathsf{OutKeys}$ *for output wires of* C*, and outputs a re-randomizable garbled circuit* $\widetilde{\mathsf{C}}$ *where input wires have keys as* $\mathsf{InpKeys}$ *and output wires have keys as* $\mathsf{OutKeys}$*.*

4. $\mathsf{Keys}^\dagger = \mathsf{Transform}(\mathsf{Keys}, r)$*: Takes as input* Keys *and randomness* r *and outputs randomized keys* $\mathsf{Keys}^\dagger$*. Also, we use* $\mathsf{Transform}(\mathsf{Keys}, \{r_1, \ldots, r_k\})$ *to denote*

$$\mathsf{Transform}(\mathsf{Transform}(\ldots(\mathsf{Transform}(\mathsf{Keys}, r_1), \ldots), r_{k-1}), r_k)$$

5. $(\widetilde{\mathsf{C}}', \mathsf{L}') \leftarrow \mathsf{Re\text{-}rand}\left((\widetilde{\mathsf{C}}, \mathsf{L}), \{r_w\}_{w \in \mathsf{Wires}(C)}\right)$*: Takes as input a re-randomizable garbled circuit* $\widetilde{\mathsf{C}}$ *and OT second messages* L *for the keys of input wires of* C *and randomness to re-randomize each wire of* C *and outputs a new functionally equivalent re-randomizable garbled circuit* $\widetilde{\mathsf{C}}'$ *and consistent OT second messages* L'*. This procedure satisfies the property of re-randomizable SFE. Moreover, the guarantee is that after randomization, for any wire* w*, the new keys for* w *in* $\widetilde{\mathsf{C}}'$ *are* $\mathsf{Transform}(\mathsf{lab}^w, r_w)$*. Finally, re-randomization of OT messages only requires*[1] $\{r_w\}_{w \in \mathsf{inp}(\mathsf{C})}$*.*

[1] In fact, each OT message for keys of a wire can be randomized consistently just given the randomness used for that wire.

For our multi-hop homomorphic scheme for RAM it will be useful to define $\mathsf{SampleKeys}(\cdot)$ as follows: $\mathsf{SampleKeys}(1^\kappa, \mathsf{W}, r) = \mathsf{Transform}(\mathsf{SampleKeys}(1^\kappa, \mathsf{W}, 0^\star), r)$.

4.3.2 Construction For a Single Execution Involving All Parties

Let $Q_1, \ldots, Q_n$ be the clients holding databases $D_1, \ldots, D_n$, respectively, and S be the server. Let the server's private input be x and secret programs of clients be $P_1, \ldots, P_n$, respectively. Let $\ell OT = (\mathsf{crsGen}, \mathsf{Commit}, \mathsf{Send}, \mathsf{Receive}, \mathsf{SendWrite}, \mathsf{ReceiveWrite})$ be an updatable laconic OT scheme with sender privacy as defined in Definition 2.2.2. Let $\mathsf{Re\text{-}GC} = (\mathsf{SampleKeys}, \mathsf{ReGCircuit}, \mathsf{ReGCircuit}', \mathsf{Transform}, \mathsf{Re\text{-}rand})$ be a re-randomizable scheme for Yao's garbled circuits given by [GHV10] (see Definition 4.3.3). Let $\mathsf{OT} = (\mathsf{OT_1}, \mathsf{OT_2}, \mathsf{OT_3})$ be a two-message oblivious transfer protocol as defined in Section 3.2.2.2.

The multi-hop RAM scheme $\mathsf{mhop\text{-}RAM} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{EncData}, \mathsf{InpEnc}, \mathsf{Eval}, \mathsf{Dec})$ is as follows: The algorithms $\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{EncData}$ are formally described in Figure 4.2.

Setup. $\mathsf{crs} \leftarrow \mathsf{Setup}(1^\kappa)$
Setup algorithm generates the common reference string for laconic OT.

Key Generation. $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\kappa)$
Each client runs this algorithm once to generate the secret-key sk and public-key pk. A client Q picks a PRF seed s as the secret key. Next, it generates the public key as the first message of OT for s and secret-key as the secret state for OT as well as PRF key.

Looking ahead, the client Q will use the PRF key s to garble his own P and to randomize the garbled program generated by all previous clients in any execution.

Database Encryption. $\widetilde{D} \leftarrow \mathsf{EncData}(\mathsf{crs}, D)$
Each client runs this algorithm at the beginning to garble the database and sends the garbled database to the server S. The garbled database is generated by executing the Hash procedure of laconic OT. This outputs an encoded database $\hat{\mathsf{D}}$ and a digest digest, both of which are given to the server S.

Input Encryption. $(\mathsf{ct}, \mathrm{x\_secret}) \leftarrow \mathsf{InpEnc}(x)$
In each execution, the server S encrypts its input x as follows: It computes the first message of OT as the ciphertext and stores the secret state of OT to be used for decryption of computation later. The ciphertext is sent the first client, w.l.o.g. Q_1. The algorithm InpEnc is described formally in Figure 4.3.

Homomorphic Evaluation. $\mathsf{ct}' \leftarrow \mathsf{Eval}\left(\mathsf{crs}, i, \{\mathsf{pk}_j\}_{j=i+1}^n, \mathsf{ct}, \mathsf{sk} = (s, \mathrm{s\_secret}), (P, t), \mathsf{digest}\right)$.
This algorithm is executed by client Q_i to generate the next ciphertext ct' given ct from client Q_{i-1}, and is described formally in Figure 4.4. This is the most involved procedure in our construction, and hence, we first provide an informal description. At a very high level,

Setup. $\mathsf{crs} \leftarrow \mathsf{Setup}(1^\kappa)$.

1. Sample $\mathsf{crs} \leftarrow \mathsf{crsGen}(1^\kappa)$.
2. Output crs.

Key Generation. $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\kappa)$.

1. Sample a PRF key $s \leftarrow \{0,1\}^\kappa$ and generate $(\mathsf{pk}, \mathrm{s\_secret}) \leftarrow \mathsf{OT}_1(s)$.
2. Output $(\mathsf{pk}, \mathsf{sk} := (s, \mathrm{s\_secret}))$.

Database Encryption. $\widetilde{D} \leftarrow \mathsf{EncData}(\mathsf{crs}, D)$.

1. $(\mathsf{digest}, \hat{\mathsf{D}}) \leftarrow \mathsf{Hash}(\mathsf{crs}, D)$.
2. Output $\widetilde{D} = (\mathsf{digest}, \hat{\mathsf{D}})$.

Figure 4.2: Description of the setup, key generation and database encryption algorithms.

Input Encryption. $(\mathsf{ct}, \mathrm{x\_secret}) \leftarrow \mathsf{InpEnc}(\mathsf{crs}, x)$.

1. $(\mathsf{ct}, \mathrm{x\_secret}) \leftarrow \mathsf{OT}_1(x)$.

Figure 4.3: Description of the input encryption algorithm.

as illustrated in Figure 4.5, the client Q_i generate the garbled program for P consisting of t garbled circuits and also re-randomize all the circuits in ct. As mentioned before, this re-randomization step is crucial to get program privacy for this client. Moreover, the re-randomization has to be done carefully so that the previous ct is consistent with the new garbled program.[2]

This procedure consists of four main steps: Let T be the number of step-circuits in ct.

Step 1 Garble the new program P: For each $\tau \in [T+1, T+t]$, client does the following: It generates a "super-circuit" that is illustrated in Figure 4.6 consisting of a CPU step circuit $\mathsf{C}^{\mathsf{step}}_\tau$ (see Figure 4.7) and PRF circuits $\mathsf{C}^{\mathsf{PRF}}_{\tau,i+1}, \ldots, \mathsf{C}^{\mathsf{PRF}}_{\tau,n}$ (see Figure 4.8). A step circuit, encodes the logic of a CPU step of a program P and PRF circuits provide a part of the randomness used in re-randomization. We will elaborate on the functionality of PRF circuits later.

The garbled program will consists of garbled circuits corresponding to all the step circuits and PRF circuits. The first step is to pick the keys for the input wires of all of these circuits. Next, we begin by describing the step circuits.

Step Circuits $\mathsf{C}^{\mathsf{step}}_\tau$ (Figure 4.7): The inputs of a step circuit (see Figure 4.6) can be partitioned into $((\mathsf{state}, \mathsf{rData}, \mathsf{digest}), \mathsf{Rd})$, where state is the current CPU state, rData is the bit-read from database, and digest is the up-to-date digest of the database. Rd corresponds to the randomness given as input to the step-circuit computed from the

[2]We do this by keeping track of the randomness used in randomizing the input wires for each garbled circuit.

Homomorphic Evaluation.

$\mathsf{ct}' \leftarrow \mathsf{Eval}\Big(\mathsf{crs}, i, \{\mathsf{pk}_j\}_{j=i+1}^{n}, \mathsf{ct} = \Big(\mathsf{L}_0, \Big\{\widetilde{\mathsf{C}}_{\tau}^{\mathsf{step}}, \{\widetilde{\mathsf{C}}_{\tau,j}^{\mathsf{PRF}}, \mathsf{L}_{\tau,j}\}_{j=i}^{n}\Big\}_{\tau\in[T]}\Big), \mathsf{sk} = (s, \mathrm{s\_secret}), (P,t), \mathsf{digest}\Big)$.

1. **Generate the "new" garbled program for** P: Generate garbled circuits $\Big\{\widetilde{\mathsf{C}}_{\tau}^{\mathsf{step}}, \{\widetilde{\mathsf{C}}_{\tau,j}^{\mathsf{PRF}}\}_{j=i+1}^{n}\Big\}_{\tau=T+1}^{T+t}$.
   a) Set $\mathsf{stateKeys}^{\tau}, \mathsf{dataKeys}^{\tau}, \mathsf{digestKeys}^{\tau}, \mathsf{RdKeys}^{\tau,j}, \mathsf{PKeys}^{\tau,j}$ for each $\tau \in [T+1, T+t]$ and $j \in [i+1, n]$ as $\mathsf{SampleKeys}(F_s(\mathtt{GC\_}\star||\tau))$ where $F_s(\mathtt{GC\_}\star||\tau)$ is the randomness used and $\star \in \{\mathtt{STATE}, \mathtt{DATA}, \mathtt{DIGEST}, \mathtt{RD}, \mathtt{P}\}$, respectively.
   Set $\mathsf{stateKeys}^{\tau}, \mathsf{dataKeys}^{\tau}, \mathsf{digestKeys}^{\tau}$ to $\mathsf{SampleKeys}(0^*)$ for $\tau = T+t+1$.
   b) *Garble* $\mathsf{C}^{\mathsf{step}}$ *circuits:* For each $\tau \in [T+1, T+t]$
   $$\widetilde{\mathsf{C}}_{\tau}^{\mathsf{step}} \leftarrow \mathsf{ReGCircuit}\left(\mathsf{C}^{\mathsf{step}}[i, \mathsf{crs}, P, \mathsf{Keys}^{\tau+1}, F_s(\mathtt{PSI}||\tau)], \left(\mathsf{Keys}^{\tau}, \{\mathsf{RdKeys}^{\tau,j}\}_{j=i+1}^{n}\right)\right),$$
   where $\mathsf{Keys}^{\tau} = (\mathsf{stateKeys}^{\tau}, \mathsf{dataKeys}^{\tau}, \mathsf{digestKeys}^{\tau})$.
   Embed labels $\mathsf{dataKeys}_0^{T+1}$ and $\mathsf{digestKeys}_{\mathsf{digest}}^{T+1}$ in $\widetilde{\mathsf{C}}_{T+1}^{\mathsf{step}}$.
   c) *Garble* $\mathsf{C}^{\mathsf{PRF}}$ *circuits:* For each $[T+1, T+t]$ and $j \in [i+1, n]$, compute
   $$\widetilde{\mathsf{C}}_{\tau,j}^{\mathsf{PRF}} \leftarrow \mathsf{ReGCircuit}'\left(\mathsf{C}^{\mathsf{PRF}}[\tau], \mathsf{PKeys}^{\tau,j}, \mathsf{RdKeys}^{\tau,j}\right).$$
2. **Generate the OT second messages for newly generated circuits**: For all $\tau \in [T+1, T+t]$ and $j \in [i+1, n]$ compute $\mathsf{L}_{\tau,j} \leftarrow \mathsf{OT}_2\left(\mathsf{pk}_j, \mathsf{PKeys}^{\tau,j}\right)$.
3. **Obtain partial labels for previous circuits**:
   a) For every $\tau \in [T]$, compute $\mathsf{M}_{\tau,i} = \mathsf{OT}_3\left(\mathsf{L}_{\tau,i}, \mathrm{s\_secret}\right)$ and $\widetilde{\mathsf{C}}_{\tau,i}^{\mathsf{PRF}}$ using input labels $\mathsf{M}_{\tau,i}$ and embed the labels in $\widetilde{\mathsf{C}}_{\tau}^{\mathsf{step}}$.
   b) If $i = 1$, then $\mathsf{L}_0 \leftarrow \mathsf{OT}_2(\mathsf{L}_0, \mathsf{stateKeys}^1)$.
4. **Re-randomize previous garbled circuits:** If $i > 1$, do the following:
   a) For each $\tau \in [T]$, re-randomize the circuit $\widetilde{\mathsf{C}}_{\tau}^{\mathsf{step}}$ using $\mathsf{Re\text{-}rand}(\cdot)$ (see Definition 4.3.3) such that the input wire keys are randomized using $F_s(\mathtt{GC\_}\star||\tau)$, where $\star \in \{\mathtt{STATE}, \mathtt{DATA}, \mathtt{DIGEST}, \mathtt{RD}\}$ for different input wires appropriately.
   b) For each $\tau \in [T]$, re-randomize the circuits $\{\mathsf{C}_{\tau,j}^{\mathsf{PRF}}\}_{j\in[i+1,n]}$ and $\{\mathsf{L}_{\tau,j}\}_{j\in[i+1,n]}$ using $\mathsf{Re\text{-}rand}(\cdot)$ such that the input wires are randomized using $F_s(\mathtt{GC\_P}||\tau)$ and output wires are randomized using $F_s(\mathtt{GC\_RD}||\tau)$.
   c) Re-randomize L_0 using $F_s(\mathtt{GC\_STATE}||1)$.
5. Output $\mathsf{ct}' = \left(\mathsf{L}_0, \left\{\widetilde{\mathsf{C}}_{\tau}^{\mathsf{step}}, \{\widetilde{\mathsf{C}}_{\tau,j}^{\mathsf{PRF}}, \mathsf{L}_{\tau,j}\}_{j=i+1}^{n}\right\}_{\tau=1}^{T+t}\right)$.

Figure 4.4: Description of the homomorphic evaluation algorithm.

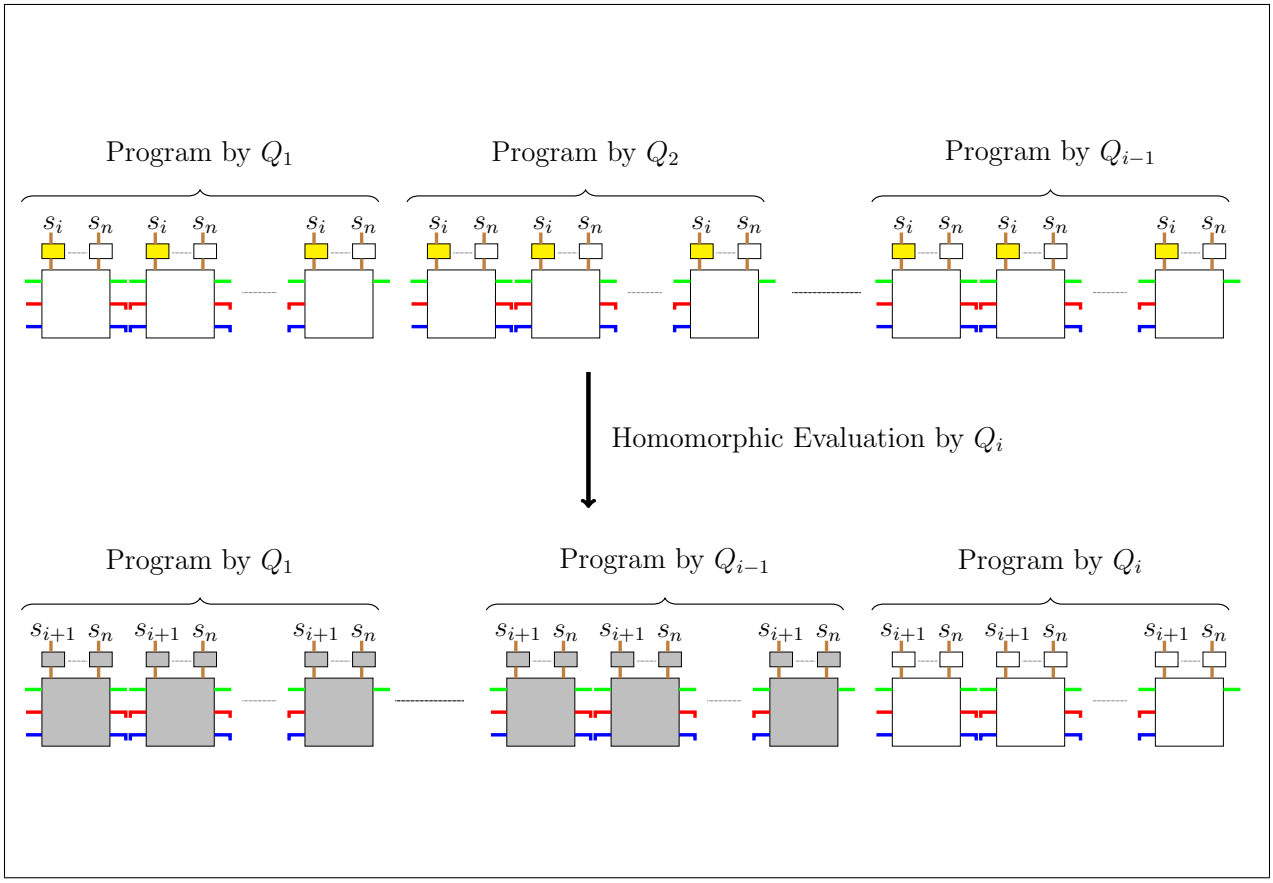


Figure 4.5: Homomorphic Evaluation by Q_i: Q_i contributes new circuits (denoted in white in the lower layer) and processes the input circuits as follows: (i) computes the yellow circuits, and (ii) re-randomizes all input circuits. The re-randomized circuits are shown in gray color.

PRF circuits. A step circuit executes one CPU step and passes on the updated state, new bit read, and new digest to the next step circuit. Note that we do not achieve this by passing the output wires of τ into input wires of $\tau + 1$. That is, the output wire of τ will not have same keys as input wires of $\tau + 1$ (Note that the two consecutive step circuits are not connected by solid lines in Figure 4.5.). Hence, the step circuit τ will have the keys of the next circuit hard-coded inside it.

Next, we explain the logic of a step-circuit. First, it computes the new $(\mathsf{state}', \mathsf{R/W}, L, \mathsf{wData})$. Next, it computes the transformed keys $\mathsf{nextKeys}^\dagger$ of the next step-circuit using the hard-coded keys and the input randomness (this uses the transform functionality of re-randomizable Yao from Section 4.3.1.2). Then, in the case of a "read" it outputs $\mathsf{stateKeys}^\dagger$ corresponding to new state', labels for data via laconic OT procedure $\mathsf{Send}(\cdot)$ for location L where the sender's inputs are $\mathsf{dataKeys}_0^\dagger, \mathsf{dataKeys}_1^\dagger$ and $\mathsf{digestKeys}^\dagger$ corresponding to digest. The case of a write is similar, but now the labels of new updated digest are transferred via laconic OT procedure $\mathsf{SendWrite}(\cdot)$. Note that it follows via

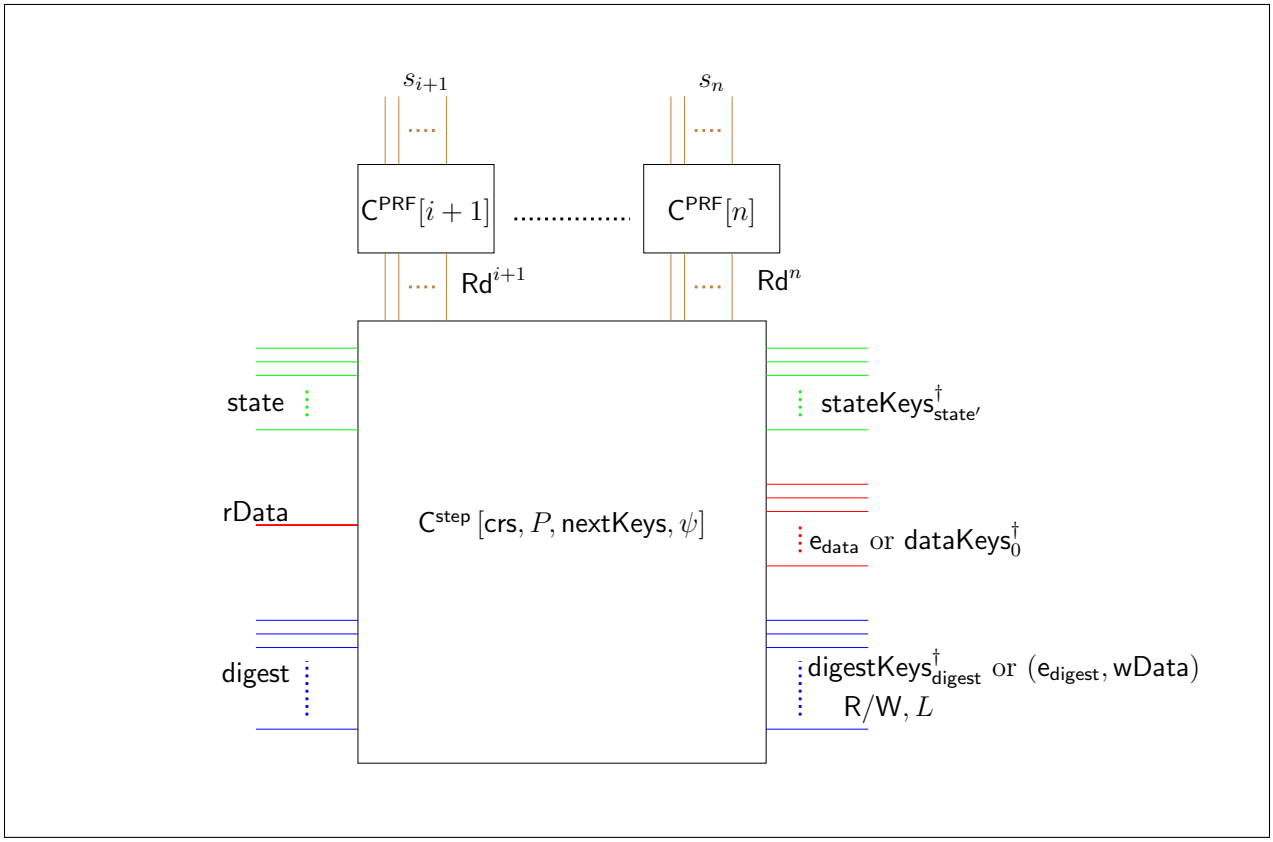


Figure 4.6: One step circuit along with the attached PRF circuits.

correctness of reads and writes of the laconic OT that the evaluator would be able to recover the correct labels for the read-data and the new digest.

The down-bend in output and input wires of step-circuits for data and digest in Figure 4.5 represents that these keys are not output in the clear, but are output using laconic OT. Correct labels will be learnt during execution using the encoded database $\widetilde{D}_i$ and laconic OT procedures.

PRF circuits $\mathsf{C}^{\mathsf{PRF}}_{\tau,j}$ (Figure 4.8): This circuit takes as input a PRF key s_j of client Q_j and outputs the PRF value corresponding to time-step τ. The use of these circuits will be clear when we describe the re-randomization step below.

All these circuits are garbled such that the keys for output wires of PRF circuits are same as keys for Rd input keys of step circuits. In Figure 4.5, this is depicted by joining the output wires of PRF circuits with Rd input wires of step circuit with a solid line. The garbled program consists of garbled step circuits and garbled PRF circuits $\{\widetilde{\mathsf{C}}^{\mathsf{step}}_{\tau}, \{\widetilde{\mathsf{C}}^{\mathsf{PRF}}_{\tau,j}\}_{j\in\{i+1,\ldots,n\}}\}_{\tau\in[T+1,T+t]}$. The client also embeds labels for $\mathsf{rData} = 0$ and digest_i in the first step circuit.

Step 2 Generate OT messages for $\mathsf{C}^{\mathsf{PRF}}_{\tau,j}$: Recall that this circuit takes as input a PRF key s_j of

Hard-coded parameters: $[i, \mathsf{crs}, P, \mathsf{nextKeys} = (\mathsf{stateKeys}, \mathsf{dataKeys}, \mathsf{digestKeys}), \psi]$.
Input: $((\mathsf{state}, \mathsf{rData}, \mathsf{digest}), (\{\omega_j, \phi_j\}_{j>i}))$.

$(\mathsf{state}', \mathsf{R/W}, L, \mathsf{wData}) := C^P_{\mathsf{CPU}}(\mathsf{state}, \mathsf{rData})$.
$\mathsf{nextKeys}^\dagger := \mathsf{Transform}(\mathsf{nextKeys}, \{\omega_j\}_{j>i})$.
Parse $\mathsf{nextKeys}^\dagger$ as $(\mathsf{stateKeys}^\dagger, \mathsf{dataKeys}^\dagger, \mathsf{digestKeys}^\dagger)$.

if $\mathsf{R/W} = \mathsf{read}$ **then**
  $\mathsf{e}_{\mathsf{data}} \leftarrow \mathsf{Send}(\mathsf{crs}, \mathsf{digest}, L, \mathsf{dataKeys}^\dagger; \psi \oplus \bigoplus_{j>i} \phi_j)$.
  return $\Big((\mathsf{stateKeys}^\dagger_{\mathsf{state}'}, \quad \mathsf{e}_{\mathsf{data}}, \quad \mathsf{digestKeys}^\dagger_{\mathsf{digest}}), \mathsf{R/W}, L\Big)$.
else
  $\mathsf{e}_{\mathsf{digest}} \leftarrow \mathsf{SendWrite}\Big(\mathsf{crs}, \mathsf{digest}, L, \mathsf{wData}, \mathsf{digestKeys}^\dagger; \psi \oplus \bigoplus_{j>i} \phi_j\Big)$.
  return $\Big((\mathsf{stateKeys}^\dagger_{\mathsf{state}'}, \quad \mathsf{dataKeys}^\dagger_0, \quad \mathsf{e}_{\mathsf{digest}}, \mathsf{wData} \quad), \mathsf{R/W}, L\Big)$.

Figure 4.7: Pseudocode of the step circuit $\mathsf{C}^{\mathsf{step}}[i, \mathsf{crs}, P, \mathsf{nextKeys}, \psi]$.

Hard-coded parameters: $[\tau]$.
Input: s.
Output: $\Big(\{F_s(\mathtt{GC\_}\star||\tau+1)\}_{\star\in\{\mathtt{STATE,DATA,DIGEST}\}}, F_s(\mathtt{LACONIC\_OT}||\tau)\Big)$.

Figure 4.8: Pseudocode of the PRF circuit $\mathsf{C}^{\mathsf{PRF}}[\tau]$.

client Q_j whose OT first message is present in pk_j. Client Q_i generates the OT second message $\mathsf{L}_{\tau,j}$ for the input keys of $\widetilde{\mathsf{C}}^{\mathsf{PRF}}_{\tau,j}$.

Step 3 Evaluating the PRF circuits for itself: Note that the ciphertext ct consists of a sequence of step circuits and PRF circuits for each step circuit corresponding to $j \in \{i, \ldots, n\}$. See Figure 4.5 where the PRF circuits for client Q_i are depicted in yellow. Q_i computes the input labels for $\widetilde{\mathsf{C}}^{\mathsf{PRF}}_{\tau,i}$ using the OT message $\mathsf{L}_{\tau,i}$ and embeds the output labels in to $\widetilde{\mathsf{C}}^{\mathsf{step}}_{\tau}$ for all $\tau \in [T]$. In other words, Q_i consumes the first PRF circuits from each step of previous ciphertext ct.

Step 4 Re-randomize the previous circuits: After consuming the first PRF circuit from each step, Q_i randomizes all the remaining circuits using appropriate randomness. Note that the input keys of $\widetilde{\mathsf{C}}^{\mathsf{step}}_{\tau+1}$ are randomized using the exact randomness that was fed into $\mathsf{C}^{\mathsf{step}}_{\tau}$ via the PRF circuit for Q_i. This makes sure that the hard-coded input keys of step $\tau+1$ are randomized consistently in the same way as how Q_i will randomize the circuit $\widetilde{\mathsf{C}}^{\mathsf{step}}_{\tau+1}$.

Hence, to conclude, the PRF circuits are present to provide the randomness needed to randomize the hard-coded keys inside the step circuits [3].

[3]Note that randomization of garbled circuits preserves the functionality. Since the keys for the next

Homomorphic Decryption. $y = \mathsf{Dec}^{\widetilde{D}_1,\cdots,\widetilde{D}_n}\left(\mathsf{crs}, \mathrm{x\_secret}, \mathsf{ct} = \left(\mathsf{L}_0, \left\{\widetilde{\mathsf{C}}_\tau^{\mathsf{step}}\right\}_{\tau\in[T]}\right)\right)$.

The algorithm is described formally in Figure 4.9. It takes as input the secret state of the server x_secret and the final ciphertext ct_n consisting of OT message for x and sequence of T step-circuits, where $T = \sum_{i\in[n]} t_i$. Note that all the PRF circuits have been evaluated already by correct parties and correct labels for RdKeys have been embedded into the step-circuits. The server does the following for decryption:

1. It obtains the stateLabels for the first circuit by running OT_3. Note that the first circuit of program of any client has labels for data and digest already embedded. Hence, now the server knows all the labels for the first circuit.
2. For $\tau \in [T]$, the server executes the circuit $\widetilde{\mathsf{C}}_\tau^{\mathsf{step}}$, and learns the labels for the next circuit via running the receiver algorithms of laconic OT correctly.

4.3.3 Correctness

Here we prove correctness (as defined in Section 4.2) for a single execution. In fact, we prove something stronger that would help us extend the scheme to multiple executions in a straight-forward manner. We prove the following two properties:

Property 1. For the above scheme, $y = P_n^{D_n}\left(\ldots P_1^{D_1}(x)\ldots\right)$, where programs, databases and input x are as defined above.

Property 2. Let $\hat{\mathsf{D}}'_i, \mathsf{digest}'_i$ denote the updated encoded database and digest with the server after the execution. We show that these are equal to $\mathsf{Hash}(\mathsf{crs}, D'_i)$, where D' results after executing $P_i^{D_i}(P_{i-1}^{D_{i-1}}(\ldots P_1^{D_1}(x)\ldots))$.

Below we prove correctness via a sequence of facts and claims.

Fact 4.3.4. *At any point in homomorphic evaluation, the circuit $\widetilde{\mathsf{C}}_{\tau,j}^{\mathsf{PRF}}$ and the second OT message for its input keys $\mathsf{L}_{\tau,j}$ are consistent.*

This follows from correctness of $\mathsf{OT}_2(\cdot,\cdot)$ procedure when it is generated and the fact that re-randomization happens consistently in Re-rand procedure of re-randomizable garbled circuits.

Fact 4.3.5. *During the homomorphic evaluation of client Q_i, in Step 3a, Figure 4.4 while obtaining partial labels, $\mathsf{M}_{\tau,i} = \mathsf{PKeys}_s^{\tau,i}$, where s is the PRF key of Q_i.*

This follows from the correctness of OT protocol as well as Fact 4.3.4.

circuit are transferred using the laconic OT, we need to feed in correct keys into the Send functions of laconic OT.

Decryption.

$y = \mathsf{Dec}^{\widetilde{D}_1,\cdots,\widetilde{D}_n}\left(\mathsf{crs}, \text{x\_secret}, \mathsf{ct} = \left(\mathsf{L}_0, \left\{\widetilde{\mathsf{C}}_\tau^{\mathsf{step}}\right\}_{\tau\in[T]}\right)\right)$, where $T = \sum_{i\in[n]} t_i$.

1. For all $i \in [n]$, parse $\widetilde{D}_i = \left(\mathsf{digest}_i, \hat{\mathsf{D}}_i\right)$.
2. Compute $\mathsf{M}_0 = \mathsf{OT}_3\left(\mathsf{L}_0, \text{x\_secret}\right)$.
3. Parse $\widetilde{\mathsf{C}}_1^{\mathsf{step}} = (\widetilde{\mathsf{C}}_1^{\mathsf{step}}, \mathsf{dataLabels}, \mathsf{digestLabels})$.
4. Define $\mathsf{Labels}^1 = (\mathsf{stateLabels}^1 = \mathsf{M}_0, \mathsf{dataLabels}^1 = \mathsf{dataLabels}, \mathsf{digestLabels}^1 = \mathsf{digestLabels})$.
5. For $\tau = 1$ to T do the following:

   Define i s.t. $\tau \in \left[\sum_{j\in[i-1]} t_j + 1, \sum_{j\in[i]} t_j\right]$.

   $(X, \mathsf{R/W}, L) := \mathsf{ReEval}\left(\widetilde{\mathsf{C}}_\tau^{\mathsf{step}}, \mathsf{Labels}^\tau\right)$.

   if $\mathsf{R/W} = \mathsf{read}$ **then**

   Parse $X = (\mathsf{stateLabels}^{\tau+1}, \mathsf{e_{data}}, \mathsf{digestLabels}^{\tau+1})$.

   $\mathsf{dataLabels}^{\tau+1} := \mathsf{Receive}^{\hat{\mathsf{D}}_i}(\mathsf{crs}, \mathsf{e_{data}}, L)$.

   else

   Parse $X = (\mathsf{stateLabels}^{\tau+1}, \mathsf{dataLabels}^{\tau+1}, \mathsf{e_{digest}}, \mathsf{wData})$.

   $\mathsf{digestLabels}^{\tau+1} := \mathsf{ReceiveWrite}^{\hat{\mathsf{D}}_i}(\mathsf{crs}, L, \mathsf{wData}, \mathsf{e_{digest}})$.

   if $\tau = \sum_{j\in[i]} t_j$ and $\tau < T$ **then**

   Parse $\widetilde{\mathsf{C}}_{\tau+1}^{\mathsf{step}} = (\widetilde{\mathsf{C}}_{\tau+1}^{\mathsf{step}}, \mathsf{dataLabels}, \mathsf{digestLabels})$

   Set $\mathsf{dataLabels}^{\tau+1} = \mathsf{dataLabels}$ and $\mathsf{digestLabels}^{\tau+1} = \mathsf{digestLabels}$.

   $\mathsf{Labels}^{\tau+1} := (\mathsf{stateLabels}^{\tau+1}, \mathsf{dataLabels}^{\tau+1}, \mathsf{digestLabels}^{\tau+1})$.
6. Decode output y using $\left(\mathsf{stateLabels}^{T+1}, \mathsf{SampleKeys}(0^*)\right)$.

Figure 4.9: Decryption algorithm for multi-hop RAM.

Fact 4.3.6. *During the homomorphic evaluation of client Q_i, in Step 3a, Figure 4.4 the labels embedded in circuit $\widetilde{\mathsf{C}}_\tau^{\mathsf{step}}$ correspond to $\mathsf{RdKeys}_{\omega_i,\phi_i}^{\tau,i}$ where $\phi_i = F_s(\texttt{LACONIC\_OT}||\tau)$ and $\omega_i = \{F_s(\texttt{GC\_}\star\,||\tau+1)\}_{\star\in\{\texttt{STATE,DATA,DIGEST}\}}$.*

This is because the functionality of the $\mathsf{C}^{\mathsf{PRF}}$ is preserved in randomization so far, Fact 4.3.5 and because the output keys of $\widetilde{\mathsf{C}}_{\tau,i}^{\mathsf{PRF}}$ and $\mathsf{RdKeys}^{\tau,i}$ are same when they are generated and are re-randomized using same randomness in Step 4b of Figure 4.4.

Recall that ct_n consists of garbled step-circuits of client Q_1 followed by Q_2 and so on.

We prove the following fact about garbled step circuits belonging to some client Q_i in final ciphertext ct_n.

Claim 4.3.7. *Consider circuits $\widetilde{\mathsf{C}}^{\mathsf{step}}_{\tau}$ and $\widetilde{\mathsf{C}}^{\mathsf{step}}_{\tau+1}$ such that both belong to program P_i for some i. Since all the PRF circuits $\widetilde{\mathsf{C}}^{\mathsf{PRF}}$ have been evaluated, the value $\mathsf{nextKeys}^{\dagger}$ in $\widetilde{\mathsf{C}}^{\mathsf{step}}_{\tau}$ is well defined. Then, $\mathsf{nextKeys}^{\dagger} = \mathsf{Keys}^{\tau+1}$ where $\mathsf{Keys}^{\tau+1}$ corresponds to the input keys for $\widetilde{\mathsf{C}}^{\mathsf{step}}_{\tau+1}$ in ct_n.*

Proof. Initially, Q_i picks $\mathsf{Keys}^{\tau+1}$ as $\mathsf{SampleKeys}(F_s(\mathtt{GC\_}\star||\tau+1))$, where $\star \in \{\mathtt{STATE}, \mathtt{DATA}, \mathtt{DIGEST}\}$ and uses them in garbling of $\widetilde{\mathsf{C}}^{\mathsf{step}}_{\tau+1}$ as well as are hardcoding inside $\mathsf{C}^{\mathsf{step}}_{\tau}$.

Then, $\widetilde{\mathsf{C}}^{\mathsf{step}}_{\tau+1}$ is randomized by clients $Q_{i+1}, \ldots, Q_n$ such that the $\mathsf{stateKeys}, \mathsf{dataKeys}$, $\mathsf{digestKeys}$ are randomized sequentially using $\omega_j = (F_{s_j}(\mathtt{GC\_STATE}||\tau+1), F_{s_j}(\mathtt{GC\_DATA}||\tau+1), F_{s_j}(\mathtt{GC\_DIGEST}||\tau+1))$. This is same as $\mathsf{Transform}(\mathsf{Keys}^{\tau+1}, \{\omega_j\}_{j>i})$ inside $\widetilde{\mathsf{C}}^{\mathsf{step}}_{\tau}$. By Fact 4.3.6, ω_j is the value used for $\mathsf{Transform}$ in $\widetilde{\mathsf{C}}^{\mathsf{step}}_{\tau}$. □

Claim 4.3.8. *The above claim also holds for $\widetilde{\mathsf{C}}^{\mathsf{step}}_{\tau}$ and $\widetilde{\mathsf{C}}^{\mathsf{step}}_{\tau+1}$ when τ is the last circuit of a program for Q_i and $\tau+1$ is the first circuit for Q_{i+1}.*

Proof. When the client Q_i generates $\widetilde{\mathsf{C}}^{\mathsf{step}}_{\tau}$, the keys hard-coded are $\mathsf{SampleKeys}(0^{\star})$. Then, this circuit is re-randomized by Q_{i+1} resulting in keys $\mathsf{SampleKeys}(F_{s_{i+1}}(\mathtt{GC\_}\star||\tau+1))$ which same as the value used by Q_{i+1} to generate the step-circuit $\widetilde{\mathsf{C}}^{\mathsf{step}}_{\tau+1}$. □

Fact 4.3.9. *The first garbled step circuit $\widetilde{\mathsf{C}}^{\mathsf{step}}_{1}$ gets evaluated on $(x, 0, \mathsf{digest}_1)$.*

This follows from correctness of OT and consistency of re-rerandomization of OT and garbled circuits similar to Fact 4.3.4.

Now, we will prove a lemma about the execution of circuits generated by client Q_1. Then, we will prove a claim about the inputs on which circuit of Q_2 is executed. Finally, the correctness of execution programs of all clients would follow in a similar manner.

Lemma 4.3.10. *Consider the program P_1 and the database D_1 of the first client and the input x of the server. Consider the execution $P_1^{D_1}(x)$ execution in the clear as $(\mathsf{state}_\tau, \mathsf{rData}_\tau)$ as the values on which $\mathsf{C}^{\mathsf{step}}$ is executed. Also, let $(\hat{\mathsf{D}}_1^{\tau}, \mathsf{digest}_\tau)$ denote the $\mathsf{Hash}(\mathsf{crs}, D_1^{\tau})$, where D_1^{τ} is the database at beginning of step τ. Then, while decryption, $\widetilde{\mathsf{C}}^{\mathsf{step}}_{\tau}$ is executed on inputs $(\mathsf{state}_\tau, \mathsf{rData}_\tau, \mathsf{digest}_\tau)$. Moreover, the encoded database held by the server before step τ is $\hat{\mathsf{D}}_1^{\tau}$.*

Proof. We will prove this lemma by induction on τ. The base case follows from Fact 4.3.9. Assume that the lemma holds for $\tau = \rho$, then we prove that the lemma holds for $\rho+1$ as follows: So it holds that $\widetilde{\mathsf{C}}^{\mathsf{step}}_{\rho}$ is executed on $(\mathsf{state}_\rho, \mathsf{rData}_\rho, \mathsf{digest}_\rho)$. Moreover, $(\hat{\mathsf{D}}_1^{\rho}, \mathsf{digest}_\rho)$ denote the $\mathsf{Hash}(\mathsf{crs}, D_1^{\rho})$. Note that $\widetilde{\mathsf{C}}^{\mathsf{step}}_{\rho}$ correctly implements its code that includes one CPU step of P_1. Hence, $(\mathsf{state}', \mathsf{R/W}, L, \mathsf{wData}) = C^{P}_{\mathsf{CPU}}(\mathsf{state}_\rho, \mathsf{rData}_\rho)$. Also, by Claim 4.3.7, $\mathsf{nextKeys}^{\dagger}$ in $\widetilde{\mathsf{C}}^{\mathsf{step}}_{\rho}$ are correct input keys for $\widetilde{\mathsf{C}}^{\mathsf{step}}_{\rho+1}$. There are following two cases:

- $\mathsf{R/W} = \mathsf{read}$: In this case, database and the digest are unchanged. New CPU state and digest are output correctly. Moreover, the labels for bit read from the memory will be learnt via $\mathsf{Receive}$ of updatable laconic OT. Correctness of these labels follows from correctness of read of laconic OT.

- $\mathsf{R/W} = \mathsf{write}$: Similar to above, in this case new state and data keys are correctly output. Moreover, the digest keys w.r.t. the new updated digest are output via laconic OT. The correctness of these labels follows from correctness of laconic OT write function $\mathsf{ReceiveWrite}$. Finally, in this function, the encoded database is updated correctly.

□

Lemma 4.3.11. *Let $\widetilde{\mathsf{C}}^{\mathsf{step}}_{t_1}$ be the last circuit of client Q_1 or program P_1. Then, during decryption, $\widetilde{\mathsf{C}}^{\mathsf{step}}_{t_1+1}$ is executed on $(y_1, 0, \mathsf{digest}_2)$, where $y_1 = P_1^{D_1}(x)$ and digest_2 is the digest for D_2.*

Proof. At the time of homomorphic evaluation, in Step 1b, Figure 4.4, labels $\mathsf{dataKeys}_0^{t_1+1}$ and $\mathsf{digestKeys}_{\mathsf{digest}}^{t_1+1}$ are embedded in $\widetilde{\mathsf{C}}^{\mathsf{step}}_{t_1+1}$. Also, by Claim 4.3.8, in the final ciphertext ct, $\mathsf{nextKeys}^{\dagger}$ inside $\widetilde{\mathsf{C}}^{\mathsf{step}}_{t_1}$ are correct keys for $\widetilde{\mathsf{C}}^{\mathsf{step}}_{t_1+1}$. Hence, the lemma holds since $\widetilde{\mathsf{C}}^{\mathsf{step}}_{t_1}$ outputs $\mathsf{stateKeys}^{\dagger}_{\mathsf{state}'}$, where $\mathsf{state}' = y_1$. □

Lemma 4.3.12. *Consider the program P_i and the database D_i of the client Q_i and the input x of the server. Consider the execution $P_i^{D_i}(y_{i-1})$ execution in the clear as $(\mathsf{state}_\tau, \mathsf{rData}_\tau)$ as the values on which $\mathsf{C}^{\mathsf{step}}$ is executed. Also, let $(\hat{\mathsf{D}}_i^\tau, \mathsf{digest}_\tau)$ denote the $\mathsf{Hash}(\mathsf{crs}, D_i^\tau)$, where D_i^τ is the database at beginning of step τ. Then, while decryption, $\widetilde{\mathsf{C}}^{\mathsf{step}}_\tau$ is executed on inputs $(\mathsf{state}_\tau, \mathsf{rData}_\tau, \mathsf{digest}_\tau)$. Moreover, the encoded database held by the server before step τ is $\hat{\mathsf{D}}_i^\tau$.*

Proof. The lemma follows via induction on number of clients where the base case is proved in Lemma 4.3.10. The rest of the proof follows simply via induction similar to Lemma 4.3.10 where at the end of each program and beginning of a new program we prove Lemma 4.3.11. This proves both the properties 1 and 2. □

4.3.4 Security Proof

Server privacy follows receiver privacy of oblivious transfer. For ease of exposition, we prove client UMA privacy for the setting of a single honest client Q_i for a single execution. At the end of this section we will show that the proof can be extended for the case of multiple honest clients and multiple executions as well.

In the following, we prove that there exists a PPT simulator $\mathsf{ihopSim}$ such that, for any set of databases $\{D_j\}_{j\in[n]}$, any sequence of compatible programs $P_1, \cdots, P_n$ running

time $t_1, \cdots, t_n$ and input x, the outputs of the following two experiments are computational indistinguishable:

Real experiment

- $(\mathsf{pk}_j, \mathsf{sk}_j) \leftarrow \mathsf{KeyGen}(1^\kappa)$ for $\forall j \in [n]$.
- $\widetilde{D}_j = (\mathsf{digest}_j, \hat{\mathsf{D}}_j) \leftarrow \mathsf{EncData}(\mathsf{crs}, D_j)$ for $\forall j \in [n]$.
- $(\mathsf{ct}_0, \mathrm{x\_secret}) \leftarrow \mathsf{InpEnc}(\mathsf{crs}, x)$.
- $\mathsf{ct}_j \leftarrow \mathsf{Eval}\left(\mathsf{crs}, j, \{\mathsf{pk}_k\}_{k=j+1}^n, \mathsf{ct}_{j-1}, \mathsf{sk}_j, (P_j, t_j), \mathsf{digest}_j\right)$ for $\forall j \in [n]$.
- Output $\mathsf{ct}_0, \{\widetilde{D}_j, \mathsf{ct}_j\}_{j\in[n]}$.

Simulated experiment

- $(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathsf{ihopSim}(1^\kappa, i)$.
- $(\mathsf{pk}_j, \mathsf{sk}_j) \leftarrow \mathsf{KeyGen}(1^\kappa; r_j)$ for $\forall j \in [n] \setminus \{i\}$. Here, r_j are uniform random coins.
- $\left(\mathsf{ct}_0, \{\widetilde{D}_j, \mathsf{ct}_j\}_{j\in[n]}\right) \leftarrow \mathsf{ihopSim}(\mathsf{crs}, x, \{\mathsf{pk}_j, \mathsf{sk}_j, D_j, t_j\}_{j\in[n]}, \{P_j, r_j\}_{j\in[n]\setminus\{i\}}, \mathsf{MemAccess}_i, y_i)$, where $y_i = {P_i}^{D_i}\left(\cdots\left({P_1}^{D_1}(x)\right)\cdots\right)$.
- Output $\mathsf{ct}_0, \{\widetilde{D}_j, \mathsf{ct}_j\}_{j\in[n]}$.

The above definition can be made semi-malicious by allowing the adversary to pick random coins r_j adversarially given the public key pk_i of honest client as follows: $\{r_j\}_{j\in[n]\setminus\{i\}} \leftarrow \mathcal{A}(1^\kappa, \mathsf{crs}, \mathsf{pk}_i)$ that will be used to define $(\mathsf{pk}_j, \mathsf{sk}_j)$ in Step 2. Our proof would also support this stronger setting as well.

Construction of ihopSim: We describe the two phases of ihopSim. In the first phase, ihopSim generates the keys of honest client Q_i as $(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathsf{KeyGen}(1^\kappa)$.

In the second phase, ihopSim is described in Figure 4.10. At a high level, ihopSim generates everything honestly except ct_i. When generating ct_i, it simulates the step circuits one by one from the last to the first using the output y_i and memory access $\mathsf{MemAccess}_i$. In particular, since ihopSim takes D_i and $\mathsf{MemAccess}_i$ as input, it can compute D_i and digest_i before every step circuit, and use that to compute the output of every step circuit. Security follows from security of re-randomization of SFE, namely re-randomized garbled circuits are indistinguishable from freshly generated ones and that freshly generated garbled circuits are indistinguishable from simulated ones.

Now we give a series of hybrids such that the first hybrid outputs $\left(\mathsf{ct}_0, \{\widetilde{D}_j, \mathsf{ct}_j\}_{j\in[n]}\right)$ in the real execution, and the last hybrid is the output of ihopSim. Notice that the only difference between the real and ideal experiments is ct_i, so all the hybrids generate everything in the same way except ct_i.

- $\hat{\mathsf{H}}_0$: Output in the real experiment.

$\left(\mathsf{ct}_0, \{\widetilde{D}_j, \mathsf{ct}_j\}_{j\in[n]}\right) \leftarrow \mathsf{ihopSim}\left(\mathsf{crs}, x, \{\mathsf{pk}_j, \mathsf{sk}_j, D_j, t_j\}_{j\in[n]}, \{P_j\}_{j\in[n]\setminus\{i\}}, \mathsf{MemAccess}_i, y_i\right)$.

1. Compute $\widetilde{D}_j = (\mathsf{digest}_j, \hat{\mathsf{D}}_j) \leftarrow \mathsf{EncData}(\mathsf{crs}, D_j)$ for $\forall j \in [n]$.
   Compute $(\mathsf{ct}_0, \mathrm{x\_secret}) \leftarrow \mathsf{InpEnc}(x)$.
   Compute $\mathsf{ct}_j \leftarrow \mathsf{Eval}\left(j, \{\mathsf{pk}_k\}_{k=j+1}^n, \mathsf{ct}_{j-1}, \mathsf{sk}_j, (P_j, t_j), \mathsf{digest}_j\right)$ for every $j \in [i-1]$.
   Pick a random function F (in the following use random values for $F(\cdot)$).
2. Let $T_j := \sum_{k\in[j]} t_k$. Generate ct_i as follows:

   a) Run the program ${P_{i-1}}^{D_{i-1}}\left(\cdots\left({P_1}^{D_1}(x)\right)\cdots\right)$ to obtain $(\mathsf{R/W}^\tau, L^\tau, \mathsf{wData}^\tau)$ for every CPU step $\tau \in [T_{i-1}]$. Obtain $(\mathsf{R/W}^\tau, L^\tau, \mathsf{wData}^\tau)$ for $\tau \in [T_{i-1}+1, T_i]$ from $\mathsf{MemAccess}_i$.

   b) $\left(\mathsf{stateKeys}^{T_i+1}, \mathsf{dataKeys}^{T_i+1}, \mathsf{digestKeys}^{T_i+1}\right) \leftarrow \mathsf{SampleKeys}(0^*)$.
   Compute $\mathsf{stateLabels}^{T_i+1}$ using y_i; compute $\mathsf{dataLabels}^{T_i+1}, \mathsf{digestLabels}^{T_i+1}$ using $(D_i, \mathsf{digest}_i, \mathsf{R/W}, L, \mathsf{wData})$ of the last CPU step.

   c) For $\tau = T_i$ downto 1, do the following:
   $(\mathsf{R/W}, L, \mathsf{wData}) := (\mathsf{R/W}^\tau, L^\tau, \mathsf{wData}^\tau)$.
   Define j s.t. $\tau \in [T_{j-1}+1, T_j]$.
   Let D be the database of Q_j before step τ.

   $\left(\mathsf{stateLabels}^{\tau+1}, \mathsf{dataLabels}^{\tau+1}, \mathsf{digestLabels}^{\tau+1}\right)$
   $\leftarrow \mathsf{Transform}\left((\mathsf{stateLabels}^{\tau+1}, \mathsf{dataLabels}^{\tau+1}, \mathsf{digestLabels}^{\tau+1}),\right.$
   $\left.\left\{F_{s_{i+1}}(\mathtt{GC\_}\star\,||\tau+1)\right\}_{\star\in\{\mathtt{STATE,DATA,DIGEST}\}} || \cdots || \left\{F_{s_n}(\mathtt{GC\_}\star\,||\tau+1)\right\}_{\star\in\{\mathtt{STATE,DATA,DIGEST}\}}\right)$.

   if $\mathsf{R/W} = \mathsf{read}$ **then**
   $\mathsf{e}_{\mathsf{data}} \leftarrow \ell\mathsf{OTSim}\left(\mathsf{crs}, D, L, \mathsf{dataLabels}^{\tau+1}\right)$.
   $X \leftarrow (\mathsf{stateLabels}^{\tau+1}, \mathsf{e}_{\mathsf{data}}, \mathsf{digestLabels}^{\tau+1})$.
   else
   $\mathsf{e}_{\mathsf{digest}} \leftarrow \ell\mathsf{OTSimWrite}\left(\mathsf{crs}, D, L, \mathsf{wData}, \mathsf{digestLabels}^{\tau+1}\right)$.
   $X \leftarrow (\mathsf{stateLabels}^{\tau+1}, \mathsf{dataLabels}^{\tau+1}, \mathsf{e}_{\mathsf{digest}}, \mathsf{wData})$

   $\left(\left\{\widetilde{\mathsf{C}}_\tau^{\mathsf{step}}, \{\widetilde{\mathsf{C}}_{\tau,j}^{\mathsf{PRF}}\}_{j=i+1}^n\right\}, \mathsf{Labels}^\tau\right) \leftarrow \mathsf{CircSim}\left(1^\kappa, \mathcal{U}, (X, \mathsf{R/W}, L)\right)$ such that the output labels of $\widetilde{\mathsf{C}}_{\tau,j}^{\mathsf{PRF}}$ are the same as input labels of $\mathsf{RdLabs}^{\tau,j}$ for $\widetilde{\mathsf{C}}_\tau^{\mathsf{step}}$.
   Parse $\mathsf{Labels}^\tau = \left(\mathsf{stateLabels}^\tau, \mathsf{dataLabels}^\tau, \mathsf{digestLabels}^\tau, \{\mathsf{PLabels}^{\tau,j}\}_{j\in[i+1,n]}\right)$.
   $\mathsf{L}_{\tau,j} \leftarrow \mathsf{OT}_2\left(\mathsf{pk}_j, (\mathsf{PLabels}^{\tau,j}, \mathsf{PLabels}^{\tau,j})\right)$ for every $j \in [i+1, n]$.

   if $\tau = T_{j-1}+1$ **then**
   Embed $\mathsf{stateLabels}^\tau$ and $\mathsf{digestLabels}^\tau$ in $\widetilde{\mathsf{C}}_\tau^{\mathsf{step}}$.
   $(\mathsf{stateKeys}^\tau, \mathsf{dataKeys}^\tau, \mathsf{digestKeys}^\tau) \leftarrow \mathsf{Transform}\Big(\mathsf{SampleKeys}(0^*),$
   $\left\{F_{s_j}(\mathtt{GC\_}\star\,||\tau)\right\} || \cdots || \left\{F_{s_{i-1}}(\mathtt{GC\_}\star\,||\tau)\right\} || \left\{F(\mathtt{GC\_}\star\,||\tau)\right\}_{\star\in\{\mathtt{STATE,DATA,DIGEST}\}}\Big)$.
   Compute $(\mathsf{dataLabels}^\tau, \mathsf{digestLabels}^\tau)$ using $(D_{j-1}, \mathsf{digest}_{j-1}, \mathsf{R/W}, L, \mathsf{wData})$ at step $\tau-1$.

   d) $\mathsf{L}_0 \leftarrow \mathsf{OT}_2\left(\mathsf{ct}_0, (\mathsf{stateLabels}^1, \mathsf{stateLabels}^1)\right)$.

   e) $\mathsf{ct}_i := \left(\mathsf{L}_0, \left\{\widetilde{\mathsf{C}}_\tau^{\mathsf{step}}, \{\widetilde{\mathsf{C}}_{\tau,j}^{\mathsf{PRF}}, \mathsf{L}_{\tau,j}\}_{j=i+1}^n\right\}_{\tau\in[T_i]}\right)$.

3. Compute $\mathsf{ct}_j \leftarrow \mathsf{Eval}\left(j, \{\mathsf{pk}_k\}_{k=j+1}^n, \mathsf{ct}_{j-1}, \mathsf{sk}_j, (P_j, t_j), \mathsf{digest}_j\right)$ for every $j \in [i+1, n]$.
4. Output $\mathsf{ct}_0, \{\widetilde{D}_j, \mathsf{ct}_j\}_{j\in[n]}$.

Figure 4.10: Simulator for multi-hop RAM.

1. Compute $\widetilde{D}_j = (\mathsf{digest}_j, \hat{\mathsf{D}}_j) \leftarrow \mathsf{EncData}(\mathsf{crs}, D_j)$ for $\forall j \in [n]$.
   Compute $(\mathsf{ct}_0, \mathrm{x\_secret}) \leftarrow \mathsf{InpEnc}(x)$.
   Compute $\mathsf{ct}_j \leftarrow \mathsf{Eval}\left(j, \{\mathsf{pk}_k\}_{k=j+1}^n, \mathsf{ct}_{j-1}, \mathsf{sk}_j, (P_j, t_j), \mathsf{digest}_j\right)$ for every $j \in [i-1]$.
   Pick a random function F (in the following use random values for $F(\cdot)$).
2. Let $T_j := \sum_{k\in[j]} t_k$. Generate ct_i as follows:
   a) Compute $\left\{\widetilde{\mathsf{C}}_\tau^{\mathsf{step}}, \{\widetilde{\mathsf{C}}_{\tau,j}^{\mathsf{PRF}}, \mathsf{L}_{\tau,j}\}_{j=i+1}^n\right\}_{\tau\in[m+1,T_i]}$ honestly as in Figure 4.4.
   b) Run the program $P_i^{D_i}\left(\cdots\left(P_1^{D_1}(x)\right)\cdots\right)$ to obtain $(\mathsf{state}^\tau, \mathsf{R/W}^\tau, L^\tau, \mathsf{wData}^\tau)$ for every $\tau \in [T_i]$.
   c) Define j s.t. $m \in [T_{j-1}+1, T_j]$.
   d) Set $\mathsf{stateKeys}^{m+1}, \mathsf{dataKeys}^{m+1}, \mathsf{digestKeys}^{m+1}$ as $\mathsf{SampleKeys}(F_{s_j}(\mathtt{GC\_}\star\,||m+1))$ where $\star \in \{\mathtt{STATE}, \mathtt{DATA}, \mathtt{DIGEST}\}$, respectively.
   If $m = T_j$, then set $\mathsf{stateKeys}^{m+1}, \mathsf{dataKeys}^{m+1}, \mathsf{digestKeys}^{m+1}$ to $\mathsf{SampleKeys}(0^*)$.
   If $j < i$, then $\left(\mathsf{stateKeys}^{m+1}, \mathsf{dataKeys}^{m+1}, \mathsf{digestKeys}^{m+1}\right)$
   $$\leftarrow \mathsf{Transform}\Big(\left(\mathsf{stateKeys}^{m+1}, \mathsf{dataKeys}^{m+1}, \mathsf{digestKeys}^{m+1}\right), \left\{F_{s_{j+1}}(\mathtt{GC\_}\star\,||m+1)\right\}\,||\cdots||\,\left\{F_{s_{i-1}}(\mathtt{GC\_}\star\,||m+1)\right\}\,||\,\{F(\mathtt{GC\_}\star\,||m+1)\}_{\star\in\{\mathtt{STATE},\mathtt{DATA},\mathtt{DIGEST}\}}\Big).$$
   e) Compute $(\mathsf{stateLabels}^{m+1}, \mathsf{dataLabels}^{m+1}, \mathsf{digestLabels}^{m+1})$ using $(\mathsf{state}^m, \mathsf{R/W}^m, L^m, \mathsf{wData}^m)$ and (D_j, digest_j) at step m.
   f) For $\tau = m$ downto 1, do the following:
   Follow the same steps as in Figure 4.10 step 2c.
   g) $\mathsf{L}_0 \leftarrow \mathsf{OT}_2\left(\mathsf{ct}_0, (\mathsf{stateLabels}^1, \mathsf{stateLabels}^1)\right)$.
   h) $\mathsf{ct}_i := \left(\mathsf{L}_0, \left\{\widetilde{\mathsf{C}}_\tau^{\mathsf{step}}, \{\widetilde{\mathsf{C}}_{\tau,j}^{\mathsf{PRF}}, \mathsf{L}_{\tau,j}\}_{j=i+1}^n\right\}_{\tau\in[T_i]}\right)$.
3. Compute $\mathsf{ct}_j \leftarrow \mathsf{Eval}\left(j, \{\mathsf{pk}_k\}_{k=j+1}^n, \mathsf{ct}_{j-1}, \mathsf{sk}_j, (P_j, t_j), \mathsf{digest}_j\right)$ for every $j \in [i+1, n]$.
4. Output $\mathsf{ct}_0, \{\widetilde{D}_j, \mathsf{ct}_j\}_{j\in[n]}$.

Figure 4.11: Decryption of hybrid H_m

- H_0: In this hybrid, replace $F_{s_i}(\cdot)$ with a truly random function F. In particular, when computing ct_i as in Figure 4.4, in steps 1a and 4, use the values generated by F; in step 3a embed labels corresponding to the values from F. The indistinguishability of this hybrid with $\hat{\mathsf{H}}_0$ follows from the pseudo-randomness of $F_{s_i}(\cdot)$ and privacy of oblivious transfer (s_i is hidden in pk_i).
- H_m ($m \in [T_i]$): Next we consider a sequence of hybrids $\mathsf{H}_1, \cdots, \mathsf{H}_{T_i}$. The description

$(\mathsf{R/W}, L, \mathsf{wData}) := (\mathsf{R/W}^\tau, L^\tau, \mathsf{wData}^\tau)$.
Define j s.t. $\tau \in [T_{j-1}+1, T_j]$.
Let (D, digest) be the database and digest of Q_j before step τ.
Let state' be the CPU state after step τ.

$\big(\mathsf{stateKeys}^{\tau+1}, \mathsf{dataKeys}^{\tau+1}, \mathsf{digestKeys}^{\tau+1}\big)$
$\leftarrow \mathsf{Transform}\big((\mathsf{stateKeys}^{\tau+1}, \mathsf{dataKeys}^{\tau+1}, \mathsf{digestKeys}^{\tau+1}),$
$\big\{F_{s_{i+1}}(\mathtt{GC}_{-} \star \,||\tau+1)\big\}_{\star\in\{\mathtt{STATE,DATA,DIGEST}\}} || \cdots || \,\{F_{s_n}(\mathtt{GC}_{-} \star \,||\tau+1)\}_{\star\in\{\mathtt{STATE,DATA,DIGEST}\}}\Big)$.

if $\mathsf{R/W} = \mathsf{read}$ **then**
$\mathsf{e}_{\mathsf{data}} \leftarrow \mathsf{Send}(\mathsf{crs}, \mathsf{digest}, L, \mathsf{dataKeys}^{\tau+1}; F(\mathtt{PSI}||\tau) \oplus \bigoplus_{j>i} F_{s_j}(\mathtt{LACONIC\_OT}||\tau))$.
$X \leftarrow (\mathsf{stateKeys}^{\tau+1}_{\mathsf{state}'}, \mathsf{e}_{\mathsf{data}}, \mathsf{digestKeys}^{\tau+1}_{\mathsf{digest}})$.
else
$\mathsf{e}_{\mathsf{digest}} \leftarrow \mathsf{SendWrite}\Big(\mathsf{crs}, \mathsf{digest}, L, \mathsf{wData}, \mathsf{digestKeys}^{\tau+1}; F(\mathtt{PSI}||\tau) \oplus \bigoplus_{j>i} F_{s_j}(\mathtt{LACONIC\_OT}||\tau)\Big)$.
$X \leftarrow (\mathsf{stateKeys}^{\tau+1}_{\mathsf{state}'}, \mathsf{dataKeys}^{\tau+1}_0, \mathsf{e}_{\mathsf{digest}}, \mathsf{wData})$.

$\left(\left\{\widetilde{\mathsf{C}}^{\mathsf{step}}_\tau, \{\widetilde{\mathsf{C}}^{\mathsf{PRF}}_{\tau,j}\}_{j=i+1}^n\right\}, \mathsf{Labels}^\tau\right) \leftarrow \mathsf{CircSim}\,(1^\kappa, \mathcal{U}, (X, \mathsf{R/W}, L))$ such that the output labels of $\widetilde{\mathsf{C}}^{\mathsf{PRF}}_{\tau,j}$ are the same as input labels of $\mathsf{RdLabs}^{\tau,j}$ for $\widetilde{\mathsf{C}}^{\mathsf{step}}_\tau$.
Parse $\mathsf{Labels}^\tau = \big(\mathsf{stateLabels}^\tau, \mathsf{dataLabels}^\tau, \mathsf{digestLabels}^\tau, \{\mathsf{PLabels}^{\tau,j}\}_{j\in[i+1,n]}\big)$.
$\mathsf{L}_{\tau,j} \leftarrow \mathsf{OT}_2\left(\mathsf{pk}_j, (\mathsf{PLabels}^{\tau,j}, \mathsf{PLabels}^{\tau,j})\right)$ for every $j \in [i+1, n]$.

if $\tau = T_{j-1}+1$ **then**
Embed $\mathsf{stateLabels}^\tau$ and $\mathsf{digestLabels}^\tau$ in $\widetilde{\mathsf{C}}^{\mathsf{step}}_\tau$.
$(\mathsf{stateKeys}^\tau, \mathsf{dataKeys}^\tau, \mathsf{digestKeys}^\tau) \leftarrow \mathsf{Transform}\Big(\mathsf{SampleKeys}(0^*),$
$\big\{F_{s_j}(\mathtt{GC}_{-} \star \,||\tau)\big\} || \cdots || \,\big\{F_{s_{i-1}}(\mathtt{GC}_{-} \star \,||\tau)\big\} || \,\{F(\mathtt{GC}_{-} \star \,||\tau)\}_{\star\in\{\mathtt{STATE,DATA,DIGEST}\}}\Big)$.
Compute $(\mathsf{dataLabels}^\tau, \mathsf{digestLabels}^\tau)$ using $(D_{j-1}, \mathsf{digest}_{j-1}, \mathsf{R/W}, L, \mathsf{wData})$ at step $\tau - 1$.

Figure 4.12: Difference of H_m and $\hat{\mathsf{H}}_m$.

of H_m is in Figure 4.11. Notice that ct_i consists of T_i step circuits with corresponding PRF circuits. In hybrid H_m, the step circuits from 1 to m are simulated while the remaining step circuits ($m+1$ to T_i) are generated honestly. Given all the programs and secret keys, the intermediate outputs as well as input/output labels of every step circuit can all be computed. Given the correct output of circuit $\mathsf{C}^{\mathsf{step}}_m$, the step circuits from 1 to m can be simulated one by one from the m-th to the first similarly as in $\mathsf{ihopSim}$.

To show H_m is indistinguishable from H_{m-1}, first notice that they are the same except $\left(\widetilde{\mathsf{C}}^{\mathsf{step}}_m, \{\widetilde{\mathsf{C}}^{\mathsf{PRF}}_{m,j}, \mathsf{L}_{m,j}\}_{j=i+1}^n\right)$ in ct_i. Consider an intermediate hybrid $\hat{\mathsf{H}}_m$ which is the same as H_m except that in step 2f when $\tau = m$, follow the steps in Figure 4.12. In particular, when $\tau = m$, $\hat{\mathsf{H}}_m$ computes the output of $\mathsf{C}^{\mathsf{step}}_m$ and uses that output to

simulate $\widetilde{\mathsf{C}}_m^{\mathsf{step}}$ by $\mathsf{CircSim}$ and OT_2. The output of $\widetilde{\mathsf{C}}_m^{\mathsf{step}}$ is the same for H_{m-1} and $\hat{\mathsf{H}}_m$. The indistinguishability of $\hat{\mathsf{H}}_m$ and H_{m-1} follows from the security of garbled circuits directly when $m \in [T_{i-1}+1, T_i]$. When $m \in [T_{i-1}]$, it follows from the security of garbled circuits and re-randomization. More precisely, the re-randomized garbled circuit is indistinguishable from a freshly generated garbled circuit, which is indistinguishable from a simulated one. Notice that the random coins used in re-randomization for $\widetilde{\mathsf{C}}_m^{\mathsf{step}}$ is $F(\mathtt{GC}\_\star||m)$, which is not used anywhere else in H_{m-1}, so it can be treated as truly random coins.

To switch from $\hat{\mathsf{H}}_m$ to H_m, we replace X in Figure 4.12 with simulated $\mathsf{e}_{\mathsf{data}}$ and $\mathsf{e}_{\mathsf{digest}}$ for $\mathsf{CircSim}$ and OT_2. The indistinguishability follows from sender privacy of updatable laconic OT and that Send and $\mathsf{SendWrite}$ both take random coins $F(\mathtt{PSI}||m)$.

- $\hat{\mathsf{H}}_{T_i}$: Output in the simulated experiment. This hybrid is the same as H_{T_i}.

Extension. The above proof can be naturally extended to provide security for multiple clients and many executions. For example in the case of two clients Q_{i_1} and Q_{i_2}, $\mathsf{ihopSim}$ first computes $\left(\mathsf{ct}_0, \{\widetilde{D}_j\}_{j\in[n]}, \{\mathsf{ct}_j\}_{j\in[i_1-1]}\right)$ honestly, then computes ct_{i_1} same as in Figure 4.10 step 2. It then computes $\{\mathsf{ct}_j\}_{j\in[i_1+1,i_2-1]}$ from ct_{i_1} by Eval, and computes ct_{i_2} same as in Figure 4.10 step 2.[4] Finally it computes $\{\mathsf{ct}_j\}_{j\in[i_2+1,n]}$ from ct_{i_2} by Eval. To show this is indistinguishable from the real execution, we consider the following hybrids:

- H_0: Output in the real experiment.
- H_1: First compute $\left(\mathsf{ct}_0, \{\widetilde{D}_j\}_{j\in[n]}, \{\mathsf{ct}_j\}_{j\in[i_2-1]}\right)$ honestly, and then compute ct_{i_2} same as in Figure 4.10 step 2. Finally it computes $\{\mathsf{ct}_j\}_{j\in[i_2+1,n]}$ from ct_{i_2} honestly by Eval.
- H_2: Output in the simulated experiment.

The above hybrids are indistinguishable because an honestly generated ct_{i_1} or ct_{i_2} is indistinguishable from a simulated one, as we have shown in the single-client case.

To simulate multiple executions, $\mathsf{ihopSim}$ can simply repeat the procedure for every execution. We note that there is no connection between executions except the digest, so they can be simulated separately given the initial digests of every execution. The hybrids go from the real experiment to the simulated experiment by replacing all the honestly generated ct's in one execution by simulated ones, one execution per hybrid.

[4]Notice that different from Figure 4.10, the simulator here doesn't take P_{i_1} as input, but the simulation can still obtain $(\mathsf{R}/\mathsf{W}^\tau, L^\tau, \mathsf{wData}^\tau)$ for every step $\tau \in [T_{i_1-1}+1, T_{i_1}]$ given D_{i_1} and $\mathsf{MemAccess}_{i_1}$, and that is enough for the simulation.

4.3.5 Extending to Multiple Executions

Recall that for correctness we also proved that the after one execution, the resulting garbled database $\widetilde{D} = (\hat{\mathsf{D}}, \mathsf{digest})$ corresponds to the output of $\mathsf{Hash}(\mathsf{crs}, D')$, where D' is the correct database resulting after the execution in the clear (See Property 2, Section 4.3.3).

Given this invariant after the first execution, the next execution happens identically as the first execution with minor differences. To run the algorithm Eval, the clients need the updated digest of their respective databases. The updated digests of all the clients taking part in an execution would be sent by the server to the first client on that execution path, and would be passed along with each ciphertext. Also, to ensure that no PRF output is used twice, each PRF invocation would take the session identifier sid as an additional input. With these changes, the second execution is identical to the first execution and hence, its correctness follows in a straight-forward manner.

Also, this does not affect the UMA-security because the simulator of the ideal world is given the databases as well as memory access pattern of the honest clients as input.

Moreover, note that this generalizes to the scenario when different subset of clients take part in different executions. Only the digests of the relevant client are passed around by the server.[5]

4.4 From UMA to Full Security

In this section we provide a fully secure multi-hop RAM scheme. We first review Oblivious RAM (ORAM), which was first introduced by Goldreich [Gol87; GO96] and Ostrovsky [Ost90; Ost92; GO96], in Section 4.4.1. We then use ORAM as a compiler to encode the memory and program into a special format that does not reveal the access pattern or data contents during an execution. We present the generic construction in Section 4.4.2 and its correctness and security proofs in Sections 4.4.3 and 4.4.4, respectively.

4.4.1 Oblivious RAM

Definition 4.4.1. *An* Oblivious RAM *scheme consists of two procedures* $(\mathsf{OData}, \mathsf{OProg})$ *with syntax:*

- $(D^*, s^*) \leftarrow \mathsf{OData}(1^\kappa, D)$*: Given a security parameter* κ *and memory* $D \in \{0,1\}^M$ *as input,* OData *outputs the encoded memory* D^* *and encoding key* s^**.*
- $P^* \leftarrow \mathsf{OProg}(1^\kappa, 1^{\log M}, 1^t, P)$*: Given a security parameter* κ*, a memory size* M*, and a program* P *that runs in time* t*,* OProg *outputs an oblivious program* P^* *that can access* D^* *as RAM and takes two inputs* x *and* s^**.*

[5] It can also extended to the setting, when a client Q_i occurs multiple times in the chain of clients in an execution. To handle this setting, the digest of D_i is passed along all the programs between two instances of this client as additional state.

Efficiency. *We require that the run-time of* OData *should be* $M \cdot \mathsf{polylog}(M) \cdot \mathsf{poly}(\kappa)$*, and the run-time of* OProg *should be* $t \cdot \mathsf{poly}(\kappa) \cdot \mathsf{polylog}(M)$*. Finally, the oblivious program* P^* *itself should run in time* $t' = t \cdot \mathsf{poly}(\kappa) \cdot \mathsf{polylog}(M)$*. Both the new memory size* $M' = |D^*|$ *and the running time* t' *should be efficiently computable from* M, t*, and* κ*.*

Correctness. *Let* $P_1, \ldots, P_\ell$ *be programs running in polynomial times* $t_1, \ldots, t_\ell$ *on memory* D *of size* M*. Let* $x_1, \ldots, x_\ell$ *be the inputs and* κ *be a security parameter. Then we require that:*

$$\Pr[(P_1^*(x_1, s^*), \ldots, P_\ell^*(x_\ell, s^*))^{D^*} = (P_1(x_1), \ldots, P_\ell(x_\ell))^D] = 1$$

where $(D^*, s^*) \leftarrow \mathsf{OData}(1^\kappa, D)$*,* $P_i^* \leftarrow \mathsf{OProg}(1^\kappa, 1^{\log M}, 1^t, P_i)$ *and* $(P_1^*(x_1, s^*), \ldots, P_\ell^*(x_\ell, s^*))^{D^*}$ *indicates running the ORAM programs on* D^* *sequentially.*

Security. *For security, we require that there exists a PPT simulator* Sim *such that for any sequence of programs* $P_1, \ldots, P_\ell$*, initial memory data* $D \in \{0,1\}^M$*, and inputs* $x_1, \ldots, x_\ell$ *we have that:*

$$(D^*, \mathsf{MemAccess}) \overset{c}{\approx} \mathsf{Sim}(1^\kappa, 1^M, \{1^{t_i}, y_i\}_{i=1}^\ell)$$

where $(y_1, \ldots, y_\ell) = (P_1(x_1), \ldots, P_\ell(x_\ell))^D$*,* $(D^*, s^*) \leftarrow \mathsf{OData}(1^\kappa, D)$*, and* $\mathsf{MemAccess}$ *corresponds to the access pattern of the CPU-step circuits during the sequential execution of the oblivious programs* $(P_1^*(x_1, s^*), \ldots, P_\ell^*(x_\ell, s^*))^{D^*}$*.*

4.4.2 Generic Construction

We prove the following theorem.

Theorem 4.4.2. *Assume there exists a UMA-secure multi-hop RAM scheme and an ORAM scheme. Then there exists a fully secure multi-hop RAM scheme. Moreover, we give a black-box construction of one given a UMA-secure multi-hop RAM and ORAM scheme.*

Proof. We first give the construction of the scheme itself and then provide a construction of an appropriate simulator to prove security. Let $(\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{EncData}, \mathsf{InpEnc}, \mathsf{Eval}, \mathsf{Dec})$ be a UMA-secure multi-hop RAM scheme and let $(\mathsf{OData}, \mathsf{OProg})$ be an ORAM scheme. We construct a new multi-hop RAM scheme $(\widehat{\mathsf{Setup}}, \widehat{\mathsf{KeyGen}}, \widehat{\mathsf{EncData}}, \widehat{\mathsf{InpEnc}}, \widehat{\mathsf{Eval}}, \widehat{\mathsf{Dec}})$ as follows:

- $\widehat{\mathsf{Setup}}(1^\kappa)$: Generate crs same as Setup.
- $\widehat{\mathsf{KeyGen}}(1^\kappa)$: Generate $(\mathsf{pk}, \mathsf{sk})$ same as KeyGen.
- $\widehat{\mathsf{EncData}}(\mathsf{crs}, D)$: Execute $(D^*, s^*) \leftarrow \mathsf{OData}(1^\kappa, D)$ followed by $\widetilde{D} \leftarrow \mathsf{EncData}(1^\kappa, D^*)$.
- $\widehat{\mathsf{InpEnc}}(x)$: Execute $(\mathsf{ct}, \mathrm{x\_secret}) \leftarrow \mathsf{InpEnc}(x)$.

- $\widehat{\mathsf{Eval}}\left(i, \{\mathsf{pk}_j\}_{j=i+1}^n, \mathsf{ct}, \mathsf{sk}, (P,t), \mathsf{digest}\right)$: Execute $(P^*, t^*) \leftarrow \mathsf{OProg}(1^\kappa, 1^{\log M}, 1^t, P)$ followed by $\mathsf{Eval}\left(i, \{\mathsf{pk}_j\}_{j=i+1}^n, \mathsf{ct}, \mathsf{sk}, (P^*[s^*], t^*), \mathsf{digest}\right)$, where P^* has s^* hard-coded inside it.
- $\widehat{\mathsf{Dec}}^{\widetilde{D}_1,\cdots,\widetilde{D}_n}(\text{x\_secret}, \mathsf{ct})$: Output $\mathsf{Dec}^{\widetilde{D}_1,\cdots,\widetilde{D}_n}(\text{x\_secret}, \mathsf{ct})$.

We prove that the construction above given by $(\widehat{\mathsf{Setup}}, \widehat{\mathsf{KeyGen}}, \widehat{\mathsf{EncData}}, \widehat{\mathsf{InpEnc}}, \widehat{\mathsf{Eval}}, \widehat{\mathsf{Dec}})$ is a fully secure multi-hop RAM scheme.

4.4.3 Correctness

For a Single Execution. First we prove correctness for a single execution, and then we will generalize to multiple executions. In a single execution, our goal is to demonstrate that

$$\mathsf{Pr}\left[\widehat{\mathsf{Dec}}^{\widetilde{D}_1,\cdots,\widetilde{D}_n}(\text{x\_secret}, \mathsf{ct}_n) = {P_n}^{D_n}\left(\cdots\left({P_1}^{D_1}(x)\right)\cdots\right)\right] = 1,$$

where $\widetilde{D}_i \leftarrow \widehat{\mathsf{EncData}}(1^\kappa, D_i)$, $(\mathsf{ct}_0, \text{x\_secret}) \leftarrow \widehat{\mathsf{InpEnc}}(x), \mathsf{ct}_i \leftarrow \widehat{\mathsf{Eval}}(i, \{\mathsf{pk}_j\}_{j=i+1}^n, \mathsf{ct}_{i-1}, \mathsf{sk}, P_i, t_i, \mathsf{digest}_i)$.

By definition, $\widehat{\mathsf{Dec}}^{\widetilde{D}_1,\cdots,\widetilde{D}_n}(\text{x\_secret}, \mathsf{ct}_n) = \mathsf{Dec}^{\widetilde{D}_1,\cdots,\widetilde{D}_n}(\text{x\_secret}, \mathsf{ct}_n)$. By the correctness of the UMA-secure multi-hop RAM scheme, we have that $\mathsf{Dec}^{\widetilde{D}_1,\cdots,\widetilde{D}_n}(\text{x\_secret}, \mathsf{ct}_n) = P_n^*[s_n^*]^{D_n^*}\left(\cdots\left(P_1^*[s_1^*]^{D_1^*}(x)\right)\cdots\right)$. Finally, by the correctness of the ORAM scheme, it holds that $P_n^*[s_n^*]^{D_n^*}\left(\cdots\left(P_1^*[s_1^*]^{D_1^*}(x)\right)\cdots\right) = {P_n}^{D_n}\left(\cdots\left({P_1}^{D_1}(x)\right)\cdots\right)$.

For Multiple Executions. To prove correctness in multiple executions, we need to show that

$$\mathsf{Pr}\left[\widehat{\mathsf{Dec}}^{\widetilde{D}_1^{(\mathsf{sid})},\cdots,\widetilde{D}_n^{(\mathsf{sid})}}\left(\text{x\_secret}^{(\mathsf{sid})}, \mathsf{ct}_n^{(\mathsf{sid})}\right) = {P_n^{(\mathsf{sid})}}^{D_n^{(\mathsf{sid})}}\left(\cdots\left({P_1^{(\mathsf{sid})}}^{D_1^{(\mathsf{sid})}}(x^{(\mathsf{sid})})\right)\cdots\right)\right] = 1,$$

where $\widetilde{D}_i^{(\mathsf{sid})}$ is the resulting garbled database after executing sid-1 homomorphic evaluations, $(\mathsf{ct}_0^{(\mathsf{sid})}, \text{x\_secret}^{(\mathsf{sid})}) \leftarrow \widehat{\mathsf{InpEnc}}(x^{(\mathsf{sid})}), \mathsf{ct}_i^{(\mathsf{sid})} \leftarrow \widehat{\mathsf{Eval}}(i, \{\mathsf{pk}_j\}_{j=i+1}^n, \mathsf{ct}_{i-1}^{(\mathsf{sid})}, \mathsf{sk}, P_i^{(\mathsf{sid})}, t_i^{(\mathsf{sid})}, \mathsf{digest}_i^{(\mathsf{sid})})$.

Recall that for correctness of the UMA-secure multi-hop RAM scheme we proved that the after every execution, the resulting garbled database $\widetilde{D}_i^{(\mathsf{sid})}$ corresponds to the output of $\mathsf{Hash}(\mathsf{crs}, D_i^{(\mathsf{sid})})$, where $D_i^{(\mathsf{sid})}$ is the correct D_i^* resulting after previous $\mathsf{sid}-1$ executions in the clear (see Property 2, Section 4.3.3). By correctness of ORAM and the underlying UMA-secure multi-hop RAM scheme, we conclude the correctness in multiple executions.

4.4.4 Security

For a Single Client in a Single Execution. Server privacy is follows by receiver privacy of oblivious transfer. Now we prove client privacy for a single honest client Q_i in a single

execution. More precisely, we prove that there exists a PPT simulator $\mathsf{ihopSim}$ such that, for any set of databases $\{D_j\}_{j\in[n]}$, any sequence of compatible programs $P_1, \cdots, P_n$ running time $t_1, \cdots, t_n$ and input x, the outputs of the following two experiments are computational indistinguishable:

Real experiment

- $(\mathsf{pk}_j, \mathsf{sk}_j) \leftarrow \mathsf{KeyGen}(1^\kappa)$ for $\forall j \in [n]$.
- $\widetilde{D}_j = (\mathsf{digest}_j, \hat{\mathsf{D}}_j) \leftarrow \mathsf{EncData}(\mathsf{crs}, D_j)$ for $\forall j \in [n]$.
- $(\mathsf{ct}_0, \mathrm{x\_secret}) \leftarrow \mathsf{InpEnc}(x)$.
- $\mathsf{ct}_j \leftarrow \mathsf{Eval}\left(j, \{\mathsf{pk}_k\}_{k=j+1}^n, \mathsf{ct}_{j-1}, \mathsf{sk}_j, (P_j, t_j), \mathsf{digest}_j\right)$ for $\forall j \in [n]$.
- Output $\mathsf{ct}_0, \{\widetilde{D}_j, \mathsf{ct}_j\}_{j\in[n]}$.

Simulated experiment

- $(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathsf{ihopSim}(1^\kappa, i)$.
- $(\mathsf{pk}_j, \mathsf{sk}_j) \leftarrow \mathsf{KeyGen}(1^\kappa; r_j)$ for $\forall j \in [n] \setminus \{i\}$. Here, r_j are uniform random coins.
- $\left(\mathsf{ct}_0, \{\widetilde{D}_j, \mathsf{ct}_j\}_{j\in[n]}\right) \leftarrow \mathsf{ihopSim}\left(\mathsf{crs}, x, \{\mathsf{pk}_j, \mathsf{sk}_j, t_j\}_{j\in[n]}, \{D_j, P_j, r_j\}_{j\in[n]\setminus\{i\}}, 1^{M_i}, y_i\right)$, where $y_i = {P_i}^{D_i}\left(\cdots\left({P_1}^{D_1}(x)\right)\cdots\right)$.
- Output $\mathsf{ct}_0, \{\widetilde{D}_j, \mathsf{ct}_j\}_{j\in[n]}$.

We let OSim be the ORAM simulator, and USim be the simulator for the UMA-secure multi-hop RAM scheme. We describe the two phases of $\mathsf{ihopSim}$. In the first phase, $\mathsf{ihopSim}$ generates the keys of honest client Q_i as $(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathsf{KeyGen}(1^\kappa)$. In the second phase, $\mathsf{ihopSim}$ proceeds as follows.

1. Compute $(D_i^*, \mathsf{MemAccess}_i) \leftarrow \mathsf{OSim}(1^\kappa, 1^{M_i}, 1^{t_i}, y_i)$.
2. Compute (D_j^*, s_j^*) from $\widehat{\mathsf{EncData}}$ and P_j^* from $\widehat{\mathsf{Eval}}$ for every $j \in [n] \setminus \{i\}$.
3. Compute $\left(\mathsf{ct}_0, \{\widetilde{D}_j, \mathsf{ct}_j\}_{j\in[n]}\right) \leftarrow \mathsf{USim}(\mathsf{crs}, x, \{\mathsf{pk}_j, \mathsf{sk}_j, t_j^*\}_{j\in[n]}, \{D_j^*, P_j^*[s_j^*], r_j\}_{j\in[n]\setminus\{i\}}, D_i^*, \mathsf{MemAccess}_i, y_i)$.
4. Output $\left(\mathsf{ct}_0, \{\widetilde{D}_j, \mathsf{ct}_j\}_{j\in[n]}\right)$.

We now prove the output of the simulator is computationally indistinguishable from the real distribution.

- H_0: Output of the real experiment.

- H_1: Compute (D_j^*, s_j^*) from $\widehat{\mathsf{EncData}}$ and P_j^* from $\widehat{\mathsf{Eval}}$ for every $j \in [n] \setminus \{i\}$. Use the honestly generated (D_i^*, s_i^*) from $\widehat{\mathsf{EncData}}$ and P_i^* from $\widehat{\mathsf{Eval}}$ to execute the program $P_i^*[s_i^*]^{D_i^*}\left(\cdots\left(P_1^*[s_1^*]^{D_1^*}(x)\right)\cdots\right)$ and obtain y_i and a sequence of memory accesses $\mathsf{MemAccess}_i$. Then run $\left(\mathsf{ct}_0, \{\widetilde{D}_j, \mathsf{ct}_j\}_{j\in[n]}\right) \leftarrow \mathsf{USim}(\mathsf{crs}, x, \{\mathsf{pk}_j, \mathsf{sk}_j, t_j^*\}_{j\in[n]}, \{D_j^*, P_j^*[s_j^*], r_j\}_{j\in[n]\setminus\{i\}}, D_i^*, \mathsf{MemAccess}_i, y_i)$ and output.
  Since $(D_i^*, \mathsf{MemAccess}_i)$ is the same as the real execution, the indistinguishability of this hybrid and H_0 follows from UMA-security of the underlying multi-hop RAM scheme.
- H_2: Output of the simulated experiment. The only thing that differs in H_1 and H_2 is how we generate D_i^* and $\mathsf{MemAccess}_i$. In H_1 they are generated honestly and in H_2 they are generated by OSim. $\mathsf{H}_1 \overset{c}{\approx} \mathsf{H}_2$ follows from the security of ORAM.

For Multiple Clients in Multiple Executions. Similar as in the proof of UMA security, the above proof can be naturally extended to provide security for multiple clients and many executions. For example in the case of two clients Q_{i_1} and Q_{i_2}, $\mathsf{ihopSim}$ first computes $\left(\mathsf{ct}_0, \{\widetilde{D}_j\}_{j\in[n]}, \{\mathsf{ct}_j\}_{j\in[i_1-1]}\right)$ honestly, then simulates ct_{i_1} same as above as if there were only one honest client Q_{i_1}. It then computes $\{\mathsf{ct}_j\}_{j\in[i_1+1,i_2-1]}$ from ct_{i_1} by $\widehat{\mathsf{Eval}}$, and simulates ct_{i_2} same as above as if there were only one honest client Q_{i_2}. Notice that when simulating ct_{i_2}, similar as in the UMA-secure scenario, $\mathsf{ihopSim}$ cannot generate $(D_{i_1}^*, s_{i_1}^*, P_{i_1}^*)$ honestly. Instead it will use the simulated $(D_{i_1}^*, \mathsf{MemAccess}_{i_1})$ generated from OSim, and that is enough for the simulation. Finally it computes $\{\mathsf{ct}_j\}_{j\in[i_2+1,n]}$ from ct_{i_2} by $\widehat{\mathsf{Eval}}$. To show this is indistinguishable from the real execution, we consider the following hybrids:

- H_0: Output in the real experiment.
- H_1: First compute $\left(\mathsf{ct}_0, \{\widetilde{D}_j\}_{j\in[n]}, \{\mathsf{ct}_j\}_{j\in[i_2-1]}\right)$ honestly, and then compute ct_{i_2} same as above as if there were only one honest client Q_{i_2}. Finally it computes $\{\mathsf{ct}_j\}_{j\in[i_2+1,n]}$ from ct_{i_2} honestly by $\widehat{\mathsf{Eval}}$.
- H_2: Output in the simulated experiment.

The above hybrids are indistinguishable because an honestly generated ct_{i_1} or ct_{i_2} is indistinguishable from a simulated one, as we have shown in the single-client case.

To simulate multiple executions, $\mathsf{ihopSim}$ should first use OSim to simulate $(D_i^*, \mathsf{MemAccess}_i)$ for every honest client Q_i in all executions, and then repeat the above procedure for every execution. In the hybrids, we start from the real execution and first replace the honestly generated ct's by simulated ones while using honestly generated $(D_i^*, \mathsf{MemAccess}_i)$, and this step follows from the UMA-security of the underlying multi-hop RAM scheme. Afterwards we replace the honestly generated $(D_i^*, \mathsf{MemAccess}_i)$ by the output of OSim, and this step follows from the security of ORAM supporting multiple executions. □

Chapter 5

Password-Based Threshold Authentication

Token-based authentication is commonly used to enable a single-sign-on experience on the web, in mobile applications and on enterprise networks using a wide range of open standards and network authentication protocols: clients sign on to an identity provider using their username/password to obtain a cryptographic token generated with a master secret key, and store the token for future accesses to various services and applications. The authentication server(s) are single point of failures that if breached, enable attackers to forge arbitrary tokens or mount offline dictionary attacks to recover client credentials.

In this chapter, we introduce and formalize the notion of password-based threshold token-based authentication which distributes the role of an identity provider among n servers. Any t servers can collectively verify passwords and generate tokens, while no $t-1$ servers can forge a valid token or mount offline dictionary attacks. This serves as a special functionality of secure computation.

For this specific application of secure computation, we introduce PASTA, a general framework that can be instantiated using any threshold token generation scheme, wherein clients can "sign-on" using a 2-round (optimal) protocol that meets our strong notions of unforgeability and password-safety.

We instantiate and implement our framework in C++ using two threshold message authentication codes (MAC) and two threshold digital signatures with different trade-offs. Our experiments show that the overhead of protecting secrets and credentials against breaches in PASTA, i.e. compared to a naïve single server solution, is extremely low (1-5%) in the most likely setting where client and servers communicate over the internet. The overhead is higher in case of MAC-based tokens over a LAN (though still only a few milliseconds) due to public-key operations in PASTA. We show, however, that this cost is inherent by proving a symmetric-key only solution impossible.

Organization. We first give a technical overview in Section 5.1. Then we introduce necessary background in Section 5.2. An important building block, threshold oblivious pseudo-

random function (TOPRF) is presented in Section 5.3. We formalize password-based threshold token-based authentication and present our PASTA framework in Section 5.4 and show experimental results in Section 5.5. Finally we prove necessity of using public-key operations in Section 5.6.

5.1 Technical Overview

We start with a *plain* password-based token generation protocol that is insecure against server breaches. As briefly mentioned in Section 1.2, the plain protocol works as follows. In the registration phase, a client registers with its username/password by storing its username and hashed password $h = \mathsf{Hash}(\text{password})$ on the identity server. In the sign-on phase, client sends its username and hashed password h' to the server; server checks if $h' = h$ for the username. If the check passes, server then uses a master secret key msk to compute a token $auth_{msk}(x)$ and sends it to client, where $auth$ is either a MAC or a digital signature and x is the data to be signed. In this solution, both the master secret key msk and the hashed password h are compromised if the server is breached. Hence clients' passwords could be recovered using offline dictionary attacks.

Threshold Solution. A natural approach for protecting the master secret key msk is to combine the above plain solution with a threshold token generation (TTG) scheme (i.e. a threshold MAC or threshold signature). TTG schemes enable us to secret share msk among n servers such that any t servers can jointly generate valid tokens while any subset of up to $t-1$ servers cannot forge valid tokens or recover msk. To combine with the plain solution, the client registers to every server by sending its username and hashed password h in the registration phase. Then in the sign-on phase, client sends to t servers its username and hashed password h'. Every server checks if the $h = h'$ for the username, and performs its portion of the TTG scheme if the check passes. This solution guarantees the security of msk when at most $t-1$ servers are breached, but clients' passwords are still vulnerable against offline dictionary attacks even if a single server is breached.

Changing Secret Information Stored on Servers. The above two naïve solutions follow the same paradigm: server issues a token or executes the TTG scheme *only if client is using the correct password*. In order to check if client is using the correct password in the sign-on phase, server needs to store some "secret information" about client's password in the registration phase. In the above solutions, this secret information is the hashed password. A fundamental problem with this is that the secret information can be computed given only the password, hence enabling offline dictionary attacks on the password. To resolve this issue, we make the stored secret information also depend on a server-side secret.

This can be achieved by a threshold oblivious pseudorandom function (TOPRF) [FIPR05]. In a TOPRF protocol, a secret key k for a pseudorandom function F is initially shared among n servers. A client can obtain a PRF value of its password $h = F_k(\text{password})$ by interacting

with t servers, without revealing any information about its password to servers. Moreover, the function $F_k(\cdot)$ is computable by any t servers, but cannot be computed by up to $t-1$ servers. To this end, the PRF value $h = F_k(\text{password})$ serves as our new secret information stored on servers, and the protocol is now secure against offline dictionary attacks.

From Four Rounds to Two Rounds. A TOPRF protocol requires at least two rounds. Hence the sign-on phase in the above protocol requires at least four rounds: client and servers run the TOPRF protocol which requires two rounds for the client to obtain h; client then sends h back to servers as a third-round message; servers verify and respond with token shares of the TTG scheme as fourth-round messages. We would like to reduce the interaction to two rounds because network latency is a major bottleneck in the protocol especially over WAN networks (see Section 5.5.2 for details).

On the one hand, in order to prevent offline dictionary attack, we require that the "secret information" be computed jointly by client and servers, which requires at least two rounds. On the other hand, servers must ensure that generation of token is only performed after the secret information is checked, which also requires two rounds, so it seems that four rounds is necessary to achieve our goal.

We resolve this deadlock by observing that the check does not have to be done on the server side. Instead of checking the secret information and then participating in the TTG scheme to generate token shares, the servers generate token shares directly and encrypt them under the secret information h using a symmetric-key encryption scheme. The ciphertexts are sent along with the second-round message of the TOPRF protocol. Now the protocol only has two rounds, and the check is done on the client side: only if the client has used the correct password in the first round of TOPRF can it calculate the correct h and decrypt the ciphertexts to obtain t token shares, and combine them to recover the final token.

Mitigating Client Impersonation Attacks. There is still a subtle security problem. Consider an attacker who compromises a single server and retrieves the secret information h of a client, and then impersonates the client to which h belongs without knowing its password by participating in a sign-on protocol with the servers. The servers generate token shares, encrypt them under h, and send back to the attacker. Since the attacker already knows h, it can decrypt all the ciphertexts and combine the token shares to obtain a valid token without ever knowing the client's password or the master secret key. This issue occurs because when reducing the round complexity from four to two, we make the servers generate token shares without checking the secret information, but encrypt them using the secret information.

We address this issue by further modifying the secret information stored on servers. A client who computes h in the registration phase only sends $h_i = \mathsf{Hash}'(h, i)$ to server i where Hash' is assumed to be a random oracle. In other words, h is never revealed to or stored by any server, and each server only learns its corresponding h_i. Later in the sign-on phase, token shares are encrypted under the h_is. The client impersonation attack no longer works

since compromising certain servers only reveals the h_is of these servers to the attacker, while the remaining h_is are still kept secret.

Multi-Client Security. In our final protocol, we require that the only allowed attack is to impersonate a certain client and try different passwords by participating in an online sign-on protocol. This type of online attack is easy to detect in practice (e.g. if the same client is trying to sign-on very frequently within a short period of time). But enforcing the same across a large set of clients is not possible. Hence an important security requirement is that attacking one client should not help in attacking any other client.

This is not true for the protocol we have described so far. Consider an attacker who does not compromise any server, and performs the above online attack on one client[1], trying all possible passwords. As a result, the attacker would obtain all the PRF values $h = PRF_k(\text{password})$ for all possible passwords. Then the attacker impersonates another client by participating in a single sign-on protocol with the servers. Since the attacker already knows all possible PRF values, it could try decrypting the ciphertexts sent from servers using the collected dictionary of PRF values (offline) to find the correct value and hence recover the password. In other words, he can leverage his online attack against one client to perform offline attacks (after a single online interaction) on many other clients. Note that including client username as part of the input to the PRF does not solve the problem either since servers have no way of checking what username the attacker incorporates in the TOPRF protocol without adding expensive zero-knowledge proofs to this effect to the construction.

One natural idea is to have a distinct TOPRF key for every client, so that PRF values learned from one client would be useless for any other client. This means that servers need to generate a sufficiently large number of TOPRF keys in the global setup phase, which is not practical. There is a simple and efficient fix: we let every client generate its own TOPRF key and secret share it between servers in the registration phase. This yields our final protocol which we formally prove to meet all our security requirements under the gap TOMDH assumption [JKKX17] in the random oracle model.

Related Work. Password-based techniques are the most common methods for authenticating users. However, the traditional approach of storing hashed passwords on the servers is susceptible to offline dictionary attacks [WW15; Dan]. Standard remedies such as *salting* or more advanced remedies such as *memory-hard functions* [Scrypt; ACK+16; DKAN; BD16; ACP+17; BZ17], pursued in the recent password-hashing competition [PHC], surely make the task of the attacker harder, but do not resolve the fundamental issue of trusting a single server.

A large body of work considers distributed token generation through threshold digital signatures [DF90; DDFY94; GHKR08; DK01; Bol03; AMN01; GJKR96; GGN16; Sho00;

[1] A smarter attacker would distribute its online attack across many clients to avoid detection. We use the single client in this example just to highlight the underlying multi-client security issue.

BS01] and threshold message authentication codes [BLMR13; NPR99; MPS+03] which can protect the master key against $t-1$ breached servers. A separate line of work on threshold password-authenticated key exchange (T-PAKE) [MSJ02; DG03; AFP05; ACFP05; CLN15; KMTG05] aims to prevent offline dictionary attacks in standard password-authenticated key exchange (PAKE) [BPR00; BMP00; KOY01; GK10; KV11; CHK+05; PS10; KOY03] by employing multiple servers.

While PAKE and T-PAKE solve the problem of establishing a secret key between a server and a client, where the client authenticates with a password, they do not solve the problem posed in this paper of distributing trust in password-based token generation. Specifically, PbTA generates tokens and provides token unforgeability, which T-PAKE does not deal with. Moreover, PbTA works in a setting where multiple clients share the same token generation set-up, and guarantees that attacks on one client do not affect the security of others. Finally, PbTA has a per-client registration phase that further differentiates it from PAKE and T-PAKE.

It is also worth noting that a straightforward composition of a T-PAKE followed by a threshold signature/MAC meets neither the efficiency nor the security requirements for PbTA. For efficiency, recall that we require minimal interaction where servers need not communicate with each other after a one-time setup procedure, and both the password verification and the token generation can be performed simultaneously in two rounds. The most efficient T-PAKE schemes require at least three rounds of interaction between the client and servers and additional communication among the servers (which could further increase when combined with threshold token generation). For security, it is unclear how to make such a composition meet our strong unforgeability and password-safety properties which we elaborate on shortly.

Another line of work focuses on constructing password-based server-aided signatures [CLNS16; XS03; Gan95; GT11; MR01]. However, they assume that apart from the password, a client also needs to use a secret state (e.g. a shared secret key) to generate a signature. In contrast, we focus on a solution in which a client only needs to use a password to generate a signature (more generally, a token).

Password-protected secret sharing (PPSS) [BJSL11; JKKX17; ACNP16; JKK14; YHCL15; CLLN14; CLN12; CEN15; JKKX16] considers the related problem of sharing a secret among multiple servers where t servers can reconstruct the secret if client's password verifies. This line of work does not meet our goal of keeping the master secret distributed at all times for use in a threshold token generation scheme. Moreover, PPSS is commonly studied in a single client setting where each client has its own unique secret. As we will see shortly, the multi-client setting and the common master-key used for all clients introduces additional technical challenges.

A very recent work of Harchol et al. [HAP18] implements and uses similar building blocks to ours, i.e. a threshold oblivious PRF [JKKX17] and a proactive variant of threshold RSA signature scheme [Sho00]. But it uses them for the different end goal of distributing server secret keys and protecting client secret keys with a password in SSH implementations. As such, it neither formalizes nor addresses the security/efficiency requirements of a password-

based token generation scheme.

5.2 Background

We use κ to denote the security parameter. Let $\mathbb{Z}$ denote the set of all integers and $\mathbb{Z}_n$ the set $\{0, 1, 2, \ldots, n-1\}$. $\mathbb{Z}_n^*$ is defined as $\mathbb{Z}_n^* := \{x \in \mathbb{Z}_n | \gcd(x, n) = 1\}$. We use $[a, b]$ for $a, b \in \mathbb{Z}$, $a \leq b$, to denote the set $\{a, a+1, \ldots, b-1, b\}$. $[b]$ denotes the set $[1, b]$. $\mathbb{N}$ denotes the set of natural numbers.

We use $x \leftarrow_\$ S$ to denote that x is sampled uniformly at random from a set S. We use PPT as a shorthand for probabilistic polynomial time and negl to denote negligible functions.

We use $[\![a]\!]$ as a shorthand for $(a, a_1, \ldots, a_n)$ where $a_1, \ldots, a_n$ are *shares* of a. A concrete scheme will specify how the shares will be generated. The value of n will be clear from context.

We use a 'require' statement in the description of an oracle to enforce some checks on the inputs. If any of the checks fail, the oracle outputs $\perp$.

In a security game, we use $\langle \mathcal{O} \rangle$ to denote the collection of all the oracles defined in the game. For e.g., if a game defines oracles $\mathcal{O}_1, \ldots, \mathcal{O}_\ell$, then for an adversary $\mathcal{A}$, $\mathcal{A}^{\langle \mathcal{O} \rangle}$ denotes that $\mathcal{A}$ has access to the collection $\langle \mathcal{O} \rangle := (\mathcal{O}_1, \ldots, \mathcal{O}_\ell)$.

Shamir's Secret Sharing. Shamir's secret sharing is a simple way to generates shares of a secret so that a threshold of the shares are sufficient to reconstruct the secret, while any smaller number hides it completely. We consider a slightly more general form of Shamir's sharing here. Let $\mathsf{GenShare}$ be an algorithm that takes inputs p, n, t, $\{(i, \alpha_i)\}_{i \in S}$ s.t. $t \leq n < p$, p is prime, $S \subseteq [0, n]$ and $|S| < t$. It picks a random polynomial f of degree at most $t-1$ over $\mathbb{Z}_p$ s.t. $f(i) = \alpha_i$ for all $i \in S$, and outputs $f(0), f(1), \ldots, f(n)$.

To generate a (t, n)-Shamir sharing of a secret $s \in \mathbb{Z}_p$, $\mathsf{GenShare}$ is given p, n, t and $(0, s)$ as inputs to produce shares $s_0, s_1, \ldots, s_n$. Using the shorthand defined above, one can write the output compactly as $[\![s]\!]$. Given any t or more of the shares, say $\{s_j\}_{j \in T}$ for $|T| \geq t$, one can efficiently find coefficients $\{\lambda_j\}_{j \in T}$ such that $s = f(0) = \sum_{j \in T} \lambda_j \cdot s_j$. However, knowledge of up to $t-1$ shares reveals no information about s if it is chosen at random from $\mathbb{Z}_p$.

Cyclic Group Generator. Let $\mathsf{GroupGen}$ be a PPT algorithm that on input 1^κ outputs $(p, g, \mathbb{G})$ where $p = \Theta(\kappa)$, p is prime, $\mathbb{G}$ is a group of order p, and g is a generator of $\mathbb{G}$. We will use multiplication to denote the group operation.

5.2.1 Hardness Assumption

Threshold oblivious PRF (TOPRF) was introduced by Jarecki et al. [JKKX17] in a recent work. They propose a simple TOPRF protocol called 2HashTDH and prove that it is UC-secure under the Gap Threshold One-More Diffie-Hellman (Gap-TOMDH) assumption in

the random oracle model. They also show that Gap-TOMDH is hard in the generic group model.

Rather than modeling TOPRF as a functionality in the UC-sense, we will explicitly formalize two natural properties for it, obliviousness and unpredictability, in Section 5.3. We will show that Jarecki et al.'s 2HashTDH protocol satisfies these properties under the same assumption. Here, we formally state the assumption.

For $q_1, \ldots, q_n \in \mathbb{N}$ and $t', t \in \mathbb{N}$ where $t' < t \leq n$, define $\mathsf{MAX}_{t',t}(q_1, \ldots, q_n)$ to be the largest value of ℓ such that there exists *binary* vectors $\mathbf{u}_1, \ldots, \mathbf{u}_\ell \in \{0,1\}^n$ such that each $\mathbf{u}_i$ has $t - t'$ number of 1's in it and $(q_1, \ldots, q_n) \geq \sum_{i \in [\ell]} \mathbf{u}_i$. (All operations on vectors are component-wise integer operations.) Looking ahead, t and t' will be the parameters in the security definition of TOPRF and PbTA (t will be the threshold and t' the number of corrupted parties).

Definition 5.2.1 (Gap-TOMDH)**.** *A cyclic group generator* $\mathsf{GroupGen}$ *satisfies the* Gap Threshold One-More Diffie-Hellman *(Gap-TOMDH) assumption if for all* t', t, n, N *such that* $t' < t \leq n$ *and for all PPT adversary* $\mathcal{A}$*, there exists a negligible function* negl *s.t.* $\mathsf{One\text{-}More}_{\mathcal{A}}(1^\kappa, t', t, n, N)$ *(Figure 5.1) outputs* 1 *with probability at most* $\mathsf{negl}(\kappa)$*.*

In this game, a random polynomial of degree $t-1$ is picked but $\mathcal{A}$ gets to choose its value at t' points (steps 3 and 4). $\mathcal{A}$ gets access to two oracles:

- $\mathcal{O}$ allows it to compute x^{k_i}, where k_i is the value of the randomly chosen polynomial at i, for k_i that it does not know. A counter q_i is incremented for every such call.
- $\mathcal{O}_{\text{DDH}}$ allows it to check if the discrete log of g_2 w.r.t. g_1 is the same as the discrete log of h_2 w.r.t. h_1.

Intuitively, to compute a pair of the form (g, g^{k_0}), $\mathcal{A}$ should somehow get access to k_0. It clearly knows k_i for $i \in \mathcal{U}$, but shares outside $\mathcal{U}$ can only be obtained in the exponent, with the help of oracle $\mathcal{O}$. One option for $\mathcal{A}$ is to invoke $\mathcal{O}$ with (i, g) for at least $t - t'$ different values of i outside of $\mathcal{U}$, and then combine them together along with the k_i it knows to obtain g^{k_0}.

If $\mathcal{A}$ sticks to this strategy, it would have to repeat it entirely to compute h^{k_0} for a different base h. It could invoke $\mathcal{O}$ on different subsets of $[n]$ for different basis, but $\mathsf{MAX}_{t',t}(q_1, \ldots, q_n)$ will be the maximum number of pairs of type (x, x^{k_0}) it will be able to generate through this process.

Certainly, an adversary is not restricted to producing pairs in the way described above. However, Gap-TOMDH assumes that no matter what strategy a PPT adversary takes, it can effectively do no better than this.

5.2.2 Threshold Token Generation

A threshold token generation (TTG) scheme distributes the task of generating tokens for authentication among a set of n servers, such that at least a threshold t number of servers

$\underline{\mathsf{One\text{-}More}_{\mathcal{A}}(1^\kappa, t', t, n, N)}$:

1. $(p, g, \mathbb{G}) \xleftarrow{\$} \mathsf{GroupGen}(1^\kappa)$
2. $g_1, \ldots, g_N \leftarrow_\$ \mathbb{G}$
3. $(\{(i, \alpha_i)\}_{i \in \mathcal{U}}, \hat{\mathsf{D}}) \xleftarrow{\$} \mathcal{A}(p, g, \mathbb{G}, g_1, \ldots, g_N)$, where $\mathcal{U} \subseteq [n]$, $|\mathcal{U}| = t'$
4. $(k_0, k_1, \ldots, k_n) \xleftarrow{\$} \mathsf{GenShare}(p, n, t, \{(i, \alpha_i)\}_{i \in \mathcal{U}})$
5. $q_1, \ldots, q_n := 0$
6. $((g'_1, h_1), \ldots, (g'_\ell, h_\ell)) \xleftarrow{\$} \mathcal{A}^{\langle \mathcal{O} \rangle}(\hat{\mathsf{D}})$
7. output 1 iff
   - $\ell > \mathsf{MAX}_{t',t}(q_1, \ldots, q_n)$,
   - $\forall\ i \in [\ell]$, $g'_i \in \{g_1, \ldots, g_N\}$ and $h_i = g_i'^{k_0}$, and
   - $\forall\ i, j \in [\ell]$ s.t. $i \neq j$, $g'_i \neq g'_j$.

$\underline{\mathcal{O}(i, x)}$:

- require: $i \in [n] \setminus \mathcal{U}, x \in \mathbb{G}$
- increment q_i by 1
- return x^{k_i}

$\underline{\mathcal{O}_{\mathrm{DDH}}(g_1, g_2, h_1, h_2)}$

- require: $g_1, g_2, h_1, h_2 \in \mathbb{G}$
- return 1 iff $\exists\ a \in \mathbb{Z}_p$ s.t. $g_2 = g_1^a$ and $h_2 = h_1^a$

Figure 5.1: The Gap-TOMDH game.

must be contacted to compute a token. TTG provides a strong unforgeability guarantee: even if $t' < t$ of the servers are corrupt, any time a token on some new value x is needed, at least $t - t'$ servers must be contacted.

We formally define a TTG scheme and the unforgeability guarantee associated with it below.

Definition 5.2.2 (Threshold Token Generation)**.** *A threshold token generation scheme* TTG *is a tuple of four PPT algorithms (*Setup*,* PartEval*,* Combine*,* Verify*) that satisfies the consistency property below.*

- $\mathsf{Setup}(1^\kappa, n, t) \to (\llbracket \mathsf{sk} \rrbracket, \mathsf{vk}, \mathsf{pp})$*. It generates a secret key* sk*, shares* sk_1*,* sk_2*, …,* sk_n *of the key, a verification key* vk*, and public parameters* pp*. Share* sk_i *is given to party* i*. (*pp *will be an implicit input in the algorithms below.)*

- $\mathsf{PartEval}(\mathsf{sk}_i, x) \to y_i$. *It generates shares of token for an input. Party i computes the i-th share y_i for x by running* PartEval *with* sk_i *and* x.
- $\mathsf{Combine}(\{i, y_i\}_{i \in S}) =: \mathsf{tk}/\perp$. *It combines the shares received from parties in the set S to generate a token* tk. *If the algorithm fails, its output is denoted by $\perp$.*
- $\mathsf{Verify}(\mathsf{vk}, x, \mathsf{tk}) =: 1/0$. *It verifies whether token* tk *is valid for x or not using the verification key* vk. *(Output 1 denotes validity.)*

Consistency. For all $\kappa \in \mathbb{N}$, any $n, t \in \mathbb{N}$ such that $t \leq n$, all $([\![\mathsf{sk}]\!], \mathsf{vk}, \mathsf{pp})$ generated by $\mathsf{Setup}(1^\kappa, n, t)$, any value x, and any set $S \subseteq [n]$ of size at least t, if $y_i \xleftarrow{\$} \mathsf{PartEval}(\mathsf{sk}_i, x)$ for $i \in S$, then $\mathsf{Verify}(\mathsf{vk}, x, \mathsf{Combine}(\{(i, y_i)\}_{i \in S})) = 1$.

Definition 5.2.3 (Unforgeability)**.** *A threshold token generation scheme* $\mathsf{TTG} := (\mathsf{Setup}, \mathsf{PartEval}, \mathsf{Combine}, \mathsf{Verify})$ *is* unforgeable *if for all PPT adversaries $\mathcal{A}$, there exists a negligible function* negl *such the probability that the following game outputs 1 is at most* $\mathsf{negl}(\kappa)$.

$\underline{\mathsf{Unforgeability}_{\mathsf{TOP},\mathcal{A}}(1^\kappa, n, t)}$*:*

- Initialize. *Run* $\mathsf{Setup}(1^\kappa, n, t)$ *to get* $([\![\mathsf{sk}]\!], \mathsf{vk}, \mathsf{pp})$. *Give* pp *to* $\mathcal{A}$.
- Corrupt. *Receive the set of corrupt parties $\mathcal{U}$ from $\mathcal{A}$, where $t' := |\mathcal{U}| < t$. Give $\{\mathsf{sk}_i\}_{i \in \mathcal{U}}$ to $\mathcal{A}$.*
- Evaluate. *In response to $\mathcal{A}$'s query $(\mathtt{Eval}, x, i)$ for $i \in [n] \backslash \mathcal{U}$, return $y_i := \mathsf{PartEval}(\mathsf{sk}_i, x)$. Repeat this step as many times as $\mathcal{A}$ desires.*
- Challenge. *$\mathcal{A}$ outputs $(x^\star, \mathsf{tk}^\star)$. Check if*
  - $|\{i \mid \mathcal{A}$ *made a query* $(\mathtt{Eval}, x^\star, i)\}| < t - t'$ *and*
  - $\mathsf{Verify}(\mathsf{vk}, x^\star, \mathsf{tk}^\star) = 1$.

  Output 1 if and only if both checks succeed.

The unforgeability property captures the requirement that it must not be possible to generate a valid token on some value if less than $t - t'$ servers are contacted with that value.

Concrete Schemes. Below we describe the concrete threshold token generation schemes we implement.

- The DDH-based DPRF scheme of Naor, Pinkas and Reingold [NPR99] as a public-key threshold MAC (Figure 5.2).
- The PRF-only DRPF scheme of Naor, Pinkas and Reingold [NPR99] as a symmetric-key MAC (Figure 5.3).

- The threshold RSA-signature scheme of Shoup [Sho00] as a threshold signature scheme based on RSA assumption (Figure 5.4).
- The pairing-based signature scheme of Boldyreva [Bol03] as a threshold signature scheme based on the gap-DDH assumption (Figure 5.5).

Ingredients: Let $\mathbb{G} = \langle g \rangle$ be a multiplicative cyclic group of prime order p in which the DDH assumption holds and $\mathsf{Hash} : \{0,1\}^* \to \mathbb{G}$ be a hash function modeled as a random oracle. Let $\mathsf{GenShare}$ be Shamir's secret sharing scheme.

- $\mathsf{Setup}(1^\kappa, n, t) \to (\llbracket \mathsf{sk} \rrbracket, \mathsf{vk}, \mathsf{pp})$. Sample $s \leftarrow_\$ \mathbb{Z}_p$ and get $(s, s_1, \ldots, s_n) \leftarrow \mathsf{GenShare}(n, t, p, (0, s))$. Set $\mathsf{pp} := (p, g, \mathbb{G})$, $\mathsf{sk}_i := s_i$ and $\mathsf{vk} := s$. Give $(\mathsf{sk}_i, \mathsf{pp})$ to party i. (pp will be an implicit input in the algorithms below.).
- $\mathsf{PartEval}(\mathsf{sk}_i, x) \to y_i$. Compute $w := \mathsf{Hash}(x)$, $h_i := w^{\mathsf{sk}_i}$ and output h_i.
- $\mathsf{Combine}(\{i, y_i\}_{i \in S}) =: \mathsf{tk}/\perp$. If $|S| < t$ output $\perp$. Otherwise parse y_i as h_i for $i \in S$ and output $\prod_{i \in S} h_i^{\lambda_{i,S}}$
- $\mathsf{Verify}(\mathsf{vk}, x, \mathsf{tk}) =: 1/0$. Return 1 if and only if $\mathsf{Hash}(x)^{\mathsf{vk}} = \mathsf{tk}$.

Figure 5.2: DDH-based DPRF construction of Naor et al. [NPR99] (public-key threshold MAC).

Ingredients: Let $f : \{0,1\}^\kappa \times \{0,1\}^* \to \{0,1\}^*$ be a pseudo-random function.

- $\mathsf{Setup}(1^\kappa, n, t) \to (\llbracket \mathsf{sk} \rrbracket, \mathsf{vk}, \mathsf{pp})$. Pick $d := \binom{n}{n-t+1}$ keys $k_1, \ldots, k_d \leftarrow_\$ \{0,1\}^\kappa$ for f. Let $D_1, \ldots, D_d$ be the d distinct $(n-t+1)$-sized subsets of $[n]$. For $i \in [n]$, let $\mathsf{sk}_i := \{k_j \mid i \in D_j \text{ for all } j \in [d]\}$ and $\mathsf{vk} := (k_1, \ldots, k_d)$. Set $\mathsf{pp} := f$ and give $(\mathsf{sk}_i, \mathsf{pp})$ to party i.
- $\mathsf{PartEval}(\mathsf{sk}_i, x) \to y_i$. Compute $h_{i,k} := f_k(x)$ for all $k \in \mathsf{sk}_i$ and output $\{h_{i,k}\}_{k \in \mathsf{sk}_i}$.
- $\mathsf{Combine}(\{i, y_i\}_{i \in S}) =: \mathsf{tk}/\perp$. If $|S| < t$ output $\perp$. Otherwise parse y_i as $\{h_{i,k}\}_{k \in \mathsf{sk}_i}$ for $i \in S$. Let $\{\mathsf{sk}'_i\}_{i \in S}$ be mutually disjoint sets such that $\cup_{i \in S} \mathsf{sk}'_i = \{k_1, \ldots, k_d\}$ and $\mathsf{sk}'_i \subseteq \mathsf{sk}_i$ for every i. Output $\oplus_{k \in \mathsf{sk}'_i, i \in S}(h_{i,k})$.
- $\mathsf{Verify}(\mathsf{vk}, x, \mathsf{tk}) =: 1/0$. Return 1 if and only if $\oplus_{i \in [d]}(f_{k_i}(x)) = \mathsf{tk}$ where $\mathsf{vk} = \{k_1, \ldots, k_d\}$.

Figure 5.3: PRF-based DPRF of Naor et al. [NPR99] (symmetric-key threshold MAC).

Ingredients: Let $\mathsf{GenShare}$ be a Shamir's secret sharing scheme and $\mathsf{Hash} : \{0,1\}^* \to \mathbb{Z}_N^*$ be a hash function modeled as a random oracle.

- $\mathsf{Setup}(1^\kappa, n, t) \to (\llbracket \mathsf{sk} \rrbracket, \mathsf{vk}, \mathsf{pp})$. Let p', q' be two randomly chosen large primes of equal length and set $p := 2p' + 1$ and $q = 2q' + 1$. Set $N := pq$. Choose another large prime e at random and compute $d \equiv e^{-1} \bmod \Phi(N)$ where $\Phi(\cdot) : \mathbb{N} \to \mathbb{N}$ is the Euler's totient function. Then $(d, d_1, \ldots, d_n) \leftarrow \mathsf{GenShare}(n, t, \Phi(N), (0, d))$. Let $\mathsf{sk}_i := d_i$ and $\mathsf{vk} := (N, e)$. Set $\mathsf{pp} := \Delta$ where $\Delta := n!$. Give $(\mathsf{pp}, \mathsf{vk}, \mathsf{sk}_i)$ to party i.
- $\mathsf{PartEval}(\mathsf{sk}_i, x) \to y_i$. Output $y_i := \mathsf{Hash}(x)^{2\Delta d_i}$.
- $\mathsf{Combine}(\{i, y_i\}_{i \in S}) =: \mathsf{tk}/\perp$. If $|S| < t$ output $\perp$, otherwise compute $z := \prod_{i \in S} y_i^{2\lambda'_{i,S}} \bmod N$ where $\lambda'_{i,S} := \lambda_{i,S}\Delta \in \mathbb{Z}$. Find integer (a, b) by Extended Euclidean GCD algorithm such that $4\Delta^2 a + eb = 1$. Then compute $\mathsf{tk} := z^a \cdot \mathsf{Hash}(x)^b \bmod N$. Output tk.
- $\mathsf{Verify}(\mathsf{vk}, x, \mathsf{tk}) = 1/0$. Return 1 if and only if $\mathsf{tk}^e = \mathsf{Hash}(x) \bmod N$.

Figure 5.4: Threshold RSA-signature scheme of Shoup [Sho00].

Ingredients: Let $\mathbb{G} = \langle g \rangle$ be a multiplicative cyclic group of prime order p that supports pairing and in which CDH is hard. In particular, there is an efficient algorithm $\mathsf{VerDDH}(g^a, g^b, g^c, g)$ that returns 1 if and only if $c = ab \bmod p$ for any $a, b, c \in \mathbb{Z}_p^*$ and 0 otherwise. Let $\mathsf{Hash} : \{0,1\}^* \to \mathbb{G}$ be a hash function modeled as a random oracle. Let $\mathsf{GenShare}$ be the Shamir's secret sharing scheme.

- $\mathsf{Setup}(1^\kappa, n, t) \to (\llbracket \mathsf{sk} \rrbracket, \mathsf{vk}, \mathsf{pp})$. Sample $s \leftarrow_\$ \mathbb{Z}_p^*$ and get $(s, s_1, \ldots, s_n) \leftarrow \mathsf{GenShare}(n, t, p, (0, s))$. Set $\mathsf{pp} := (p, g, \mathbb{G})$, $\mathsf{sk}_i := s_i$ and $\mathsf{vk} := g^s$. Give $(\mathsf{sk}_i, \mathsf{pp})$ to party i.
- $\mathsf{PartEval}(\mathsf{sk}_i, x) \to y_i$. Compute $w := \mathsf{Hash}(x)$, $h_i := w^{\mathsf{sk}_i}$ and output h_i.
- $\mathsf{Combine}(\{i, y_i\}_{i \in S}) =: \mathsf{tk}/\perp$. If $|S| < t$ output $\perp$. Otherwise parse y_i as h_i for $i \in S$ and output $\prod_{i \in S} h_i^{\lambda_{i,S} \bmod p}$
- $\mathsf{Verify}(\mathsf{vk}, x, \mathsf{tk}) =: 1/0$. Return 1 if and only if $\mathsf{VerDDH}(\mathsf{Hash}(x), \mathsf{vk}, \mathsf{tk}, g) = 1$.

Figure 5.5: Pairing-based threshold signature scheme of Boldyreva [Bol03].

5.3 Threshold Oblivious Pseudo-Random Function

A pseudo-random function (PRF) family is a keyed family of deterministic functions. A function chosen at random from the family is indistinguishable from a random function. Oblivious PRF (OPRF) is an extension of PRF to a two-party setting where a server S

holds the key and a party P holds an input [FIPR05]. S can help P in computing the PRF value on the input but in doing so P should not get any other information and S should not learn P's input.

Jarecki et al. [JKKX17] extend OPRF to a multi-server setting so that a threshold number t of the servers are needed to compute the PRF on any input. Furthermore, a collusion of at most $t-1$ servers learns no information about the input. They propose a functionality for TOPRF and show how to realize it in a UC-secure way. We instead treat TOPRF as a set of algorithms that must satisfy two natural properties, unpredictability and obliviousness.

5.3.1 Syntax

Definition 5.3.1 (Threshold Oblivious Pseudo-Random Function)**.** *An $(\mathcal{X}, \mathcal{R})$-threshold oblivious pseudo-random function (TOPRF)* TOP *is a tuple of four PPT algorithms* (Setup, Encode, Eval, Combine) *that satisfies the consistency property below.*

- $\mathsf{Setup}(1^\kappa, n, t) \to (\llbracket \mathsf{sk} \rrbracket, \mathsf{pp})$. *It generates n secret key shares sk_1, sk_2, ..., sk_n and public parameters* pp. *Share sk_i is given to party i. (*pp *will be an implicit input in the algorithms below.)*
- $\mathsf{Encode}(x, \rho) =: c$. *It generates an encoding c of $x \in \mathcal{X}$ using randomness $\rho \in \mathcal{R}$.*
- $\mathsf{Eval}(\mathsf{sk}_i, c) =: z_i$. *It generates shares of TOPRF value from an encoding. Party i computes the i-th share z_i from c by running* Eval *with sk_i and c.*
- $\mathsf{Combine}(x, \{(i, z_i)\}_{i \in S}, \rho) =: h/\perp$. *It combines the shares received from parties in the set S using randomness ρ to generate a value h. If the algorithm fails, its output is denoted by $\perp$.*

Consistency. *For all $\kappa \in \mathbb{N}$, any $n, t \in \mathbb{N}$ such that $t \leq n$, all $(\llbracket \mathsf{sk} \rrbracket, \mathsf{pp})$ generated by $\mathsf{Setup}(1^\kappa, n, t)$, any value $x \in \mathcal{X}$, any randomness $\rho, \rho' \in \mathcal{R}$, and any two sets $S, S' \subseteq [n]$ of size at least t, if $c := \mathsf{Encode}(x, \rho)$, $c' := \mathsf{Encode}(x, \rho')$, $z_i := \mathsf{Eval}(\mathsf{sk}_i, c)$ for $i \in S$, and $z'_j := \mathsf{Eval}(\mathsf{sk}_j, c')$ for $j \in S'$, then $\mathsf{Combine}(x, \{(i, z_i)\}_{i \in S}, \rho) = \mathsf{Combine}(x, \{(j, z'_j)\}_{j \in S'}, \rho') \neq \perp$.*

Thus, irrespective of the randomness used to encode an x and the set of parties whose shares are combined, the output of Combine does not change (as long as Combine is given the same randomness used for encoding). We call this output the output of the TOPRF on x, and denote it by $\mathsf{TOP}(\mathsf{sk}, x)$.

Public Combine. We also consider a public combine algorithm PubCombine that could be run by anyone with access to just the partial evaluations. It would be used to check if a purported set of evaluations can lead to the right PRF value or not. Formally, for $Z := \{(i, z_i)\}_{i \in S}$ generated in the same manner as that for the consistency property, and any arbitrary $Z^\star := \{(i, z_i^\star)\}_{i \in S}$, if $\mathsf{PubCombine}(Z) = \mathsf{PubCombine}(Z^\star)$ then $\mathsf{Combine}(x, Z, \rho) =$

$\mathsf{Combine}(x, Z^\star, \rho)$. More importantly though, if the former equality does not hold then the later must not hold either (with high probability).

5.3.2 Security Properties

We want a TOPRF scheme to satisfy two properties, unpredictability and obliviousness. Unpredictability mandates that it must be difficult to predict TOPRF output on a random value, and obliviousness mandates that the random value itself is hard to guess even if the TOPRF output is available.

$\underline{\mathsf{Unpredictability}_{\mathsf{TOP},\mathcal{A}}(1^\kappa, n, t)}$:

- $([\![\mathsf{sk}]\!], \mathsf{pp}) \xleftarrow{\$} \mathsf{Setup}(1^\kappa, n, t)$
- $\mathcal{U} \xleftarrow{\$} \mathcal{A}(\mathsf{pp})$
- $\tilde{x} \leftarrow_\$ \mathcal{X}$
- $q_1, \ldots, q_n := 0$
- $h^\star \xleftarrow{\$} \mathcal{A}^{\langle\mathcal{O}\rangle}(\{\mathsf{sk}_i\}_{i\in\mathcal{U}})$
- output 1 iff $\mathsf{TOP}(\mathsf{sk}, \tilde{x}) = h^\star$

$\underline{\mathcal{O}_{\mathsf{enc\&eval}}()}$:

- $c := \mathsf{Encode}(\tilde{x}, \rho)$ for $\rho \leftarrow_\$ \mathcal{R}$
- for $i \in [n] \setminus \mathcal{U}$, $z_i \xleftarrow{\$} \mathsf{Eval}(\mathsf{sk}_i, c)$
- return $c, \{z_i\}_{i\in[n]\setminus\mathcal{U}}$

$\underline{\mathcal{O}_{\mathsf{eval}}(i, c)}$:

- require: $i \in [n] \setminus \mathcal{U}$
- increment q_i by 1
- return $\mathsf{Eval}(\mathsf{sk}_i, c)$

$\underline{\mathcal{O}_{\mathsf{check}}(h)}$:

- return 1 if $h = \mathsf{TOP}(\mathsf{sk}, \tilde{x})$; else return 0

Figure 5.6: The unpredictability game.

Definition 5.3.2 (Unpredictability)**.** *A* $(\mathcal{X}, \mathcal{R})$*-TOPRF* $\mathsf{TOP} := (\mathsf{Setup}, \mathsf{Encode}, \mathsf{Eval}, \mathsf{Combine})$ *is* unpredictable *if for all* $n, t \in \mathbb{N}$*,* $t \leq n$*, and PPT adversaries* $\mathcal{A}$*, there ex-*

ists a negligible function negl *s.t.*

$$\Pr[\mathsf{Unpredictability}_{\mathsf{TOP},\mathcal{A}}(1^\kappa, n, t) = 1] \leq \frac{\mathsf{MAX}_{|\mathcal{U}|,t}(q_1, \ldots, q_n)}{|\mathcal{X}|} + \mathsf{negl}(\kappa), \tag{5.1}$$

where $\mathsf{Unpredictability}$ *is defined in Figure 5.6.*

Our unpredictability definition provides several interfaces to an adversary $\mathcal{A}$. Oracle $\mathcal{O}_{\mathsf{enc\&eval}}$ can be called any number of times to get different sets of partial evaluations on the challenge input $\tilde{x}$, but the randomness used in this process is not revealed to $\mathcal{A}$. If no query is made to $\mathcal{O}_{\mathsf{eval}}$, so that none of the q_i change, then $\mathcal{A}$'s probability of guessing the TOPRF output on $\tilde{x}$ should be negligible (see Eq. (5.1)). In other words, any number of partial evaluations by themselves should not help at all.

$\mathcal{A}$ could, however, encode an arbitrary input itself, get partial evaluations through $\mathcal{O}_{\mathsf{eval}}$, and then combine them to learn the TOPRF output. It could also check if this output is same as the TOPRF output on the challenge input through $\mathcal{O}_{\mathsf{check}}$. Thus, by repeatedly querying $\mathcal{O}_{\mathsf{eval}}$, adversary can increase its chances of making the right guess. Eq. (5.1) requires that the probability of success should be no more than the maximum number of TOPRF outputs $\mathcal{A}$ can learn through this process over the size of password space. In some sense, this is the best we can hope to achieve.

Definition 5.3.3 (Obliviousness)**.** *An* $(\mathcal{X}, \mathcal{R})$*-TOPRF* $\mathsf{TOP} := (\mathsf{Setup}, \mathsf{Encode}, \mathsf{Eval}, \mathsf{Combine})$ *is* oblivious *if for all* $n, t \in \mathbb{N}$*,* $t \leq n$*, and all PPT adversaries* $\mathcal{A}$*, there exists a negligible function* negl *s.t.*

$$\Pr[\mathsf{Obliviousness}_{\mathsf{TOP},\mathcal{A}}(1^\kappa, n, t) = 1] \leq \frac{\mathsf{MAX}_{|\mathcal{U}|,t}(q_1, \ldots, q_n) + 1}{|\mathcal{X}|} + \mathsf{negl}(\kappa), \tag{5.2}$$

where $\mathsf{Obliviousness}$ *is defined in Figure 5.7.*

The obliviousness definition differs from unpredictability in small but crucial ways. Unlike the unpredictability game, $\mathcal{A}$ directly gets h from $\mathcal{O}_{\mathsf{enc\&eval}}$ because our goal is not to challenge the adversary on guessing the TOPRF output. $\mathcal{A}$ can still use $\mathcal{O}_{\mathsf{eval}}$ to learn new TOPRF outputs and check (by itself) it they match with h or not. Thus it can improve its chances of guessing $\tilde{x}$. The bound of Eq. 5.2 differs slightly from that of Eq. 5.1 though: there is an extra additive factor of 1 in the former case. This is to take into account that $\mathcal{A}$ can output a guess for $\tilde{x}$ different from the ones it has tried in the game.

5.3.3 Construction

We recall here the TOPRF construction, 2HashTDH, of Jarecki et al. [JKKX17, Section 3], for an input space $\mathcal{X}$. We refer to this construction as TOP from here onwards.

$\underline{\mathsf{Obliviousness}_{\mathsf{TOP},\mathcal{A}}(1^\kappa, n, t)}$:

- $([\![\mathsf{sk}]\!], \mathsf{pp}) \xleftarrow{\$} \mathsf{Setup}(1^\kappa, n, t)$
- $\mathcal{U} \xleftarrow{\$} \mathcal{A}(\mathsf{pp})$
- $\tilde{x} \leftarrow_{\$} \mathcal{X}$
- $q_1, \ldots, q_n := 0$
- $x^\star \xleftarrow{\$} \mathcal{A}^{\langle\mathcal{O}\rangle}(\{\mathsf{sk}_i\}_{i\in\mathcal{U}})$
- output 1 iff $x^\star = \tilde{x}$

$\underline{\mathcal{O}_{\mathsf{enc\&eval}}()}$:

- $c := \mathsf{Encode}(\tilde{x}, \rho)$ for $\rho \leftarrow_{\$} \mathcal{R}$
- for $i \in [n]$, $z_i \xleftarrow{\$} \mathsf{Eval}(\mathsf{sk}_i, c)$
- $h := \mathsf{Combine}(\tilde{x}, \{(i, z_i)\}_{i\in[n]}, \rho)$
- return $c, \{z_i\}_{i\in[n]\setminus\mathcal{U}}, h$

$\underline{\mathcal{O}_{\mathsf{eval}}(i, c)}$:

- require: $i \in [n] \setminus \mathcal{U}$
- increment q_i by 1
- return $\mathsf{Eval}(\mathsf{sk}_i, c)$

Figure 5.7: The obliviousness game.

- $\mathsf{Setup}(1^\kappa, n, t)$. Run $\mathsf{GroupGen}(1^\kappa)$ to get $(p, g, \mathbb{G})$. Pick an sk at random from $\mathbb{Z}_p$. Let $[\![\mathsf{sk}]\!] \xleftarrow{\$} \mathsf{GenShare}(p, n, t, (0, \mathsf{sk}))$ be a (t, n)-Shamir sharing of sk. Let $\mathsf{Hash}_1 : \mathcal{X} \times \mathbb{G} \to \{0,1\}^\kappa$ and $\mathsf{Hash}_2 : \mathcal{X} \to \mathbb{G}$ be hash functions. Output $[\![\mathsf{sk}]\!]$ and $\mathsf{pp} := (p, g, \mathbb{G}, n, t, \mathsf{Hash}_1, \mathsf{Hash}_2)$.
- $\mathsf{Encode}(x, \rho)$. Output $\mathsf{Hash}_2(x)^\rho$.
- $\mathsf{Eval}(\mathsf{sk}_i, c)$. Output c^{sk_i}.
- $\mathsf{Combine}(x, \{(i, z_i)\}_{i\in S}, \rho)$. If $|S| < t - 1$, output $\perp$. Else, use S to find coefficients $\{\lambda_i\}_{i\in S}$, compute $z := \prod_{i\in S} z_i^{\lambda_i}$, and output $\mathsf{Hash}_1(x \| z^{\rho^{-1}})$.

It is easy to see that TOP is a consistent $(\mathcal{X}, \mathbb{Z}_p^*)$-TOPRF scheme.

Public Combine. One can define $\mathsf{PubCombine}(\{(i, z_i)\}_{i\in S})$ for 2HashTDH to output z, the intermediate value in $\mathsf{Combine}$. Given x, z and ρ, the output of $\mathsf{Combine}$ is fixed. So, if an arbitrary set of partial evaluations produces the same z, $\mathsf{Combine}$ would output the same

thing. Moreover, if $\mathsf{PubCombine}$ produces a $z^\star$ different from z, then $z^{\star\rho^{-1}} \neq z^{\rho^{-1}}$, and output of $\mathsf{Combine}$ will be different with high probability (assuming that Hash_1 is collision-resistant).

5.3.4 Security Proof

We prove unpredictability and obliviousness of TOP in Sections 5.3.4.1 and 5.3.4.2, respectively.

5.3.4.1 Unpredictability

In this section we prove output unpredictability of our construction. Suppose there exists a PPT adversary $\mathcal{A}$ such that

$$\Pr[\mathsf{Unpredictability}_{\mathsf{TOP},\mathcal{A}}(1^\kappa, n, t) = 1] \geq \frac{\mathsf{MAX}_{|\mathcal{U}|,t}(q_1, \ldots, q_n)}{|\mathcal{X}|} + \mathsf{non\text{-}negl}(\kappa). \tag{5.3}$$

We will consider two cases of $\mathcal{A}$. In the first case, there exists $k \in \mathbb{N}$ such that when $\mathcal{A}$ calls Hash_1 with k distinct valid (x, y) pairs, $\mathsf{MAX}_{|\mathcal{U}|,t}(q_1, \ldots, q_n) < k$. In this case, we will use $\mathcal{A}$ to break the gap TOMDH assumption. In the second case, for any $k \in \mathbb{N}$, when $\mathcal{A}$ calls Hash_1 with k valid (x, y) pairs, $\mathsf{MAX}_{|\mathcal{U}|,t}(q_1, \ldots, q_n) \geq k$. In this case, we will prove that information theoretically Formula 5.3 does not hold.

First Case. There exists $k \in \mathbb{N}$ such that when $\mathcal{A}$ calls Hash_1 with k distinct valid (x, y) pairs, $\mathsf{MAX}_{|\mathcal{U}|,t}(q_1, \ldots, q_n) < k$. Then we construct an adversary $\mathcal{B}$ that breaks the gap TOMDH assumption (see Definition 5.2.1).

We construct $\mathcal{B}$ as follows. It first receives $(p, g, \mathbb{G}, g_1, \ldots, g_N)$ from the TOMDH game $\mathsf{One\text{-}More}_{\mathcal{B}}(1^\kappa, t', t, n, N)$, presents $\mathsf{pp} := (p, g, \mathbb{G}, n, t, \mathsf{Hash}_1, \mathsf{Hash}_2)$ to $\mathcal{A}$ and gets back $\mathcal{U}$. It then generates $\{\alpha_i\}_{i\in\mathcal{U}}$ at random, sends $\{(i, \alpha_i)\}_{i\in\mathcal{U}}$ to the TOMDH game, and sends $\{\alpha_i\}_{i\in\mathcal{U}}$ to $\mathcal{A}$. It then samples $\tilde{x} \leftarrow_\$ \mathcal{X}$, set $\mathcal{L} := []$, set $\mathsf{LIST} := \emptyset$, and set $k := 0$. Then $\mathcal{B}$ computes $g_1^{\alpha_i}$ for $i \in \mathcal{U}$, calls $\mathcal{O}(i, g_1)$ to get $g_1^{\alpha_i}$ for all $i \in [n] \setminus \mathcal{U}$, and computes $y_1 := g_1^{\mathsf{sk}}$. It adds (g_1, y_1) to LIST, sets $q = 1$, and handles $\mathcal{A}$'s oracle queries as follows:

- On $\mathcal{A}$'s call to $\mathcal{O}_{\mathsf{enc\&eval}}()$: Pick an unused g_j where $j \in [N]$, set $c := g_j$, compute $z_i \xleftarrow{\$} \mathsf{Eval}(\mathsf{sk}_i, c)$ for $i \in \mathcal{U}$ and call $\mathcal{O}(i, c)$ to get z_i for all $i \in [n] \setminus \mathcal{U}$. Use $[n]$ to find coefficients $\{\lambda_i\}_{i\in[n]}$ and compute $y_j := \prod_{i\in[n]} z_i^{\lambda_i}$. Add (g_j, y_j) to LIST, and increment q by 1. Compute $h := \mathsf{Hash}_1(\tilde{x}, y_j)$, and return $(c, \{z_i\}_{i\in[n]\setminus\mathcal{U}}, h)$ to $\mathcal{A}$.
- On $\mathcal{A}$'s call to $\mathcal{O}_{\mathsf{eval}}(i, c)$: Call $\mathcal{O}(i, c)$ in the TOMDH game and return the output to $\mathcal{A}$.
- On $\mathcal{A}$'s call to $\mathsf{Hash}_1(x, y)$: If $x \notin \mathcal{L}$, compute $\mathsf{Hash}_1(x, y)$ honestly and return to $\mathcal{A}$. Otherwise, let $g_j := \mathcal{L}[x]$. If $\log_{g_j} y = \log_{g_1} y_1$ and $(g_j, \star) \notin \mathsf{LIST}$ (i.e., (g_j, y) is a new valid pair), increment k by 1, add (g_j, y) to LIST, and output LIST in the TOMDH game if $\mathsf{MAX}_{|\mathcal{U}|,t}(q_1, \ldots, q_n) < k$. Compute $\mathsf{Hash}_1(x, y)$ honestly and return to $\mathcal{A}$.

- On $\mathcal{A}$'s call to $\mathsf{Hash}_2(x)$: If $x \in \mathcal{L}$, return $\mathcal{L}[x]$; otherwise, pick an unused g_j where $j \in [N]$, set $\mathcal{L}[x] := g_j$ and return g_j to $\mathcal{A}$.

$\mathcal{A}$'s view in the game $\mathsf{Unpredictability}_{\mathsf{TOP},\mathcal{A}}(1^\kappa, n, t)$ is information theoretically indistinguishable from the view simulated by $\mathcal{B}$ in the random oracle model. This can be proved via a hybrid argument:

H_0: The first hybrid is $\mathcal{A}$'s view in the real-world game $\mathsf{Unpredictability}_{\mathsf{TOP},\mathcal{A}}(1^\kappa, n, t)$.

H_1: This hybrid is the same as H_0 except that in the response to $\mathcal{O}_{\mathsf{enc\&eval}}()$, c is randomly sampled as $c \leftarrow_\$ \mathbb{G}$. This hybrid is information theoretically indistinguishable from H_0 because $\mathbb{G}$ is a cyclic group of prime order.

H_2: This hybrid is the same as H_1 except that the output of $\mathsf{Hash}_2(\cdot)$ is a truly random group element in $\mathbb{G}$. The indistinguishability of H_1 and H_2 follows from the random oracle model.

H_3: This hybrid is $\mathcal{A}$'s view simulated by $\mathcal{B}$. It is the same as H_2 except that the random group elements are replaced by g_j's where $j \in [N]$. Since g_j's are also randomly sampled from $\mathbb{G}$ in the TOMDH game, the two hybrids are indistinguishable.

From the construction of $\mathcal{B}$, we know that

$$\mathsf{MAX}_{t',t}(q'_1, \ldots, q'_n) = \mathsf{MAX}_{|\mathcal{U}|,t}(q_1, \ldots, q_n) + q,$$

where $\mathsf{MAX}_{t',t}(q'_1, \ldots, q'_n)$ is from the TOMDH game, and $\mathsf{MAX}_{|\mathcal{U}|,t}(q_1, \ldots, q_n)$ is from the unpredictability game. Since $\mathsf{MAX}_{|\mathcal{U}|,t}(q_1, \ldots, q_n) < k$, the output of $\mathcal{B}$ has the following number of valid pairs:

$$\begin{aligned} |\mathsf{LIST}| &= k + q \\ &> \mathsf{MAX}_{|\mathcal{U}|,t}(q_1, \ldots, q_n) + q \\ &= \mathsf{MAX}_{t',t}(q'_1, \ldots, q'_n). \end{aligned}$$

Therefore, $\mathcal{B}$ breaks the gap TOMDH assumption.

Second Case. For any $k \in \mathbb{N}$, when $\mathcal{A}$ calls Hash_1 with k valid (x, y) pairs, $\mathsf{MAX}_{|\mathcal{U}|,t}(q_1, \ldots, q_n) \geq k$.

We define a predicting game in Figure 5.8. Information theoretically we have that for any PPT adversary $\mathcal{A}$, there exists a negligible function negl s.t.

$$\Pr[\mathsf{Predicting}_{\mathcal{A}}(1^\kappa) = 1] \leq \frac{k}{|\mathcal{X}|} + \mathsf{negl}(\kappa).$$

We will use $\mathcal{A}$ to construct an adversary $\mathcal{B}$ that breaks the predicting game. The construction of $\mathcal{B}$ is the following. It first runs $\mathsf{Setup}(1^\kappa, n, t)$ to generate $([\![\mathsf{sk}]\!], \mathsf{pp})$, presents pp to $\mathcal{A}$ and gets back $\mathcal{U}$. It thens gives $\{\mathsf{sk}_i\}_{i \in \mathcal{U}}$ to $\mathcal{A}$. It sets $\mathcal{L} := []$, and then handles $\mathcal{A}$'s oracle queries as follows:

$\underline{\mathsf{Predicting}_{\mathcal{A}}(1^\kappa)}$:

1. for every $x \in \mathcal{X}$: $h \leftarrow_\$ \{0,1\}^\kappa$, $\mathcal{M}[x] := h$
2. $\tilde{x} \leftarrow_\$ \mathcal{X}$, $\tilde{h} := \mathcal{M}[\tilde{x}]$
3. $k := 0$
4. $h^\star \xleftarrow{\$} \mathcal{A}^{\langle\mathcal{O}\rangle}(1^\kappa)$
5. output 1 iff $h^\star = \tilde{h}$

$\underline{\mathcal{O}_{\mathsf{compute}}(x)}$:

- increment k by 1
- return $\mathcal{M}[x]$

$\underline{\mathcal{O}_{\mathsf{compare}}(h)}$:

- return 1 if $h = \tilde{h}$; else return 0

Figure 5.8: The predicting game.

- On $\mathcal{A}$'s call to $\mathcal{O}_{\mathsf{enc\&eval}}()$: Sample $c \leftarrow_\$ \mathbb{G}$, compute $z_i \xleftarrow{\$} \mathsf{Eval}(\mathsf{sk}_i, c)$ for $i \in [n]$, and return $(c, \{z_i\}_{i\in[n]\setminus\mathcal{U}})$ to $\mathcal{A}$.
- On $\mathcal{A}$'s call to $\mathcal{O}_{\mathsf{eval}}(i, c)$: Return $\mathsf{Eval}(\mathsf{sk}_i, c)$.
- On $\mathcal{A}$'s call to $\mathcal{O}_{\mathsf{check}}(h)$: Call $\mathcal{O}_{\mathsf{compare}}(h)$ and return the output to $\mathcal{A}$.
- On $\mathcal{A}$'s call to $\mathsf{Hash}_1(x, y)$:
  a. If $(x, y) \in \mathcal{L}$, let $h := \mathcal{L}[(x, y)]$.
  b. If $(x, y) \notin \mathcal{L}$ and $y \neq \mathsf{Hash}_2(x)^{\mathsf{sk}}$, then sample $h \leftarrow_\$ \{0,1\}^\kappa$ and set $\mathcal{L}[(x, y)] := h$.
  c. If $(x, y) \notin \mathcal{L}$ and $y = \mathsf{Hash}_2(x)^{\mathsf{sk}}$ (i.e., (x, y) is a valid pair), call $\mathcal{O}_{\mathsf{compute}}(x)$ to get h. Set $\mathcal{L}[(x, y)] := h$.

  Return h to $\mathcal{A}$.
- On $\mathcal{A}$'s call to $\mathsf{Hash}_2(x)$: Compute $\mathsf{Hash}_2(x)$ honestly and return the output.

$\mathcal{A}$'s view in the game $\mathsf{Unpredictability}_{\mathsf{TOP},\mathcal{A}}(1^\kappa, n, t)$ is information theoretically indistinguishable from the view simulated by $\mathcal{B}$ in the random oracle model. This can be proved via a hybrid argument:

H_0: The first hybrid is $\mathcal{A}$'s view in the real-world game $\mathsf{Unpredictability}_{\mathsf{TOP},\mathcal{A}}(1^\kappa, n, t)$.

H_1: This hybrid is the same as H_0 except that in the response to $\mathcal{O}_{\mathsf{enc\&eval}}()$, c is randomly sampled as $c \leftarrow_{\$} \mathbb{G}$. This hybrid is information theoretically indistinguishable from H_0 because $\mathbb{G}$ is a cyclic group of prime order.

H_2: This hybrid is the same as H_1 except that the output of $\mathsf{Hash}_1(\cdot)$ is a truly random string. The indistinguishability of H_1 and H_2 follows from the random oracle model.

H_3: This hybrid is $\mathcal{A}$'s view simulated by $\mathcal{B}$. It is the same as H_2 except that $\tilde{x}$ is not sampled in the game, but sampled in the predicting game $\mathsf{Predicting}_{\mathcal{B}}(1^\kappa)$, and that $\mathsf{Hash}_1(x, y)$ for valid (x, y) pairs are sampled in predicting game. Since these values are randomly sampled in both hybrids, they are indistinguishable.

Therefore, if $\mathcal{A}$ breaks the game $\mathsf{Unpredictability}_{\mathsf{TOP},\mathcal{A}}(1^\kappa, n, t)$, then $\mathcal{B}$ breaks the predicting game:

$$\begin{aligned}\Pr[\mathsf{Predicting}_{\mathcal{B}}(1^\kappa) = 1] &\geq \frac{\mathsf{MAX}_{|\mathcal{U}|,t}(q_1, \ldots, q_n)}{|\mathcal{X}|} + \mathsf{non\text{-}negl}(\kappa) \\ &\geq \frac{k}{|\mathcal{X}|} + \mathsf{non\text{-}negl}(\kappa).\end{aligned}$$

This is information theoretically impossible, leading to a contradiction, and hence concludes the proof.

5.3.4.2 Input Obliviousness

In this section we prove input obliviousness of our construction. Suppose there exists a PPT adversary $\mathcal{A}$ such that

$$\Pr[\mathsf{Obliviousness}_{\mathsf{TOP},\mathcal{A}}(1^\kappa, n, t) = 1] \geq \frac{\mathsf{MAX}_{|\mathcal{U}|,t}(q_1, \ldots, q_n) + 1}{|\mathcal{X}|} + \mathsf{non\text{-}negl}(\kappa). \tag{5.4}$$

We consider two cases of $\mathcal{A}$. In the first case, there exists $k \in \mathbb{N}$ such that when $\mathcal{A}$ calls Hash_1 with k distinct valid (x, y) pairs, $\mathsf{MAX}_{|\mathcal{U}|,t}(q_1, \ldots, q_n) < k$. In this case, we will use $\mathcal{A}$ to break the gap TOMDH assumption. In the second case, for any $k \in \mathbb{N}$, when $\mathcal{A}$ calls Hash_1 with k valid (x, y) pairs, $\mathsf{MAX}_{|\mathcal{U}|,t}(q_1, \ldots, q_n) \geq k$. In this case, we will prove that information theoretically Formula 5.4 does not hold.

First Case. There exists $k \in \mathbb{N}$ such that when $\mathcal{A}$ calls Hash_1 with k distinct valid (x, y) pairs, $\mathsf{MAX}_{|\mathcal{U}|,t}(q_1, \ldots, q_n) < k$. Then we construct an adversary $\mathcal{B}$ that breaks the gap TOMDH assumption (see Definition 5.2.1). The proof is the same as the first case in Section 5.3.4.1.

Second Case. Whenever $\mathcal{A}$ calls Hash_1 with a valid (x, y) pair for the k-th time, $\mathsf{MAX}_{|\mathcal{U}|,t}(q_1, \ldots, q_n) \geq k$ at that time.

We define a guessing game in Figure 5.9. Information theoretically we have that for any PPT adversary $\mathcal{A}$, there exists a negligible function negl s.t.

$$\Pr[\mathsf{Guessing}_{\mathcal{A}}(1^\kappa) = 1] \leq \frac{k+1}{|\mathcal{X}|} + \mathsf{negl}(\kappa).$$

$\underline{\mathsf{Guessing}_{\mathcal{A}}(1^\kappa)}$:

1. $\tilde{x} \leftarrow_\$ \mathcal{X}$
2. $k := 0$
3. $x^\star \xleftarrow{\$} \mathcal{A}^{\langle \mathcal{O} \rangle}(1^\kappa)$
4. output 1 iff $x^\star = \tilde{x}$

$\underline{\mathcal{O}_{\mathsf{guess}}(x)}$:

- increment k by 1
- return 1 if $x = \tilde{x}$; else return 0

Figure 5.9: The guessing game.

We will use $\mathcal{A}$ to construct an adversary $\mathcal{B}$ that breaks the guessing game. The construction of $\mathcal{B}$ is the following. It first runs $\mathsf{Setup}(1^\kappa, n, t)$ to generate $([\![\mathsf{sk}]\!], \mathsf{pp})$, presents pp to $\mathcal{A}$ and gets back $\mathcal{U}$. It thens gives $\{\mathsf{sk}_i\}_{i \in \mathcal{U}}$ to $\mathcal{A}$. It samples $\tilde{h} \leftarrow_\$ \{0,1\}^\kappa$, set $\mathcal{L} := []$, and then handles $\mathcal{A}$'s oracle queries as follows:

- On $\mathcal{A}$'s call to $\mathcal{O}_{\mathsf{enc\&eval}}()$: Sample $c \leftarrow_\$ \mathbb{G}$, compute $z_i \xleftarrow{\$} \mathsf{Eval}(\mathsf{sk}_i, c)$ for $i \in [n]$, and return $(c, \{z_i\}_{i \in [n] \setminus \mathcal{U}}, \tilde{h})$ to $\mathcal{A}$.
- On $\mathcal{A}$'s call to $\mathcal{O}_{\mathsf{eval}}(i, c)$: Return $\mathsf{Eval}(\mathsf{sk}_i, c)$.
- On $\mathcal{A}$'s call to $\mathsf{Hash}_1(x, y)$:
  a. If $(x, y) \in \mathcal{L}$, let $h := \mathcal{L}[(x, y)]$.
  b. If $(x, y) \notin \mathcal{L}$ and $y \neq \mathsf{Hash}_2(x)^{\mathsf{sk}}$, then sample $h \leftarrow_\$ \{0,1\}^\kappa$ and set $\mathcal{L}[(x, y)] := h$.
  c. If $(x, y) \notin \mathcal{L}$ and $y = \mathsf{Hash}_2(x)^{\mathsf{sk}}$ (i.e., (x, y) is a valid pair), call $\mathcal{O}_{\mathsf{guess}}(x)$. If the output if 1, then $h := \tilde{h}$; otherwise sample $h \leftarrow_\$ \{0,1\}^\kappa$. Set $\mathcal{L}[(x, y)] := h$.

  Return h to $\mathcal{A}$.
- On $\mathcal{A}$'s call to $\mathsf{Hash}_2(x)$: Compute $\mathsf{Hash}_2(x)$ honestly and return the output.

$\mathcal{A}$'s view in the game $\mathsf{Obliviousness}_{\mathsf{TOP},\mathcal{A}}(1^\kappa, n, t)$ is information theoretically indistinguishable from the view simulated by $\mathcal{B}$ in the random oracle model. This can be proved via a hybrid argument:

H_0: The first hybrid is $\mathcal{A}$'s view in the real-world game $\mathsf{Obliviousness}_{\mathsf{TOP},\mathcal{A}}(1^\kappa, n, t)$.

H_1: This hybrid is the same as H_0 except that in the response to $\mathcal{O}_{\mathsf{enc\&eval}}()$, c is randomly sampled as $c \leftarrow_\$ \mathbb{G}$. This hybrid is information theoretically indistinguishable from H_0 because $\mathbb{G}$ is a cyclic group of prime order.

H_2: This hybrid is the same as H_1 except that the output of $\mathsf{Hash}_1(\cdot)$ is a truly random string. The indistinguishability of H_1 and H_2 follows from the random oracle model.

H_3: This hybrid is $\mathcal{A}$'s view simulated by $\mathcal{B}$. It is the same as H_2 except that $\tilde{x}$ is not sampled in the game, but sampled in the guessing game $\mathsf{Guessing}_{\mathcal{B}}(1^\kappa)$. Since $\tilde{x}$ is randomly sampled as $\tilde{x} \leftarrow_\$ \mathcal{X}$ in both hybrids, they are indistinguishable.

Therefore, if $\mathcal{A}$ breaks the game $\mathsf{Obliviousness}_{\mathsf{TOP},\mathcal{A}}(1^\kappa, n, t)$, then $\mathcal{B}$ breaks the guessing game:

$$\begin{aligned}\Pr[\mathsf{Guessing}_{\mathcal{B}}(1^\kappa) = 1] &\geq \frac{\mathsf{MAX}_{|\mathcal{U}|,t}(q_1, \ldots, q_n) + 1}{|\mathcal{X}|} + \mathsf{non\text{-}negl}(\kappa) \\ &\geq \frac{k+1}{|\mathcal{X}|} + \mathsf{non\text{-}negl}(\kappa).\end{aligned}$$

This is information theoretically impossible, leading to a contradiction, and hence concludes the proof.

5.4 Password-Based Threshold Authentication

In a password-based threshold authentication (PbTA) system, there are n servers and any number of clients. PbTA is naturally split into four phases: (i) during a global set-up phase, a master secret key is shared among the servers, which they later use to generate authentication tokens, (ii) in the registration phase, a client C computes sign-up messages (one for each server) based on its username and password and sends them to the servers. Each server processes the message it receives and stores a unique record for that client. (iii) in the sign-on phase, a client initiates authentication by sending a request message that incorporates its username/password and additional information to be included in the token. Each server computes a response using its record for the client. This response contains shares of the authentication token the client eventually wants to obtain. If client's password is a match he is able to combine and finalize the token shares into a single valid token for future accesses. (iv) The finalized token can be verified using a verification algorithm that takes a public or private (depending on the token type) verification key to validate that the token was generated using the unique master secret key. The verification process can also be distributed among multiple servers (may be required for MAC-based tokens) but for simplicity we use a centralized verification phase.

We also note that in a PbTA scheme, clients need not store any persistent secret information. The only secret they need to sign-on is their password which would not be stored anywhere. The device(s) a client uses to sign-on can store certain public parameters of the system (e.g. the identities of the servers).

For simplicity, we assume that clients choose passwords uniformly at random from a space P. Our definitions can be extended to the general case.

Below we define the syntax and security of PbTA in Sections 5.4.1 and 5.4.2, respectively. Next we present our PASTA framework in Section 5.4.3 and prove its security in Section 5.4.4.

5.4.1 Syntax

Definition 5.4.1 (Password-based Threshold Authentication)**.** *A PbTA scheme* Π *is a tuple of seven PPT algorithms (*$\mathsf{GlobalSetup}$*,* SignUp*,* Store*,* $\mathsf{Request}$*,* $\mathsf{Respond}$*,* $\mathsf{Finalize}$*,* Verify*) that satisfies the correctness requirement below.*

- $\mathsf{GlobalSetup}(1^\kappa, n, t, \mathsf{P}) \to (\llbracket \mathsf{sk} \rrbracket, \mathsf{vk}, \mathsf{pp})$. *It takes the security parameter, number of servers* n*, a threshold* t *and the space of passwords* P *as inputs. It outputs a secret key* sk*, shares* $\mathsf{sk}_1, \mathsf{sk}_2, \ldots, \mathsf{sk}_n$ *of the key, and a verification key* vk*. The public parameters* pp *include all the inputs to* $\mathsf{GlobalSetup}$ *and some other information if needed.*

  pp *will be an implicit input in the algorithms below. The* n *servers will be denoted by* $S_1, \ldots, S_n$*.* S_i *receives* $(\mathsf{sk}_i, \mathsf{pp})$ *and initializes a set of records* $\text{REC}_i := \emptyset$*, for* $i \in [n]$*.*

Registration Phase.

- $\mathsf{SignUp}(C, \mathsf{pwd}) \to \{(C, \mathsf{msg}_i)\}_{i \in [n]}$. *It takes as inputs a client id* C *and a password* $\mathsf{pwd} \in \mathsf{P}$*, and outputs a message for each server.*

- $\mathsf{Store}(C, \mathsf{msg}_i) =: R_{i,C}$. *It takes as input a client id* C *and a message* msg_i*, and outputs a record* $R_{i,C}$*.* S_i *stores* $(C, R_{i,C})$ *in its list of records* REC_i *if no record for* C *exists; otherwise, it does nothing.*

Sign-On Phase.

- $\mathsf{Request}(C, \mathsf{pwd}, x, \mathcal{T}) \to (\hat{\mathsf{D}}, \{(C, x, \mathsf{req}_i)\}_{i \in \mathcal{T}})$. *It takes as inputs a client id* C*, a password* pwd*, a value* x*, and a set* $\mathcal{T} \subseteq [n]$*, and outputs a secret state* $\hat{\mathsf{D}}$ *and request messages* $\{\mathsf{req}_i\}_{i \in \mathcal{T}}$*. For* $i \in \mathcal{T}$*,* (C, x, req_i) *is sent to* S_i*.*

- $\mathsf{Respond}(\mathsf{sk}_i, \text{REC}_i, C, x, \mathsf{req}_i) \to \mathsf{res}_i$. *It takes as inputs the secret key share* sk_i*, the record set* REC_i*, a client id* C*, a value* x *and a request message* req_i*, and outputs a response message* res_i*.*

- $\mathsf{Finalize}(\hat{\mathsf{D}}, \{\mathsf{res}_i\}_{i \in \mathcal{T}}) =: \mathsf{tk}$. *It takes as input a secret state* $\hat{\mathsf{D}}$ *and response messages* $\{\mathsf{res}_i\}_{i \in \mathcal{T}}$*, and outputs a token* tk*.*

Verification.

- $\mathsf{Verify}(\mathsf{vk}, C, x, \mathsf{tk}) \to \{0,1\}$. *It takes as inputs the verification key* vk*, a client id* C*, a value* x *and a token* tk*, and outputs* 1 *(denotes validity) or* 0.

Correctness. *For all* $\kappa \in \mathbb{N}$*, any* $n, t \in \mathbb{N}$ *such that* $t \leq n$*, any password space* P*, all* $([\![\mathsf{sk}]\!], \mathsf{vk}, \mathsf{pp})$ *generated by* $\mathsf{Setup}(1^\kappa, n, t, \mathsf{P})$*, any client id* C*, any password* $\mathsf{pwd} \in \mathsf{P}$*, any value* x*, and any* $\mathcal{T} \subseteq [n]$ *of size at least* t*, if*

- $((C, \mathsf{msg}_1), \ldots, (C, \mathsf{msg}_n)) \xleftarrow{\$} \mathsf{SignUp}(C, \mathsf{pwd})$,
- $R_{i,C} := \mathsf{Store}(C, \mathsf{msg}_i)$ *for* $i \in [n]$,
- $(\hat{\mathsf{D}}, \{(C, x, \mathsf{req}_i)\}_{i \in \mathcal{T}}) \xleftarrow{\$} \mathsf{Request}(C, \mathsf{pwd}, x, \mathcal{T})$,
- $\mathsf{res}_i \xleftarrow{\$} \mathsf{Respond}(\mathsf{sk}_i, \mathrm{REC}_i, C, x, \mathsf{req}_i)$ *for* $i \in \mathcal{T}$*, and*
- $\mathsf{tk} := \mathsf{Finalize}(\hat{\mathsf{D}}, \{\mathsf{res}_i\}_{i \in \mathcal{T}})$,

then $\mathsf{Verify}(\mathsf{vk}, C, x, \mathsf{tk}) = 1$.

5.4.2 Security Properties

We define security properties for PbTA with the help of a security game, described in Figure 5.10 in detail. In the security game, an adversary $\mathcal{A}$ gets access to a number of oracles, which run PbTA algorithms and do some bookkeeping. [2]

We do not allow the adversary to interfere with the registration phase. We assume that registration happens over secure channels. In practice, a client would establish a TLS connection with the servers over which it will send the sign-up messages. (Thus, the *actual* number of rounds in registration could be several.) The sign-on phase, however, is completely under the control of the adversary. Adversary can insert, delete or modify messages sent between clients and servers, even if client/server is not corrupt. This is captured by providing $\mathcal{A}$ access to three oracles for the three different algorithms of the sign-on phase (as opposed to just one oracle for registration). $\mathcal{A}$ can give any input to these oracles.

At the start of the game, $\mathsf{GlobalSetup}$ is run to generate shares of the master secret, verification key, public parameters and decryption keys. Public parameters are given to $\mathcal{A}$. It outputs the set of servers $\mathcal{U}$ it wants to corrupt and the client $C^\star$ it wants to target.

A number of variables are initialized before $\mathcal{A}$ is given access to the oracles. $\mathcal{V}$ keeps track of clients as they are corrupted in the game, through $\mathcal{O}_{\mathsf{corrupt}}$ oracle. $\mathsf{PwdList}$ stores clients' passwords in the form of $(id, password)$ pairs, indexed by id, as they sign-up. $\mathsf{ReqList}_{C,i}$

[2] During a run of an oracle, if an algorithm does not produce a valid output, then the oracle stops immediately and returns $\perp$. We do not make this explicit in Figure 5.10 for simplicity.

$\underline{\mathsf{SecGame}_{\Pi,\mathcal{A}}(1^\kappa, n, t, \mathsf{P})}$:

- $([\![\mathsf{sk}]\!], \mathsf{vk}, \mathsf{pp}, (\mathsf{SK}_1, \ldots, \mathsf{SK}_n)) \xleftarrow{\$} \mathsf{GlobalSetup}(1^\kappa, n, t, \mathsf{P})$
- $(\mathcal{U}, C^\star, \mathsf{st}_{\mathsf{adv}}) \xleftarrow{\$} \mathcal{A}(\mathsf{pp})$ *# $\mathcal{U}$: corrupt servers, $C^\star$: targeted client*
- $\mathcal{V} := \emptyset$ *# set of corrupt clients*
- $\mathsf{PwdList} := \emptyset$ *# list of (C, pwd) pairs, indexed by C*
- $\mathsf{ReqList}_{C,i} := \emptyset$ for $i \in [n]$ *# token requests C makes to S_i*
- $\mathsf{ct} := 0$, $\mathsf{LiveSessions} = []$ *# LiveSessions is indexed by ct*
- $\mathsf{TokList} := \emptyset$ *# list of tokens generated through $\mathcal{O}_{\mathsf{final}}$*
- $Q_{C,i} := 0$ for all C and $i \in [n]$
- $Q_{C,x} := 0$ for all C and x
- $\mathsf{out} \xleftarrow{\$} \mathcal{A}^{\langle\mathcal{O}\rangle}(\{\mathsf{sk}_i\}_{i\in\mathcal{U}}, \{\mathsf{SK}_i\}_{i\in\mathcal{U}}, \mathsf{st}_{\mathsf{adv}})$

$\underline{\mathcal{O}_{\mathsf{corrupt}}(C)}$.

- $\mathcal{V} := \mathcal{V} \cup \{C\}$
- if $(C, \star) \in \mathsf{PwdList}$, return $\mathsf{PwdList}[C]$

$\underline{\mathcal{O}_{\mathsf{register}}(C)}$.

- require: $\mathsf{PwdList}[C] = \bot$
- $\mathsf{pwd} \leftarrow_{\$} \mathsf{P}$
- add (C, pwd) to $\mathsf{PwdList}$
- $((C, \mathsf{msg}_1), \ldots, (C, \mathsf{msg}_n)) \xleftarrow{\$} \mathsf{SignUp}(C, \mathsf{pwd})$
- $R_{i,C} := \mathsf{Store}(C, \mathsf{msg}_i)$ for all $i \in [n]$
- add $R_{i,C}$ to REC_i for all $i \in [n]$

$\underline{\mathcal{O}_{\mathsf{req}}(C, x, \mathcal{T})}$.

- require: $\mathsf{PwdList}[C] \neq \bot$
- $(\hat{\mathsf{D}}, \{\mathsf{req}_i\}_{i\in\mathcal{T}}) \xleftarrow{\$} \mathsf{Request}(C, \mathsf{PwdList}[C])$
- $\mathsf{LiveSessions}[\mathsf{ct}] := \hat{\mathsf{D}}$
- add req_i to $\mathsf{ReqList}_{C,i}$ for $i \in \mathcal{T}$
- increment ct by 1
- return $\{\mathsf{req}_i\}_{i\in\mathcal{T}}$

$\underline{\mathcal{O}_{\mathsf{resp}}(i, C, x, \mathsf{req}_i)}$.

- $\mathsf{res}_i \xleftarrow{\$} \mathsf{Respond}(\mathsf{sk}_i, \mathrm{REC}_i, C, x, \mathsf{req}_i)$
- if $\mathsf{req}_i \notin \mathsf{ReqList}_{C,i}$, increment $Q_{C,i}$ by 1
- increment $Q_{C,x}$ by 1
- return res_i

$\underline{\mathcal{O}_{\mathsf{final}}(\mathsf{ct}, \{\mathsf{res}_i\}_{i\in S})}$.

- $\hat{\mathsf{D}} := \mathsf{LiveSessions}[\mathsf{ct}]$
- $\mathsf{tk} := \mathsf{Finalize}(\hat{\mathsf{D}}, \{\mathsf{res}_i\}_{i\in S})$
- add tk to TokList
- return tk

$\underline{\mathcal{O}_{\mathsf{verify}}(C, x, \mathsf{tk})}$.

- return $\mathsf{Verify}(\mathsf{vk}, C, x, \mathsf{tk})$

Figure 5.10: Security game for PbTA.

stores the requests generated by C for the i-th server. These requests will not be counted against the adversary, as we will see later.

$\mathcal{O}_{\mathsf{req}}(C, x, \mathcal{T})$ allows $\mathcal{A}$ to start a sign-on session of C—who may not be corrupt—with servers in $\mathcal{T}$ to generate a token on x. $\mathcal{O}_{\mathsf{req}}$ runs $\mathsf{Request}$ to generate request messages, using the password of C stored in $\mathsf{PwdList}$. While these messages are revealed to $\mathcal{A}$, C's intermediate state $\hat{\mathsf{D}}$ is stored in $\mathsf{LiveSessions}$ at position ct. $\mathcal{A}$ can resume this sign-on session at any point in the future by invoking $\mathcal{O}_{\mathsf{final}}$ with ct and any arbitrary responses from the servers in $\mathcal{T}$.

$\mathcal{O}_{\mathsf{resp}}$ can be invoked to get responses from a server as part of the sign-on phase. $\mathcal{A}$ can invoke $\mathcal{O}_{\mathsf{resp}}$ with any message (C, x, req_i) of its choice. $\mathcal{O}_{\mathsf{server}}$ does not check if req_i was indeed generated by C or not; a response is generated anyway, and returned to the adversary. However, if the request req_i was not generated by C before, then this could give some advantage to $\mathcal{A}$ in attacking C; so we increment a counter $Q_{C,i}$ in this case. A different counter $Q_{C,x}$ is incremented even if req_i was generated by C. This is just to count the number of times different servers are invoked on C and x. If this number is less than even $t - |\mathcal{U}|$, then $\mathcal{A}$ should not be able to generate a token on (C, x) except with negligible probability (see Def. 5.4.3, first point).

Note that the counters Q are separate for each client. If $\mathcal{O}_{\mathsf{server}}$ is invoked with a certain client id, then the counters for just that id are updated. When we define the security properties for PbTA below, only the counters for $C^\star$ (the target client) are taken into account. Thus, we consider $\mathcal{A}$ to be attacking $C^\star$ only when it reveals this id to the servers. In other words, we do not allow $\mathcal{A}$ to gain any advantage in attacking $C^\star$ if it pretends to be someone else.

$\mathcal{O}_{\mathsf{final}}$, as mentioned before, can be used to resume a sign-on session. Client's state $\hat{\mathsf{D}}$ is retrieved from $\mathsf{LiveSessions}$, and $\mathsf{Finalize}$ is run on $\hat{\mathsf{D}}$ and the server responses given as input. $\mathcal{A}$ can provide any arbitrary response on behalf of any server—even the ones that are not corrupt. The token generated through $\mathsf{Finalize}$ is given to $\mathcal{A}$ and added to $\mathsf{TokList}$. Finally, $\mathcal{A}$ can use $\mathcal{O}_{\mathsf{verify}}$ to check if a token is valid or not.

We are now ready to formally state the two security properties we would like any PbTA scheme to satisfy.

Definition 5.4.2 (Password Safety)**.** *A PbTA scheme* Π *is* password safe *if for all* $n, t \in \mathbb{N}$*,* $t \leq n$*, all password space* P *and all PPT adversary* $\mathcal{A}$ *in* $\mathsf{SecGame}_{\Pi,\mathcal{A}}(1^\kappa, n, t, \mathsf{P})$ *(Figure 5.10), there exists a negligible function* negl *s.t.*

$$\Pr[C^\star \notin \mathcal{V} \wedge \mathsf{out} = \mathsf{PwdList}[C^\star] \neq \perp] \leq \frac{\mathsf{MAX}_{|\mathcal{U}|,t}(Q_{C^\star,1}, \ldots, Q_{C^\star,n}) + 1}{|\mathsf{P}|} + \mathsf{negl}(\kappa). \quad (5.5)$$

To get some intuition into the above definition, consider the following attack. $\mathcal{A}$ guesses a password for $C^\star$, generates request messages on its own (so that it knows the intermediate state), invokes $\mathcal{O}_{\mathsf{resp}}$ to get the corresponding responses, combines them using $\mathsf{Finalize}$ to get a token, and finally checks if the token is valid or not. If the password guess was correct, then the token would be valid by the correctness property of PbTA.

As such, $\mathcal{A}$ is not restricted to attacking a PbTA scheme in the above manner. However, we require that, essentially, this is the best it can do. $\mathsf{MAX}_{|\mathcal{U}|,t}(Q_{C^\star,1},\ldots,Q_{C^\star,n})$ in Eq. 5.5 gives a bound on the number of password attempts $\mathcal{A}$ can make through the above attack.

We do not penalize $\mathcal{A}$ for just replaying the requests generated by C itself by not incrementing $Q_{C,i}$ in those cases. The additive factor of 1 captures the possibility that $\mathcal{A}$ can output a new guess at the end of the game (similar to the obliviousness property for TOPRF, see Def. 5.3.3).

Definition 5.4.3 (Unforgeability)**.** *A PbTA scheme* Π *is* unforgeable *if for all* $n, t \in \mathbb{N}$, $t \leq n$, *all password space* P *and all PPT adversary* $\mathcal{A}$ *in* $\mathsf{SecGame}_{\Pi,\mathcal{A}}(1^\kappa, n, t, \mathsf{P})$ *(Figure 5.10), there exists a negligible function* negl *s.t.*

- *if* $Q_{C^\star,x^\star} < t - |\mathcal{U}|$,

$$\Pr[\mathsf{Verify}(\mathsf{vk}, C^\star, x^\star, \mathsf{tk}^\star) = 1] \leq \mathsf{negl}(\kappa); \tag{5.6}$$

- *else*

$$\Pr[C^\star \notin \mathcal{V} \wedge \mathsf{tk}^\star \notin \mathsf{TokList} \wedge \mathsf{Verify}(\mathsf{vk}, C^\star, x^\star, \mathsf{tk}^\star) = 1] \leq \frac{\mathsf{MAX}_{|\mathcal{U}|,t}(Q_{C^\star,1},\ldots,Q_{C^\star,n})}{|\mathsf{P}|} + \mathsf{negl}(\kappa), \tag{5.7}$$

where $\mathcal{A}$*'s output* out *is parsed as* $(x^\star, \mathsf{tk}^\star)$.

The security game for unforgeability is the same as password-safety (Figure 5.10) but $\mathcal{A}$ produces a token $\mathsf{tk}^\star$ now. The probability of it being valid on $(C^\star, x^\star)$ depends on several cases. First, if the value of $Q_{C^\star,x^\star}$ is smaller than even $t - |\mathcal{U}|$, then $\mathcal{A}$ didn't even contact *enough* servers on $(C^\star, x^\star)$. So we would like its probability of producing a valid token to be negligible. (Eq. 5.6 also captures that querying servers on (C, x) for a different C or x than $C^\star$ and $x^\star$ should not help.)

If $\mathcal{A}$ does contact enough servers and $C^\star$ was corrupted, then $\mathcal{A}$ can easily generate a valid token; so this case is not interesting. However, if $C^\star$ is not corrupt but $\mathcal{A}$ is able to guess its password, then it can also produce a valid token (with respect to $C^\star$ only). Comparing Eq. 5.7 and 5.5, one can see that unforgeability property basically requires that this is the best $\mathcal{A}$ can do.

5.4.3 PASTA: Our Construction

In this section we present PASTA, our framework for building PbTA schemes. PASTA provides a way to combine any threshold token generation scheme (TTG) and any threshold oblivious PRF (TOP) in a black-box way to build a PbTA scheme that provides strong password-safety and unforgeability guarantees. Figure 5.11 provides a complete description of the framework.

PASTA uses the two main underlying primitives, TTG and TOP, in a fairly light-weight manner. The sign-on phase, which consists of Request, Respond and Finalize, does not add any public-key operations on top of what the primitives may have. Request runs TOP.Encode once; Respond runs both TOP.Eval and TTG.PartEval, but only once each; and, Finalize runs TOP.Combine and TTG.Combine once each. Even though number of decryptions in Finalize is proportional to t, these operations are very fast symmetric-key operations. Thus, PASTA makes minimal use of the two primitives that it builds on and its performance is mainly governed by the efficiency of these primitives.

PASTA needs a *key-binding* symmetric-key encryption scheme so that when a ciphertext is decrypted with a wrong key, decryption fails [Fis99]. Key-binding can be obtained very efficiently in the random oracle model, for e.g., by appending a hash of the secret key to every ciphertext.

For the sign-on phase, PASTA assumes that the servers communicate to clients over authenticated channels so that an adversary cannot send arbitrary messages to a client on behalf of honest servers. PASTA does *not* assume that these channels provide any confidentiality. Observe that if there is an authenticated channel in the other direction, namely the servers can identity the sender of every message they receive, then passwords are not needed, and hence a PbTA scheme is moot.

An important feature of PASTA, especially from the point of view of proving security, is that the use of TOP and TTG overlaps very slightly. The output of TOP is used to encrypt the partial evaluations of TTG but, apart from that, they operate independently. Thus, even if TTG is broken in some manner, it would not affect the safety of clients' passwords. Furthermore, even if TOP is broken, a threshold number of servers would still be needed to generate a token. However, PASTA must prevent against several other attack scenarios, as captured by the game in Figure 5.10. The formal security guarantee of PASTA is stated as follows.

Theorem 5.4.4 (Security of PASTA)**.** *If* TTG *is an unforgeable threshold token generation scheme (Def. 5.2.3),* TOP *is an unpredictable (Def. 5.3.2) and oblivious (Def. 5.3.3) TOPRF, and* SKE *is a key-binding CPA-secure symmetric-key encryption scheme, then the PbTA scheme PASTA as described in Figure 5.11 is password-safe (Def. 5.4.2) and unforgeable (Def. 5.4.3) when* Hash *is modeled as a random oracle.*

5.4.4 Security Proof

Password-safety and unforgeability properties are proved in Sections 5.4.4.1 and 5.4.4.2 respectively.

5.4.4.1 Password Safety

PASTA's password safety primarily relies on the obliviousness of TOP. Intuitively, if we use a TOPRF on clients' passwords, then obliviousness of TOPRF would make it hard to guess

Ingredients:

- A threshold token generation scheme $\mathsf{TTG} := (\mathsf{TTG.Setup}, \mathsf{TTG.PartEval}, \mathsf{TTG.Combine}, \mathsf{TTG.Verify})$.
- A threshold oblivious PRF $\mathsf{TOP} := (\mathsf{TOP.Setup}, \mathsf{TOP.Encode}, \mathsf{TOP.Eval}, \mathsf{TOP.Combine})$.
- A symmetric-key encryption scheme $\mathsf{SKE} := (\mathsf{SKE.Encrypt}, \mathsf{SKE.Decrypt})$.
- A hash function Hash.

$\underline{\mathsf{GlobalSetup}(1^\kappa, n, t, \mathsf{P}) \to (\llbracket \mathsf{sk} \rrbracket, \mathsf{vk}, \mathsf{pp})}$.

- Run $\mathsf{TTG.Setup}(1^\kappa, n, t)$ to get $(\llbracket \mathsf{tsk} \rrbracket, \mathsf{tvk}, \mathsf{tpp})$.
- Set $\mathsf{sk}_i := \mathsf{tsk}_i$ for all $i \in [n]$, $\mathsf{vk} := \mathsf{tvk}$ and $\mathsf{pp} := (\kappa,\ n,\ t,\ \mathsf{P},\ \mathsf{tpp})$.

$\underline{\mathsf{SignUp}(C, \mathsf{pwd}) \to ((C, \mathsf{msg}_1), \ldots, (C, \mathsf{msg}_n))}$.

- Run $\mathsf{TOP.Setup}(1^\kappa, n, t)$ to get $(\llbracket k \rrbracket, \mathsf{opp})$.
- Compute $h := \mathsf{TOP}(k, \mathsf{pwd})$ and $h_i = \mathsf{Hash}(h \| i)$ for $i \in [n]$.
- Set $\mathsf{msg}_i := (k_i, h_i)$ for $i \in [n]$.

$\underline{\mathsf{Store}(\mathsf{SK}_i, C, \mathsf{msg}_i) =: R_{i,C}}$.

- Parse msg_i as (k_i, h_i).
- Set $R_{i,C} := (k_i, h_i)$

$\underline{\mathsf{Request}(C, \mathsf{pwd}, x, \mathcal{T}) \to (\{(C, x, \mathsf{req}_i)\}_{i \in \mathcal{T}}, \hat{\mathsf{D}})}$.

- If $|\mathcal{T}| < t$, output $\perp$.
- Pick a ρ at random. Run $\mathsf{TOP.Encode}(\mathsf{pwd}, \rho)$ to get c.
- Set $\mathsf{req}_i := c$ for all $i \in [n]$ and $\hat{\mathsf{D}} := (C, \mathsf{pwd}, \rho, \mathcal{T})$.

$\underline{\mathsf{Respond}(\mathsf{sk}_i, \mathrm{REC}_i, C, x, \mathsf{req}_i) \to \mathsf{res}_i}$.

- If $R_{i,C} \notin \mathrm{REC}_i$, output $\perp$. Else, parse $R_{i,C}$ as (k_i, h_i).
- Run $\mathsf{TOP.Eval}(k_i, \mathsf{req}_i)$ to get z_i.
- Run $\mathsf{TTG.PartEval}(\mathsf{tsk}_i, C \| x)$ to get y_i.
- Set $\mathsf{res}_i := (z_i, \mathsf{SKE.Encrypt}(h_i, y_i))$.

$\underline{\mathsf{Finalize}(\hat{\mathsf{D}}, \{\mathsf{res}_i\}_{i \in S}) \to \mathsf{tk}}$.

- Parse res_i as (z_i, ctxt_i) and $\hat{\mathsf{D}}$ as $(C, \mathsf{pwd}, \rho, \mathcal{T})$.
- If $S \neq \mathcal{T}$, output $\perp$.
- Run $\mathsf{TOP.Combine}(\mathsf{pwd}, \{(i, z_i)\}_{i \in \mathcal{T}}, \rho)$ to get h.
- For all $i \in \mathcal{T}$, compute $h_i := \mathsf{Hash}(h \| i)$ and $y_i := \mathsf{SKE.Decrypt}(h_i, \mathsf{ctxt}_i)$.
- Finally, set tk to be $\mathsf{TTG.Combine}(\{i, y_i\}_{i \in \mathcal{T}})$.

(If any of $\mathsf{TOP.Combine}$, $\mathsf{SKE.Decrypt}$ or $\mathsf{TTG.Combine}$ fail, $\perp$ is output.)

$\underline{\mathsf{Verify}(\mathsf{vk}, C, x, \mathsf{tk}) \to \{0, 1\}}$.

- Output $\mathsf{TTG.Verify}(\mathsf{tvk}, C \| x, \mathsf{tk})$.

Figure 5.11: A complete description of PASTA.

them. Formally, we build an adversary $\mathcal{B}$ that can translate an adversary $\mathcal{A}$'s advantage in the password-safety game into a similar advantage in the TOPRF obliviousness game Obliviousness (Figure 5.7). $\mathcal{B}$ will run $\mathcal{A}$ *internally* simulating the password-safety for it, while playing the role of adversary *externally* in Obliviousness.

$\mathcal{B}$ can implicitly set the targeted client $C^\star$'s password to be the random value chosen in Obliviousness. If $\mathcal{A}$ guesses the password, $\mathcal{B}$ can output the same guess. However, to simulate SecGame properly for $\mathcal{A}$, $\mathcal{B}$ needs to run the oracles in a way that $\mathcal{A}$ cannot tell the difference. In particular, $\mathcal{B}$ needs partial TOPRF evaluations z_i on the password for $\mathcal{O}_{\mathsf{resp}}$, the final TOPRF value for $\mathcal{O}_{\mathsf{register}}$ and the randomness ρ used for encoding for $\mathcal{O}_{\mathsf{final}}$. $\mathcal{B}$ can take help of the oracles $\mathcal{O}_{\mathsf{eval}}$ and $\mathcal{O}_{\mathsf{enc\&eval}}$ provided by Obliviousness to handle the first two problems, but there is no way to get ρ in Obliviousness.

Intermediate Hybrid. We tackle the latter problem first by going through a hybrid. We refer to the original game as H_0 and the new game as H_1. H_0 is described in Figure 5.12; it basically replaces Π in Figure 5.10 with PASTA. H_1 is described in Figure 5.13. In H_1, several oracles behave differently for the targeted client $C^\star$. $\mathcal{O}_{\mathsf{req}}$ evaluates the TOPRF in advance for $C^\star$. It stores the partial evaluations z_i and the final result h in LiveSessions itself. Importantly, it does *not* store ρ. When $\mathcal{O}_{\mathsf{resp}}$ is invoked, it checks if $C^\star$ generated req_i for S_i before ($\mathsf{req}_i \in \mathsf{ReqList}_{C,i}$). If yes, then z_i is picked up from LiveSessions. Now, whether a z_i computed in advance is used in $\mathcal{O}_{\mathsf{resp}}$ or not makes no difference from the point of the adversary because z_i is derived deterministically from k_i and req_i.

Oracle $\mathcal{O}_{\mathsf{final}}$ also behaves differently for $C^\star$. First, note that if $\mathsf{TOP.PubCombine}(\{z_i\}_{i\in\mathcal{T}})$ is equal to $\mathsf{TOP.PubCombine}(\{z'_i\}_{i\in\mathcal{T}})$, then combining either set will lead to the same value. The only difference in H_1 is that h was computed beforehand. Once again, for the same reason as above, this makes no difference.

The crucial step where H_0 and H_1 differ is when the two outputs of PubCombine do not match. While H_0 does not do any test of this kind, H_1 simply outputs $\perp$. For these hybrids to be indistinguishable, we need to argue that *had* the outputs of PubCombine not matched in H_0, it would have output $\perp$ as well (at least with a high probability).

Note that the *right* z_i and h are well-defined for H_0; they can be derived from pwd and ρ. If one were to do the public combine test in this hybrid and it fails, then $h' \neq h$ with high probability. Therefore, using the collision resistance of Hash, one can argue that $h'_i \neq h_i$. Now, observe that there must be an honest S_j in $\mathcal{T}$, so ctxt'_j could only have been generated by S_j (recall our authenticated channels assumption). When ciphertext ctxt'_j, which was encrypted under h_j, is decrypted with $h'_j \neq h_j$, decryption fails with high probability due to the key-binding property of SKE. Thus, H_0 returns $\perp$ just like H_1.

Reduction. Now that we know that absence of encoding randomness ρ would not prevent a successful simulation of $\mathcal{O}_{\mathsf{final}}$, we come back to the task of exploiting TOPRF obliviousness to hide the targeted client's password. Towards this, adversary $\mathcal{B}$ is formally described in Figure 5.14. When $\mathcal{B}$ outputs a message, it should be interpreted as sending the message to

the obliviousness game. Let's now go through the differences between H_1 and $\mathcal{B}$'s simulation of it one by one.

Simulation of $\mathcal{O}_{\mathsf{register}}$ differs only for $C = C^\star$. In H_1, a randomly chosen password for $C^\star$ is used to compute h, while in $\mathcal{B}$'s simulation, $C^\star$'s password is implicitly set to be the random input $\tilde{x}$ chosen by $\mathsf{Obliviousness}$ and $\mathcal{O}_{\mathsf{enc\&eval}}$ is called to get h. Clearly, this difference does not affect $\mathcal{A}$. There is one other difference though: while all of $k_1, \ldots, k_n$ are known in H_1, $\mathcal{B}$ knows k_i for corrupt servers only. As a result, $\mathcal{B}$ defines $R_{i,C^\star}$ to be $(\mathbf{0}, h_i)$ for $i \in [n] \setminus \mathcal{U}$.

Like the registration oracle, request oracle behaves differently only when $C = C^\star$. However, one can easily see that the difference is insignificant: while H_1 computes c, z_i and h using $\mathsf{PwdList}[C^\star]$, $\mathcal{B}$ invokes $\mathcal{O}_{\mathsf{enc\&eval}}$ to get them, which uses $\tilde{x}$.

Finally, $\mathcal{B}$ invokes $\mathcal{O}_{\mathsf{eval}}$ to get z_i in the simulation of $\mathcal{O}_{\mathsf{resp}}$ (because it does not know k_i for honest servers) but it is computed directly in H_1. This does not make any difference either. The important thing to note is that the counter $Q_{C^\star,i}$ is incremented if and only if the counter q_i of $\mathsf{Obliviousness}$ is incremented. As a result, the final value of $Q_{C^\star,i}$ will be the same as q_i. Therefore, $\mathcal{B}$ will successfully translate $\mathcal{A}$'s probability of guessing $C^\star$'s password to guessing $\tilde{x}$.

5.4.4.2 Unforgeability

First we handle the easier case of $Q_{C^\star,x^\star} < t - |\mathcal{U}|$. Here $C^\star$ could even be corrupt, so $\mathcal{A}$ may know its password. Note that $Q_{C^\star,x^\star}$ is incremented on every invocation of $\mathcal{O}_{\mathsf{resp}}(i, C^\star, x^\star, \mathsf{req}_i)$ irrespective of the value of i and whether or not $\mathsf{req}_i \in \mathsf{ReqList}_{i,C^\star}$. So if $Q_{C^\star,x^\star} < t - |\mathcal{U}|$, $\mathcal{A}$ simply doesn't have enough shares to generate a valid token, irrespective of whether $C^\star$ is corrupt or not. One can formally prove unforgeability in this case by invoking the unforgeability of the threshold token generation scheme TTG (Definition 5.2.3). We skip the details.

When $Q_{C^\star,x^\star} \geq t - |\mathcal{U}|$, unforgeability can only be expected when $C^\star$ is never corrupted. We need to show that generating a valid token for $(C^\star, x^\star)$ for any $x^\star$ effectively amounts to guessing $C^\star$'s password. Indistinguishability of H_0 (Figure 5.12) and H_1 (Figure 5.13) still holds because it just relies on the properties of $\mathsf{PubCombine}$ and authenticated channels.

We now wish to build an adversary $\mathcal{B}'$ that can use an adversary $\mathcal{A}$ who breaks the unforgeability guarantee of PASTA to break the unpredictability of TOPRF. The first natural question to ask is whether $\mathcal{B}'$ can break unpredictability of TOPRF in the same way as $\mathcal{B}$ broke obliviousness. Not quite, because there are some key differences in the two settings:

- Even though both $\mathcal{B}$ and $\mathcal{B}'$ get access to an oracle $\mathcal{O}_{\mathsf{enc\&eval}}$ that both encodes and evaluates, $\mathcal{B}$'s oracle returns the final TOPRF output h while $\mathcal{B}'$'s oracle doesn't. So it is not clear how h_i will be generated by $\mathcal{O}_{\mathsf{register}}$ and $\mathcal{O}_{\mathsf{final}}$ for $C^\star$.

- $\mathcal{B}$ was able to use the output of $\mathcal{A}$ for the password-safety game directly into the obliviousness game, but $\mathcal{B}'$ cannot. $\mathcal{A}$ now outputs a token for the authentication

$\underline{\mathsf{SecGame}_{\mathrm{PASTA},\mathcal{A}}(1^\kappa, n, t, \mathsf{P})}$:

- $(\llbracket \mathsf{tsk} \rrbracket, \mathsf{tvk}, \mathsf{tpp}) \xleftarrow{\$} \mathsf{TTG.Setup}(1^\kappa, n, t)$
- set $\mathsf{sk}_i := \mathsf{tsk}_i$ for all $i \in [n]$, $\mathsf{vk} := \mathsf{tvk}$ and $\mathsf{pp} := (\kappa,\ n,\ t,\ \mathsf{P},\ \mathsf{tpp})$.
- $(\mathcal{U}, C^\star, \mathsf{st}_{\mathsf{adv}}) \xleftarrow{\$} \mathcal{A}(\mathsf{pp})$
- $\mathcal{V}, \mathsf{PwdList}, \mathsf{TokList} := \emptyset$, $\mathsf{ReqList}_{C,i} := \emptyset$ for $i \in [n]$
- $\mathsf{ct} := 0$, $\mathsf{LiveSessions} = []$
- $Q_{C,i}, Q_{C,x} := 0$ for all C, $i \in [n]$ and x
- $\mathsf{out} \xleftarrow{\$} \mathcal{A}^{\langle \mathcal{O} \rangle}(\{\mathsf{sk}_i\}_{i\in\mathcal{U}}, \{\mathsf{SK}_i\}_{i\in\mathcal{U}}, \mathsf{st}_{\mathsf{adv}})$

$\underline{\mathcal{O}_{\mathsf{corrupt}}(C)}$.

- $\mathcal{V} := \mathcal{V} \cup \{C\}$
- if $(C, \star) \in \mathsf{PwdList}$, return $\mathsf{PwdList}[C]$

$\underline{\mathcal{O}_{\mathsf{register}}(C)}$.

- require: $\mathsf{PwdList}[C] = \perp$
- $\mathsf{pwd} \leftarrow_{\$} \mathsf{P}$
- add (C, pwd) to $\mathsf{PwdList}$
- $(\llbracket k \rrbracket, \mathsf{opp}) \xleftarrow{\$} \mathsf{TOP.Setup}(1^\kappa, n, t)$
- $h := \mathsf{TOP}(k, \mathsf{pwd})$ and $h_i := \mathsf{Hash}(h \| i)$ for $i \in [n]$
- $R_{i,C} := (k_i, h_i)$ for all $i \in [n]$
- add $R_{i,C}$ to REC_i for all $i \in [n]$

$\underline{\mathcal{O}_{\mathsf{req}}(C, x, \mathcal{T})}$.

- if $\mathsf{PwdList}[C] = \perp$ or $|\mathcal{T}| < t$, output $\perp$
- $c := \mathsf{TOP.Encode}(\mathsf{PwdList}[C], \rho)$ for a random ρ
- set $\mathsf{req}_i := c$ for $i \in [n]$
- $\mathsf{LiveSessions}[\mathsf{ct}] := (C, \mathsf{PwdList}[C], \rho, \mathcal{T})$
- add req_i to $\mathsf{ReqList}_{C,i}$ for $i \in \mathcal{T}$
- increment ct by 1
- return $\{\mathsf{req}_i\}_{i\in\mathcal{T}}$

$\underline{\mathcal{O}_{\mathsf{resp}}(i, C, x, \mathsf{req}_i)}$.

- require: $i \in [n] \setminus \mathcal{U}$
- if $R_{i,C} \notin \mathrm{REC}_i$, return $\perp$; else, parse $R_{i,C}$ as (k_i, h_i)
- if $\mathsf{req}_i \notin \mathsf{ReqList}_{C,i}$, increment $Q_{C,i}$ by 1
- $z_i := \mathsf{TOP.Eval}(k_i, \mathsf{req}_i)$
- $y_i \xleftarrow{\$} \mathsf{TTG.PartEval}(\mathsf{tsk}_i, C \| x)$
- set $\mathsf{res}_i := (z_i, \mathsf{SKE.Encrypt}(h_i, y_i))$
- increment $Q_{C,x}$ by 1
- return res_i

$\underline{\mathcal{O}_{\mathsf{final}}(\mathsf{ct}, \{\mathsf{res}_i\}_{i\in S})}$.

- $\hat{\mathsf{D}} := \mathsf{LiveSessions}[\mathsf{ct}]$
- parse res_i as (z'_i, ctxt'_i) and $\hat{\mathsf{D}}$ as $(C, \mathsf{pwd}, \rho, \mathcal{T})$.
- if $S \neq \mathcal{T}$, output $\perp$
- $h' := \mathsf{TOP.Combine}(\mathsf{pwd}, \{(i, z'_i)\}_{i\in\mathcal{T}}, \rho)$
- for $i \in \mathcal{T}$, $h'_i := \mathsf{Hash}(h' \| i)$ and $y'_i := \mathsf{SKE.Decrypt}(h'_i, \mathsf{ctxt}'_i)$
- set $\mathsf{tk} := \mathsf{TTG.Combine}(\{i, y'_i\}_{i\in\mathcal{T}})$
- add tk to $\mathsf{TokList}$
- return tk

$\underline{\mathcal{O}_{\mathsf{verify}}(C, x, \mathsf{tk})}$.

- return $\mathsf{TTG.Verify}(\mathsf{tvk}, C \| x, \mathsf{tk})$

Figure 5.12: $\mathsf{SecGame}$ in hybrid H_0 for PASTA.

$\mathsf{SecGame}_{\mathrm{PASTA},\mathcal{A}}(1^\kappa, n, t, \mathsf{P})$:

Same as H_0*, except the following oracles behave differently when* $C = C^\star$*. Below, we describe their behavior for this case only, highlighting the differences in red. When* $C \neq C^\star$*, they behave in the same way as* H_0*.*

$\mathcal{O}_{\mathsf{req}}(C^\star, x, \mathcal{T})$.

- if $\mathsf{PwdList}[C^\star] = \bot$ or $|\mathcal{T}| < t$, output $\bot$
- $c := \mathsf{TOP.Encode}(\mathsf{PwdList}[C^\star], \rho)$ for a random ρ
- set $\mathsf{req}_i := c$ for $i \in [n]$
- $z_i := \mathsf{TOP.Eval}(k_i, \mathsf{req}_i)$
- $h := \mathsf{TOP.Combine}(\mathsf{PwdList}[C^\star], \{(i, z_i)\}_{i\in\mathcal{T}}, \rho)$
- $\mathsf{LiveSessions}[\mathsf{ct}] := (C^\star, c, \{(i, z_i)\}_{i\in\mathcal{T}}, h)$
- add req_i to $\mathsf{ReqList}_{C^\star,i}$ for $i \in \mathcal{T}$
- increment ct by 1
- return $\{\mathsf{req}_i\}_{i\in\mathcal{T}}$

$\mathcal{O}_{\mathsf{resp}}(i, C^\star, x, \mathsf{req}_i)$.

- require: $i \in [n] \setminus \mathcal{U}$
- if $R_{i,C^\star} \notin \mathrm{REC}_i$, return $\bot$; else, parse $R_{i,C^\star}$ as (k_i, h_i)
- if $\mathsf{req}_i \notin \mathsf{ReqList}_{C^\star,i}$, increment $Q_{C^\star,i}$ by 1
- if $(\mathsf{req}_i \notin \mathsf{ReqList}_{C^\star,i})$:
  - $z_i := \mathsf{TOP.Eval}(k_i, \mathsf{req}_i)$
- else:
  - let z_i be the value associated with i in the entry $(C^\star, \mathsf{req}_i, \ldots, (i, z_i)\ldots)$ in $\mathsf{LiveSessions}$
- $y_i \xleftarrow{\$} \mathsf{TTG.PartEval}(\mathsf{tsk}_i, C^\star \| x)$
- set $\mathsf{res}_i := (z_i, \mathsf{SKE.Encrypt}(h_i, y_i))$
- increment $Q_{C^\star,x}$ by 1
- return res_i

$\mathcal{O}_{\mathsf{final}}(\mathsf{ct}, \{\mathsf{res}_i\}_{i\in S})$.

- $(C^\star, c, \{(i, z_i)\}_{i\in\mathcal{T}}, h) := \mathsf{LiveSessions}[\mathsf{ct}]$
- parse res_i as (z'_i, ctxt'_i)
- if $S \neq \mathcal{T}$, output $\bot$
- if $(\mathsf{TOP.PubCombine}(\{z_i\}_{i\in\mathcal{T}}) \neq \mathsf{TOP.PubCombine}(\{z'_i\}_{i\in\mathcal{T}}))$:
  - return $\bot$
- for $i \in \mathcal{T}$, $h'_i := \mathsf{Hash}(h \| i)$ and $y'_i := \mathsf{SKE.Decrypt}(h'_i, \mathsf{ctxt}'_i)$
- set $\mathsf{tk} := \mathsf{TTG.Combine}(\{i, y'_i\}_{i\in\mathcal{T}})$
- add tk to TokList
- return tk

Figure 5.13: SecGame in hybrid H_1.

$\mathcal{B}_{\mathcal{A}}(1^{\kappa}, n, t, \mathsf{P})$:
Same as H_1*, except the following oracles are simulated differently when* $C = C^{\star}$*. Below, these oracles are described for this case only, with the differences highlighted in red. Output whatever* $\mathcal{A}$ *does.*

$\mathcal{O}_{\mathsf{register}}(C^{\star})$.

- require: $\mathsf{PwdList}[C^{\star}] = \perp$
- add $(C^{\star}, \mathsf{unknown})$ to PwdList
- output $\mathcal{U}$, get back $\{k_i\}_{i \in \mathcal{U}}$
- query $\mathcal{O}_{\mathsf{enc\&eval}}$ to get $(c, \{z_i\}_{i \in [n]}, h)$
- $h_i := \mathsf{Hash}(h \| i)$ for $i \in [n]$
- for $i \in [n] \setminus \mathcal{U}$, $R_{i,C^{\star}} := (\mathbf{0}, h_i)$
- for $i \in \mathcal{U}$, $R_{i,C^{\star}} := (k_i, h_i)$
- add $R_{i,C^{\star}}$ to REC_i for all $i \in [n]$

$\mathcal{O}_{\mathsf{req}}(C^{\star}, x, \mathcal{T})$.

- if $\mathsf{PwdList}[C^{\star}] = \perp$ or $|\mathcal{T}| < t$, output $\perp$
- query $\mathcal{O}_{\mathsf{enc\&eval}}$ to get $(c, \{z_i\}_{i \in [n]}, h)$
- set $\mathsf{req}_i := c$ for $i \in [n]$
- $\mathsf{LiveSessions}[\mathsf{ct}] := (C, c, \{i, z_i\}_{i \in \mathcal{T}}, h)$
- add req_i to $\mathsf{ReqList}_{C^{\star},i}$ for $i \in \mathcal{T}$
- increment ct by 1
- return $\{\mathsf{req}_i\}_{i \in \mathcal{T}}$

$\mathcal{O}_{\mathsf{resp}}(i, C^{\star}, x, \mathsf{req}_i)$.

- require: $i \in [n] \setminus \mathcal{U}$
- if $R_{i,C^{\star}} \notin \mathrm{REC}_i$, return $\perp$; else, parse $R_{i,C^{\star}}$ as (k_i, h_i)
- if $\mathsf{req}_i \notin \mathsf{ReqList}_{C^{\star},i}$, increment $Q_{C^{\star},i}$ by 1
- if $(\mathsf{req}_i \notin \mathsf{ReqList}_{C^{\star},i})$:
  - query $\mathcal{O}_{\mathsf{eval}}(i, \mathsf{req}_i)$ to get z_i
- else:
  - let z_i be the value associated with i in the entry $(C^{\star}, \mathsf{req}_i, \ldots, (i, z_i) \ldots)$ in LiveSessions
- $y_i \xleftarrow{\$} \mathsf{TTG.PartEval}(\mathsf{tsk}_i, C^{\star} \| x)$
- set $\mathsf{res}_i := (z_i, \mathsf{SKE.Encrypt}(h_i, y_i))$
- increment $Q_{C^{\star},x}$ by 1
- return res_i

Figure 5.14: Description of adversary $\mathcal{B}$.

scheme while $\mathcal{B}'$ is supposed to guess the TOPRF output h on the (hidden) password of $C^\star$.

As a result, $\mathcal{B}'$'s behavior differs from $\mathcal{B}$ in the following manner. At the start of the simulation, $\mathcal{B}'$ picks random numbers $r_1, \ldots, r_n$ and will use them instead of $h_1, \ldots, h_n$ in the registration oracle. LiveSessions cannot contain h anymore, so when it is needed in the finalize oracle, $r_1, \ldots, r_n$ will be used once again. If $\mathcal{A}$ queries Hash on $h' \| i$ at any time, $\mathcal{B}'$ will query $\mathcal{O}_{\mathsf{check}}$ on h' to check if $h' = h$ or not. If $\mathcal{O}_{\mathsf{check}}$ returns 1, then $\mathcal{B}'$ sends r_i to $\mathcal{A}$.

This also gives a way for $\mathcal{B}'$ to guess h. If $\mathcal{A}$ queries Hash for some $h' \| i$ and $\mathcal{O}_{\mathsf{check}}$ returns 1 on h', then $\mathcal{B}'$ just outputs h' in the unpredictability game. However, we don't have the guarantee that $\mathcal{A}$ will make such a query. All we know is that $\mathcal{A}$ can produce a valid token. Hence, we must argue that $\mathcal{A}$ can produce a valid token only if it queries Hash on h.

Any token share sent by the i-th server is encrypted with h_i. At a high level, $\mathcal{A}$ needs to decrypt at least one token share from an honest server, say j-th, to construct a token. The only way to get this key is by querying Hash on $h \| j$.

5.5 Performance Evaluation

We implement PASTA for four types of threshold token generation schemes: a block-cipher based MAC [NPR99], a DDH-based (requires exponentiations) MAC [NPR99], a pairing based signature [Bol03] and an RSA based signature [Sho00]. In this section we report on the performance of these instantiations.

5.5.1 Implementation Details

PASTA is a generic construction consisting of two building blocks: a threshold oblivious pseudo-random function and a threshold token generation scheme. We implement PASTA with the 2HashTDH TOPRF protocol of Jarecki et al. [JKKX17] and the aforementioned TTG schemes (see Section 5.2.2 for their descriptions) to obtain four types of tokens. To the best of our knowledge, most of these TTG schemes were not implemented before.

We implement pseudorandom functions (PRFs) using AES-NI and hash functions using Blake2 [Blake2]. The elliptic curve operations, pairing operations, and RSA operations are implemented using the Relic library [Relic]. The key length in AES-NI is 128 bits. The cyclic group used in 2HashTDH TOPRF and the DDH based MAC is the group $\mathbb{G}_1$ on 256-bit Barreto-Naehrig curves (BN-curves) [BN06]. Pairing is implemented on 256-bit BN-curves. The key length in RSA based signature is 2048 bits.

In order to evaluate the performance, we implement various settings described below. The experiments are run on a single server with 2x 24-core 2.2 GHz CPUs and 64 GB of RAM. We run all the parties on different cores of the same server (1 core per server), and simulate network connections using the Linux tc command: a LAN setting with 10 Gbps network bandwidth and 0.1 ms round-trip latency; a WAN setting with 40 Mbps network bandwidth and a simulated 80 ms round-trip latency.

5.5.2 Token Generation Time

Table 5.1 shows the total runtime for a client to generate a single token in the sign-on phase after registration in our PASTA protocol. We show experiments for various types of tokens in the LAN and WAN settings and different values of (n, t) where n is the number of servers and t is the threshold. The reported time is an average of 10,000 token requests. We discuss a few observations below.

	(n,t)	Sym-MAC	Public-MAC	Pairing-Sig	RSA-Sig
LAN	$(2,2)$	1.3	1.7	1.7	14.5
	$(3,2)$	1.3	1.7	1.7	14.5
	$(6,2)$	1.3	1.7	1.7	14.5
	$(10,2)$	1.3	1.7	1.7	14.5
	$(10,3)$	1.6	2.1	2.1	15.1
	$(10,5)$	2.3	3.0	3.0	16.8
	$(10,7)$	3.0	3.9	3.9	19.1
	$(10,10)$	4.1	5.4	5.4	22.6
WAN	$(2,2)$	81.4	81.8	81.8	94.6
	$(3,2)$	81.4	81.8	81.8	94.6
	$(6,2)$	81.4	81.8	81.8	94.6
	$(10,2)$	81.4	81.9	81.9	94.6
	$(10,3)$	81.7	82.2	82.2	95.0
	$(10,5)$	82.4	83.1	83.1	96.9
	$(10,7)$	83.1	83.9	83.9	99.2
	$(10,10)$	84.2	85.4	85.4	102.8

Table 5.1: Total runtime (in milliseconds) of our PASTA protocol for generating a single token for the number of servers n and threshold t in LAN and WAN settings.

Notice that for the same threshold $t = 2$ and the same type of token, different values of n result in similar runtime. This is aligned with our construction: for a threshold t, the client only needs to communicate with t servers, and the communication and computation cost for every server is the same, hence the total runtime should also be the same. Therefore, the total runtime is independent of n and only depends on the threshold t. For other values of threshold t, we only report the runtime for $n = 10$; the runtime for other values of n would be roughly the same.

Also notice that for the same (n, t) and same type of token, the runtime in the WAN setting is roughly the runtime in the LAN setting plus 80 ms round-trip latency. This is because in our protocol, the client sends a request to t servers and receive their responses in parallel. The communication complexity is very small, hence the bulk of communication overhead is roughly the round-trip latency. It is worth noting that the PASTA protocol has the minimal two rounds of interaction, and hence this overhead is inevitable in the WAN setting.

The runtime of public-key based MAC and pairing based signature are almost the same under the same setting. This is because in our implementation, TTG schemes for public-key based MAC and pairing based signature are both implemented on the 256-bit Barreto-Naehrig curves (BN-curves) [BN06] in group $\mathbb{G}_1$. This group supports Type-3 pairing operation and is believed to satisfy the Decisional Diffie-Hellman (DDH) assumption, hence a good fit for both primitives.

We do not report the runtime for user registration because (i) it is done only once for every user and (ii) it is more efficient than obtaining a token.

5.5.3 Time Breakdown

We show the runtime breakdown for three different (n, t) values in Table 5.2 in the LAN setting. For each value of (n, t) in the table, the first row is the total runtime, and the second and third rows are the computation time on the client side and on a single server, respectively.

	Sym-MAC	Public-MAC	Pairing-Sig	RSA-Sig
$(10, 2)$	1.3	1.7	1.7	14.5
Client	1.0	1.2	1.2	2.8
Server	0.2	0.4	0.4	11.4
$(10, 5)$	2.3	3.0	3.0	16.8
Client	1.9	2.4	2.4	5.2
Server	0.2	0.4	0.4	11.4
$(10, 10)$	4.1	5.4	5.4	22.6
Client	3.7	4.6	4.6	10.7
Server	0.2	0.4	0.4	11.5

Table 5.2: Breakdown of runtime (in milliseconds) in LAN setting.

As shown in the table, for the same token type the computation time on a single server does not vary. On the other hand, the computation on the client grows with the threshold. Figure 5.15 shows the dependence of the computation time at the client side on the threshold t. For all four types of tokens, the computation time grows linearly in the threshold t.

5.5.4 Comparison with Naïve Solutions

We implement two naïve solutions to compare with our PASTA protocol:

- **Plain Solution.** The client signs on to a single server with its username/password. The server verifies its credential and then issues an authentication token using a master secret key.

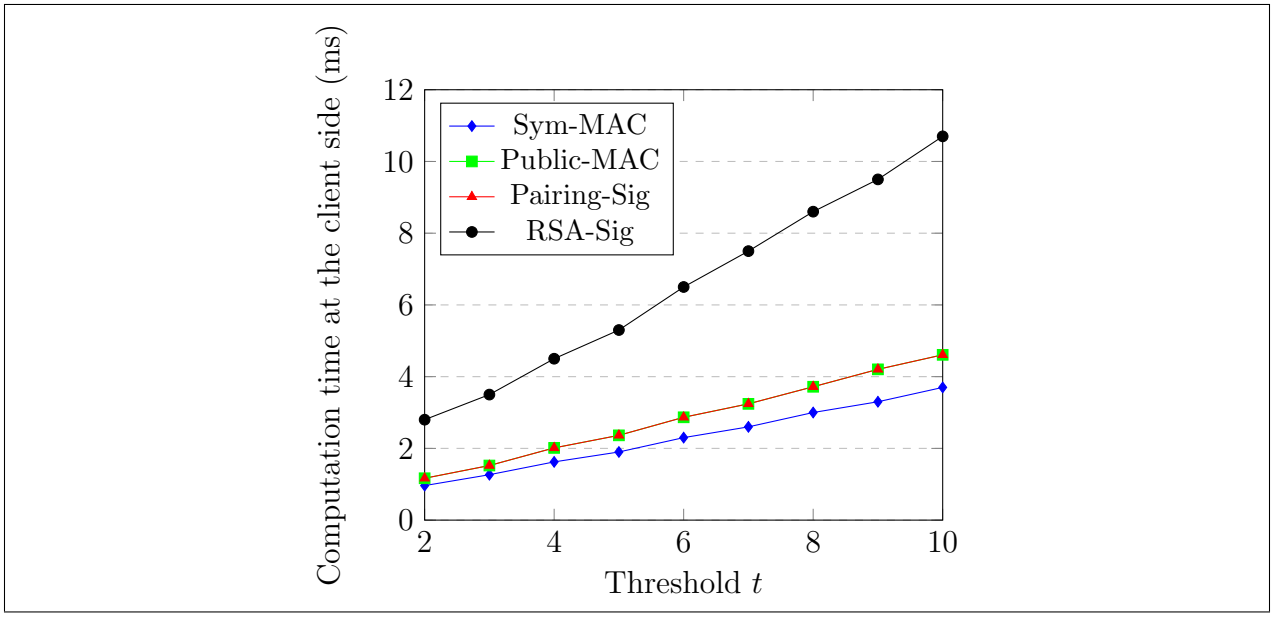


Figure 5.15: Growth of computation time (in milliseconds) at client's side with threshold t (where n is fixed to 10).

- **Threshold Solution.** This solution utilizes a threshold token generation scheme. The secret key shares sk_1, sk_2, ..., sk_n of the threshold scheme are distributed among the n servers. The client signs on with its username/password to t servers, where each server verifies its credential and then issues a share of the token. The client combines the shares received from the servers to generate the final token.

In the plain solution, a breached server would enable the attacker to (i) recover the master secret key and (ii) perform offline dictionary attacks to recover users' passwords. Comparing our solution with the plain solution presents the extra cost of protecting both the master secret key and users' passwords. In the threshold solution, if up to $t-1$ servers are breached, then the master secret key stays secure, but users' passwords are vulnerable to offline dictionary attacks. Comparing our solution with the threshold solution gives the extra cost of protecting users' passwords.

Table 5.3 shows the total runtime for a client to generate a single token after registration using the plain solution and the threshold solution for different values of (n, t). The reported time is an average of 10,000 token requests in LAN and WAN settings. For the same setting and the same type of token, the runtime in the WAN network is roughly the runtime in the LAN network plus 80 ms round-trip latency, for the same reason discussed above for the PASTA protocol. Notice that in the threshold solution, the total runtime is independent of n and only depends on the threshold t. Hence we only report the runtime for the same $n = 10$ and different thresholds.

We compare our solution with the two naïve solutions and show the multiplicative overhead of our solution in Figure 5.16. The two figures represent the comparison in the LAN

		(n,t)	Sym-MAC	Public-MAC	Pairing-Sig	RSA-Sig
LAN	Plain		0.1	0.4	0.4	11.3
	Threshold	$(10,2)$	0.1	0.6	0.6	13.2
		$(10,3)$	0.1	0.6	0.6	13.3
		$(10,5)$	0.2	0.9	0.9	14.4
		$(10,7)$	0.3	1.2	1.2	16.0
		$(10,10)$	0.4	1.5	1.5	18.6
WAN	Plain		80.2	80.5	80.5	91.4
	Threshold	$(10,2)$	80.2	80.7	80.7	93.3
		$(10,3)$	80.2	80.7	80.7	93.4
		$(10,5)$	80.3	81.0	81.0	94.5
		$(10,7)$	80.4	81.3	81.3	96.1
		$(10,10)$	80.5	81.6	81.6	98.8

Table 5.3: Total runtime (in milliseconds) for generating a single token through naïve solutions, for various settings in LAN and WAN networks.

and WAN network, respectively. Different types of tokens are represented in different colors. In each picture, the first set of four bars represent the overhead of our solution compared to the plain solution. Note that there is no notion of (n,t) in the plain solution, hence we pick a setting $(5,3)$ for our solution to compare with the plain solution. If comparing the plain solution with other (n,t) settings of our solution, the results would be slightly different. The remaining five sets of bars in each figure represent the overhead of our solution compared to the threshold solution for various values of (n,t). When comparing with those, we use the same (n,t) setting of our solution.

In the LAN network, notice that there is nearly no overhead for the RSA-based token generation. The overhead for public-key based MAC and pairing based signature is a small constant. There is a higher overhead for symmetric-key based MAC. This is because the naïve solutions only involve symmetric-key operations while our solution involves public-key operations, which is much more expensive. This overhead is necessary as we prove in Section 5.6 that public-key operations are necessary to achieve a password-based threshold authentication (PbTA) system.

In the WAN network, since the most time-consuming component is the network latency in our protocol as well as the naïve solutions, the overhead of our solution compared with the naïve solutions is fairly small. As shown in Figure 5.16, the overhead is less than 5% in all the settings and all token types.

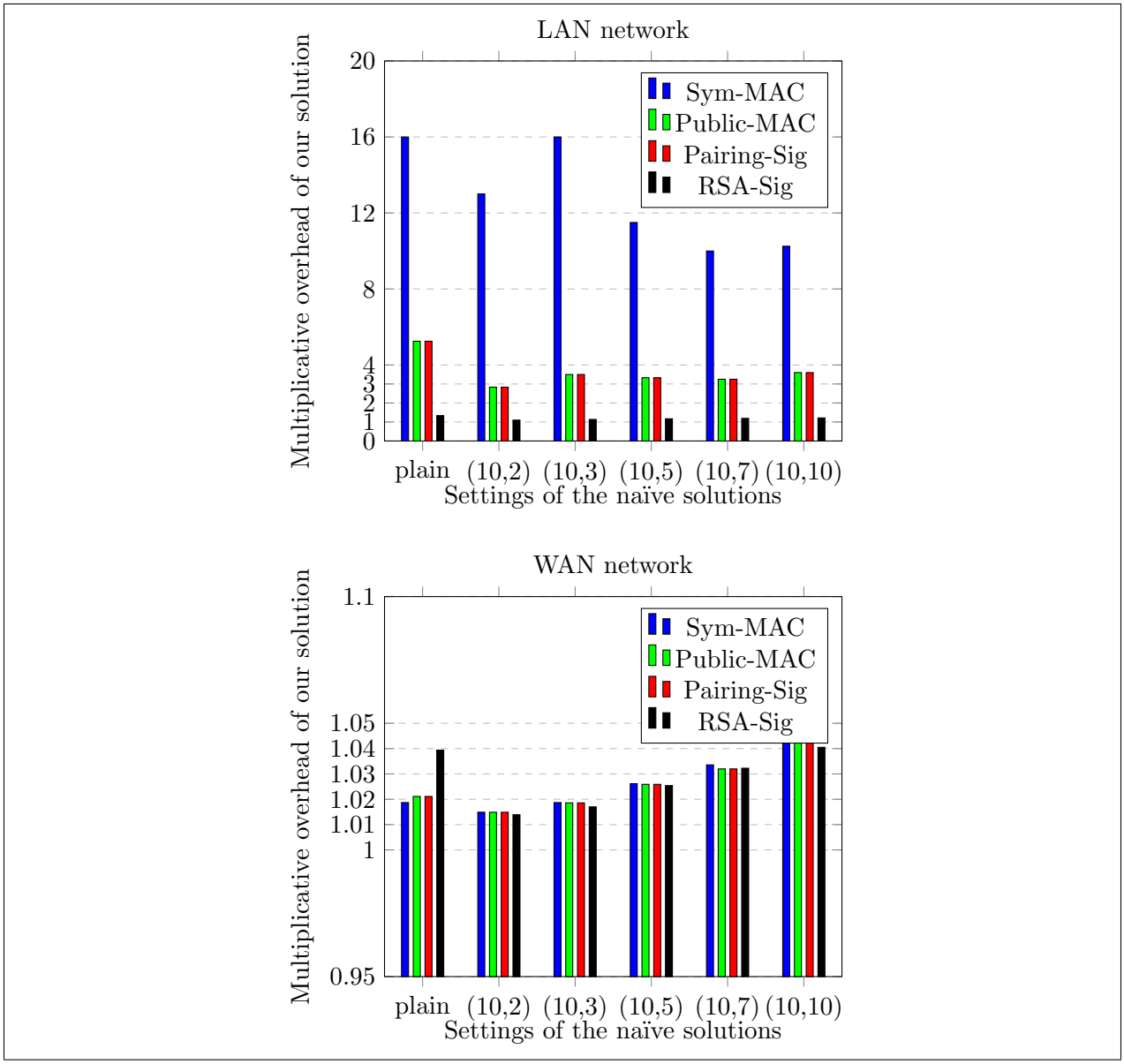


Figure 5.16: Multiplicative overhead of our solution in runtime compared to naïve solutions in LAN and WAN networks.

5.6 Necessity of Public-Key Operations

In both the registration phase and sign-on phase of PASTA, we instantiate the TOPRF component with the 2HashTDH protocol of Jarecki et al. [JKKX17] which uses public-key operations. Therefore, all the instantiations of PASTA use public-key operations even if the threshold token generation scheme is purely symmetric-key. This could become a significant overhead in some cases compared to the naïve insecure solutions (see Section 5.5.4 for details).

So the natural question is whether public-key operations can be avoided, or, put differently, is it just an artifact of PASTA and its instantiations? In this section we prove that public-key operations are indeed necessary to construct any secure PbTA scheme.

In more detail, we prove that if one can construct PbTA that only makes black-box use of one-way functions, then a secure two-party key agreement protocol can also be constructed by only making black-box use of one-way functions, which would imply $P \neq NP$ [IR90]. Hence this gives us evidence that it is unlikely to construct PbTA using only symmetric-key operations.

Overview. We now give an overview of our proof technique. At a high level, we construct a secure key-agreement protocol from PbTA in a black-box way. As a result, if one can construct PbTA that only makes black-box use of one-way functions, then our construction is a secure key-agreement protocol that only makes black-box use of one-way functions, contradicting the impossibility result of Impagliazzo and Rudich [IR90].

To construct the secure key-agreement protocol, think of the two parties P_1 and P_2 in the key-agreement protocol as a client C and the set of all servers in the PbTA protocol, respectively. We set the password space to contain only one password pwd, which means the password of C is known to both parties. Thus P_2, which represents the set of all servers, can run $\mathsf{GlobalSetup}$ and the registration phase of C on its own. Then the two parties run the sign-on phase so that P_1 obtains a token for $x = 0$. Since both parties know the password, P_2 can emulate the sign-on phase on its own to generate a token for $x = 0$. The resulting token is the agreed key by both parties.

Notice that the generated token might not be a valid output for the key agreement protocol, but the two parties can apply randomness extractor to the token and obtain randomness to generate a valid key. We omit the details here.

The security of the key-agreement protocol relies on the unforgeability of the PbTA scheme. Intuitively speaking, if there exists a PPT adversary that outputs the agreed token by P_1 and P_2 with non-negligible probability, then the adversary is able to generate a valid token in the PbTA scheme with non-negligible probability, without making any fake requests to the servers (thus keeping all $Q_{C,i}$ to zero), contradicting the unforgeability property. Next we give provide a formal proof.

Theorem 5.6.1. *A secure two-party key agreement protocol can be constructed from any PbTA scheme in a black-box way.*

Proof. Let $\Pi = (\mathsf{GlobalSetup}, \mathsf{SignUp}, \mathsf{Request}, \mathsf{Respond}, \mathsf{Finalize}, \mathsf{Verify})$ be a PbTA scheme. The secure two-party key agreement protocol is presented in Figure 5.17.

The protocol uses PbTA in a black-box way. Since the tokens $\mathsf{tk}, \mathsf{tk}'$ are generated using the same C, x, and secret key, they are equivalent. Hence the two parties agree on a token (which can be used to extract randomness to generate a key). Now we show that if there exists a PPT adversary $\mathcal{A}$ that outputs the agreed token by P_1, P_2 in the key-agreement

Let the password space be $\mathsf{P} := \{\mathsf{pwd}\}$, set $n := 2, t := 2$, let $\mathcal{T}$ be the set of all servers in Π_{sym}, and set $x := 0$.

1. P_2 executes the following:
   - Run $\mathsf{GlobalSetup}(1^\kappa, n, t, \mathsf{P}) \to (\llbracket \mathsf{sk} \rrbracket, \mathsf{vk}, \mathsf{pp})$.
   - Run $\mathsf{SignUp}(C, \mathsf{pwd}) \to ((C, \mathsf{msg}_1), \ldots, (C, \mathsf{msg}_n))$.
   - Set $R_{i,C} := \mathsf{msg}_i$.
   - Send (pp, C) to P_1.
2. On receiving the first message from P_2, P_1 does the following:
   - Run $\mathsf{Request}(C, \mathsf{pwd}, x, \mathcal{T}) \to (\{(C, x, \mathsf{req}_i)\}_{i\in\mathcal{T}}, \hat{\mathsf{D}})$.
   - Send $\{\mathsf{req}_i\}_{i\in\mathcal{T}}$ to P_2.
3. On receiving the message from P_1, P_2 does the following:
   - Run $\mathsf{Respond}(\mathsf{sk}_i, \mathrm{REC}_i, C, x, \mathsf{req}_i) \to \mathsf{res}_i$ for $i \in \mathcal{T}$.
   - Send $\{\mathsf{res}_i\}_{i\in\mathcal{T}}$ to P_1.
   - Emulate the protocol:
     a. $\mathsf{Request}(C, \mathsf{pwd}, x, \mathcal{T}) \to (\{(C, x, \mathsf{req}'_i)\}_{i\in\mathcal{T}}, \hat{\mathsf{D}}')$.
     b. $\mathsf{Respond}(\mathsf{sk}_i, \mathrm{REC}_i, C, x, \mathsf{req}'_i) \to \mathsf{res}'_i$ for $i \in \mathcal{T}$.
     c. $\mathsf{Finalize}(\hat{\mathsf{D}}', \{\mathsf{res}'_i\}_{i\in\mathcal{T}}) \to \mathsf{tk}'$.
   - Output tk'.
4. On receiving the second message from P_2, P_1 executes the following:
   - Run $\mathsf{Finalize}(\hat{\mathsf{D}}, \{\mathsf{res}_i\}_{i\in\mathcal{T}}) \to \mathsf{tk}$.
   - Output tk.

Figure 5.17: Secure two-party key agreement protocol.

protocol, then we can construct another adversary $\mathcal{B}$ that breaks unforgeability of the PbTA scheme.

$\mathcal{B}$ does not corrupt any server or client. It then calls $\mathcal{O}_{\mathsf{signup}}(C)$ to obtain $\{\mathsf{msg}_i\}_{i\in[n]}$, and calls $\mathcal{O}_{\mathsf{server}}(i, \mathsf{store}, \mathsf{msg}_i)$ to register C for all $i \in \mathcal{T}$. Then it calls $\mathcal{O}_{\mathsf{req}}(C, \mathsf{pwd}, 0, \mathcal{T})$ to obtain $\{\mathsf{req}_i\}_{i\in\mathcal{T}}$, and calls $\mathcal{O}_{\mathsf{server}}(i, \mathsf{respond}, \mathsf{req}_i)$ to obtain res_i for all $i \in \mathcal{T}$. $\mathcal{B}$ runs $\mathcal{A}$ with input being the transcript of the key-agreement protocol, consisting of $\{(\mathsf{pp}, C), \{\mathsf{req}_i\}_{i\in\mathcal{T}}, \{\mathsf{res}_i\}_{i\in\mathcal{T}}\}$, and obtains an output $\widetilde{\mathsf{tk}}$ from $\mathcal{A}$. Then $\mathcal{B}$ simply outputs $(C, 0, \widetilde{\mathsf{tk}})$.

In the security game $\mathsf{SecGame}_{\Pi,\mathcal{A}}(1^\kappa, n, t, \mathsf{P})$ (Figure 5.10) for $\mathcal{B}$, we have $Q_{C,i} = 0$ for all

i. By the unforgeability definition, there exists a negligible function negl such that

$$\Pr[\mathsf{Verify}(\mathsf{vk}, C, 0, \widetilde{\mathsf{tk}}) = 1] \leq \mathsf{negl}(\kappa).$$

However, $\mathcal{A}$'s token $\widetilde{\mathsf{tk}}$ is valid with non-negligible probability, leading to a contradiction. □

Combining the above theorem with the result of Impagliazzo and Rudich [IR90], we have the following corollary:

Corollary 5.6.2. *If there exists a PbTA scheme that only makes black-box use of one-way functions, then $P \neq NP$.*

This corollary provides us with evidence that it is unlikely to construct PbTA schemes that only makes black-box use of one-way functions. Notice that we did not rule out the possibility of getting around this problem by making non-black-box use of one-way functions. We leave that as an interesting open problem.

Bibliography

[Blake2] *BLAKE2 - fast secure hashing*. https://blake2.net/. Accessed on May 16, 2019.

[Relic] D. F. Aranha and C. P. L. Gouvêa. *RELIC is an Efficient LIbrary for Cryptography*. https://github.com/relic-toolkit/relic.

[Scrypt] Tarsnap. *Scrypt*. https://github.com/Tarsnap/scrypt. Github Repository: Accessed on May 16, 2019.

[ACFP05] Michel Abdalla, Olivier Chevassut, Pierre-Alain Fouque, and David Pointcheval. "A Simple Threshold Authenticated Key Exchange from Short Secrets". In: *ASIACRYPT 2005*. Ed. by Bimal K. Roy. Vol. 3788. LNCS. Springer, Heidelberg, Dec. 2005, pp. 566–584. DOI: 10.1007/11593447_31.

[ACK+16] Joël Alwen, Binyi Chen, Chethan Kamath, Vladimir Kolmogorov, Krzysztof Pietrzak, and Stefano Tessaro. "On the Complexity of Scrypt and Proofs of Space in the Parallel Random Oracle Model". In: *EUROCRYPT 2016, Part II*. Ed. by Marc Fischlin and Jean-Sébastien Coron. Vol. 9666. LNCS. Springer, Heidelberg, May 2016, pp. 358–387. DOI: 10.1007/978-3-662-49896-5_13.

[ACNP16] Michel Abdalla, Mario Cornejo, Anca Nitulescu, and David Pointcheval. "Robust Password-Protected Secret Sharing". In: *ESORICS 2016, Part II*. Ed. by Ioannis G. Askoxylakis, Sotiris Ioannidis, Sokratis K. Katsikas, and Catherine A. Meadows. Vol. 9879. LNCS. Springer, Heidelberg, Sept. 2016, pp. 61–79. DOI: 10.1007/978-3-319-45741-3_4.

[ACP+17] Joël Alwen, Binyi Chen, Krzysztof Pietrzak, Leonid Reyzin, and Stefano Tessaro. "Scrypt Is Maximally Memory-Hard". In: *EUROCRYPT 2017, Part III*. Ed. by Jean-Sébastien Coron and Jesper Buus Nielsen. Vol. 10212. LNCS. Springer, Heidelberg, Apr. 2017, pp. 33–62. DOI: 10.1007/978-3-319-56617-7_2.

[ADT11] Giuseppe Ateniese, Emiliano De Cristofaro, and Gene Tsudik. "(If) Size Matters: Size-Hiding Private Set Intersection". In: *PKC 2011*. Ed. by Dario Catalano, Nelly Fazio, Rosario Gennaro, and Antonio Nicolosi. Vol. 6571. LNCS. Springer, Heidelberg, Mar. 2011, pp. 156–173. DOI: 10.1007/978-3-642-19379-8_10.

[AFP05] Michel Abdalla, Pierre-Alain Fouque, and David Pointcheval. "Password-Based Authenticated Key Exchange in the Three-Party Setting". In: *PKC 2005*. Ed. by Serge Vaudenay. Vol. 3386. LNCS. Springer, Heidelberg, Jan. 2005, pp. 65–84. DOI: 10.1007/978-3-540-30580-4_6.

[AIKW13] Benny Applebaum, Yuval Ishai, Eyal Kushilevitz, and Brent Waters. "Encoding Functions with Constant Online Rate or How to Compress Garbled Circuits Keys". In: *CRYPTO 2013, Part II*. Ed. by Ran Canetti and Juan A. Garay. Vol. 8043. LNCS. Springer, Heidelberg, Aug. 2013, pp. 166–184. DOI: 10.1007/978-3-642-40084-1_10.

[AIR01] William Aiello, Yuval Ishai, and Omer Reingold. "Priced Oblivious Transfer: How to Sell Digital Goods". In: *EUROCRYPT 2001*. Ed. by Birgit Pfitzmann. Vol. 2045. LNCS. Springer, Heidelberg, May 2001, pp. 119–135. DOI: 10.1007/3-540-44987-6_8.

[ALSZ13] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. "More efficient oblivious transfer and extensions for faster secure computation". In: *ACM CCS 2013*. Ed. by Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung. ACM Press, Nov. 2013, pp. 535–548. DOI: 10.1145/2508859.2516738.

[Amazon] *Amazon OpenID*. https://docs.aws.amazon.com/cognito/latest/developerguide/open-id.html. Accessed on May 16, 2019.

[AMMM18] Shashank Agrawal, Peihan Miao, Payman Mohassel, and Pratyay Mukherjee. "PASTA: PASsword-based Threshold Authentication". In: *ACM CCS 2018*. Ed. by David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang. ACM Press, Oct. 2018, pp. 2042–2059. DOI: 10.1145/3243734.3243839.

[AMN01] Michel Abdalla, Sara K. Miner, and Chanathip Namprempre. "Forward-Secure Threshold Signature Schemes". In: *CT-RSA 2001*. Ed. by David Naccache. Vol. 2020. LNCS. Springer, Heidelberg, Apr. 2001, pp. 441–456. DOI: 10.1007/3-540-45353-9_32.

[BCCT12] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. "From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again". In: *ITCS 2012*. Ed. by Shafi Goldwasser. ACM, Jan. 2012, pp. 326–349. DOI: 10.1145/2090236.2090263.

[BCG+13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. "SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge". In: *CRYPTO 2013, Part II*. Ed. by Ran Canetti and Juan A. Garay. Vol. 8043. LNCS. Springer, Heidelberg, Aug. 2013, pp. 90–108. DOI: 10.1007/978-3-642-40084-1_6.

[BD16] Jeremiah Blocki and Anupam Datta. "CASH: A Cost Asymmetric Secure Hash Algorithm for Optimal Password Protection". In: *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*. 2016, pp. 371–386. DOI: 10.1109/CSF.2016.33. URL: https://doi.org/10.1109/CSF.2016.33.

[BdMW16] Florian Bourse, Rafaël del Pino, Michele Minelli, and Hoeteck Wee. "FHE Circuit Privacy Almost for Free". In: *CRYPTO 2016, Part II*. Ed. by Matthew Robshaw and Jonathan Katz. Vol. 9815. LNCS. Springer, Heidelberg, Aug. 2016, pp. 62–89. DOI: 10.1007/978-3-662-53008-5_3.

[Bea96] Donald Beaver. "Correlated Pseudorandomness and the Complexity of Private Computations". In: *28th ACM STOC*. ACM Press, May 1996, pp. 479–488. DOI: 10.1145/237814.237996.

[BGL+15] Nir Bitansky, Sanjam Garg, Huijia Lin, Rafael Pass, and Sidharth Telang. "Succinct Randomized Encodings and their Applications". In: *47th ACM STOC*. Ed. by Rocco A. Servedio and Ronitt Rubinfeld. ACM Press, June 2015, pp. 439–448. DOI: 10.1145/2746539.2746574.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. "Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation (Extended Abstract)". In: *20th ACM STOC*. ACM Press, May 1988, pp. 1–10. DOI: 10.1145/62212.62213.

[BHHO08] Dan Boneh, Shai Halevi, Michael Hamburg, and Rafail Ostrovsky. "Circular-Secure Encryption from Decision Diffie-Hellman". In: *CRYPTO 2008*. Ed. by David Wagner. Vol. 5157. LNCS. Springer, Heidelberg, Aug. 2008, pp. 108–125. DOI: 10.1007/978-3-540-85174-5_7.

[BHR12] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. "Foundations of garbled circuits". In: *ACM CCS 2012*. Ed. by Ting Yu, George Danezis, and Virgil D. Gligor. ACM Press, Oct. 2012, pp. 784–796. DOI: 10.1145/2382196.2382279.

[BJSL11] Ali Bagherzandi, Stanislaw Jarecki, Nitesh Saxena, and Yanbin Lu. "Password-protected secret sharing". In: *ACM CCS 2011*. Ed. by Yan Chen, George Danezis, and Vitaly Shmatikov. ACM Press, Oct. 2011, pp. 433–444. DOI: 10.1145/2046707.2046758.

[BLMR13] Dan Boneh, Kevin Lewi, Hart William Montgomery, and Ananth Raghunathan. "Key Homomorphic PRFs and Their Applications". In: *CRYPTO 2013, Part I*. Ed. by Ran Canetti and Juan A. Garay. Vol. 8042. LNCS. Springer, Heidelberg, Aug. 2013, pp. 410–428. DOI: 10.1007/978-3-642-40041-4_23.

[BLSV18] Zvika Brakerski, Alex Lombardi, Gil Segev, and Vinod Vaikuntanathan. "Anonymous IBE, Leakage Resilience and Circular Security from New Assumptions". In: *EUROCRYPT 2018, Part I*. Ed. by Jesper Buus Nielsen and Vincent Rijmen. Vol. 10820. LNCS. Springer, Heidelberg, Apr. 2018, pp. 535–564. DOI: 10.1007/978-3-319-78381-9_20.

[BMP00] Victor Boyko, Philip D. MacKenzie, and Sarvar Patel. "Provably Secure Password-Authenticated Key Exchange Using Diffie-Hellman". In: *EUROCRYPT 2000*. Ed. by Bart Preneel. Vol. 1807. LNCS. Springer, Heidelberg, May 2000, pp. 156–171. DOI: 10.1007/3-540-45539-6_12.

[BN06] Paulo S. L. M. Barreto and Michael Naehrig. "Pairing-Friendly Elliptic Curves of Prime Order". In: *SAC 2005*. Ed. by Bart Preneel and Stafford Tavares. Vol. 3897. LNCS. Springer, Heidelberg, Aug. 2006, pp. 319–331. DOI: 10.1007/11693383_22.

[Bol03] Alexandra Boldyreva. "Threshold Signatures, Multisignatures and Blind Signatures Based on the Gap-Diffie-Hellman-Group Signature Scheme". In: *PKC 2003*. Ed. by Yvo Desmedt. Vol. 2567. LNCS. Springer, Heidelberg, Jan. 2003, pp. 31–46. DOI: 10.1007/3-540-36288-6_3.

[BPR00] Mihir Bellare, David Pointcheval, and Phillip Rogaway. "Authenticated Key Exchange Secure against Dictionary Attacks". In: *EUROCRYPT 2000*. Ed. by Bart Preneel. Vol. 1807. LNCS. Springer, Heidelberg, May 2000, pp. 139–155. DOI: 10.1007/3-540-45539-6_11.

[BS01] Mihir Bellare and Ravi Sandhu. *The Security of Practical Two-Party RSA Signature Schemes*. Cryptology ePrint Archive, Report 2001/060. http://eprint.iacr.org/2001/060. 2001.

[BV11a] Zvika Brakerski and Vinod Vaikuntanathan. "Efficient Fully Homomorphic Encryption from (Standard) LWE". In: *52nd FOCS*. Ed. by Rafail Ostrovsky. IEEE Computer Society Press, Oct. 2011, pp. 97–106. DOI: 10.1109/FOCS.2011.12.

[BV11b] Zvika Brakerski and Vinod Vaikuntanathan. "Fully Homomorphic Encryption from Ring-LWE and Security for Key Dependent Messages". In: *CRYPTO 2011*. Ed. by Phillip Rogaway. Vol. 6841. LNCS. Springer, Heidelberg, Aug. 2011, pp. 505–524. DOI: 10.1007/978-3-642-22792-9_29.

[BZ17] Jeremiah Blocki and Samson Zhou. "On the Depth-Robustness and Cumulative Pebbling Cost of Argon2i". In: *TCC 2017, Part I*. Ed. by Yael Kalai and Leonid Reyzin. Vol. 10677. LNCS. Springer, Heidelberg, Nov. 2017, pp. 445–465. DOI: 10.1007/978-3-319-70500-2_15.

[Can01] Ran Canetti. "Universally Composable Security: A New Paradigm for Cryptographic Protocols". In: *42nd FOCS*. IEEE Computer Society Press, Oct. 2001, pp. 136–145. DOI: 10.1109/SFCS.2001.959888.

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgård. "Multiparty Unconditionally Secure Protocols (Extended Abstract)". In: *20th ACM STOC*. ACM Press, May 1988, pp. 11–19. DOI: 10.1145/62212.62214.

[CDG+17] Chongwon Cho, Nico Döttling, Sanjam Garg, Divya Gupta, Peihan Miao, and Antigoni Polychroniadou. "Laconic Oblivious Transfer and Its Applications". In: *CRYPTO 2017, Part II*. Ed. by Jonathan Katz and Hovav Shacham. Vol. 10402. LNCS. Springer, Heidelberg, Aug. 2017, pp. 33–65. DOI: 10.1007/978-3-319-63715-0_2.

[CEN15] Jan Camenisch, Robert R. Enderlein, and Gregory Neven. "Two-Server Password-Authenticated Secret Sharing UC-Secure Against Transient Corruptions". In: *PKC 2015*. Ed. by Jonathan Katz. Vol. 9020. LNCS. Springer, Heidelberg, Mar. 2015, pp. 283–307. DOI: 10.1007/978-3-662-46447-2_13.

[CHJV15] Ran Canetti, Justin Holmgren, Abhishek Jain, and Vinod Vaikuntanathan. "Succinct Garbling and Indistinguishability Obfuscation for RAM Programs". In: *47th ACM STOC*. Ed. by Rocco A. Servedio and Ronitt Rubinfeld. ACM Press, June 2015, pp. 429–437. DOI: 10.1145/2746539.2746621.

[CHK+05] Ran Canetti, Shai Halevi, Jonathan Katz, Yehuda Lindell, and Philip D. MacKenzie. "Universally Composable Password-Based Key Exchange". In: *EUROCRYPT 2005*. Ed. by Ronald Cramer. Vol. 3494. LNCS. Springer, Heidelberg, May 2005, pp. 404–421. DOI: 10.1007/11426639_24.

[CHK04] Ran Canetti, Shai Halevi, and Jonathan Katz. "Chosen-Ciphertext Security from Identity-Based Encryption". In: *EUROCRYPT 2004*. Ed. by Christian Cachin and Jan Camenisch. Vol. 3027. LNCS. Springer, Heidelberg, May 2004, pp. 207–222. DOI: 10.1007/978-3-540-24676-3_13.

[CLLN14] Jan Camenisch, Anja Lehmann, Anna Lysyanskaya, and Gregory Neven. "Memento: How to Reconstruct Your Secrets from a Single Password in a Hostile Environment". In: *CRYPTO 2014, Part II*. Ed. by Juan A. Garay and Rosario Gennaro. Vol. 8617. LNCS. Springer, Heidelberg, Aug. 2014, pp. 256–275. DOI: 10.1007/978-3-662-44381-1_15.

[CLN12] Jan Camenisch, Anna Lysyanskaya, and Gregory Neven. "Practical yet universally composable two-server password-authenticated secret sharing". In: *ACM CCS 2012*. Ed. by Ting Yu, George Danezis, and Virgil D. Gligor. ACM Press, Oct. 2012, pp. 525–536. DOI: 10.1145/2382196.2382252.

[CLN15] Jan Camenisch, Anja Lehmann, and Gregory Neven. "Optimal Distributed Password Verification". In: *ACM CCS 2015*. Ed. by Indrajit Ray, Ninghui Li, and Christopher Kruegel. ACM Press, Oct. 2015, pp. 182–194. DOI: 10.1145/2810103.2813722.

[CLNS16] Jan Camenisch, Anja Lehmann, Gregory Neven, and Kai Samelin. "Virtual Smart Cards: How to Sign with a Password and a Server". In: *SCN 16*. Ed. by Vassilis Zikas and Roberto De Prisco. Vol. 9841. LNCS. Springer, Heidelberg, Aug. 2016, pp. 353–371. DOI: 10.1007/978-3-319-44618-9_19.

[COV15] Melissa Chase, Rafail Ostrovsky, and Ivan Visconti. "Executable Proofs, Input-Size Hiding Secure Computation and a New Ideal World". In: *EUROCRYPT 2015, Part II*. Ed. by Elisabeth Oswald and Marc Fischlin. Vol. 9057. LNCS. Springer, Heidelberg, Apr. 2015, pp. 532–560. DOI: 10.1007/978-3-662-46803-6_18.

[CS02] Ronald Cramer and Victor Shoup. "Universal Hash Proofs and a Paradigm for Adaptive Chosen Ciphertext Secure Public-Key Encryption". In: *EUROCRYPT 2002*. Ed. by Lars R. Knudsen. Vol. 2332. LNCS. Springer, Heidelberg, Apr. 2002, pp. 45–64. DOI: 10.1007/3-540-46035-7_4.

[CS98] Ronald Cramer and Victor Shoup. "A Practical Public Key Cryptosystem Provably Secure Against Adaptive Chosen Ciphertext Attack". In: *CRYPTO'98*. Ed. by Hugo Krawczyk. Vol. 1462. LNCS. Springer, Heidelberg, Aug. 1998, pp. 13–25. DOI: 10.1007/BFb0055717.

[CV12] Melissa Chase and Ivan Visconti. "Secure Database Commitments and Universal Arguments of Quasi Knowledge". In: *CRYPTO 2012*. Ed. by Reihaneh Safavi-Naini and Ran Canetti. Vol. 7417. LNCS. Springer, Heidelberg, Aug. 2012, pp. 236–254. DOI: 10.1007/978-3-642-32009-5_15.

[Dan] Daniel Sewell. *Offline Password Cracking: The Attack and the Best Defense Against It*. https://www.alpinesecurity.com/blog/offline-password-cracking-the-attack-and-the-best-defense-against-it. Accessed on May 16, 2019.

[DDFY94] Alfredo De Santis, Yvo Desmedt, Yair Frankel, and Moti Yung. "How to Share a Function Securely". In: *26th ACM STOC*. ACM Press, May 1994, pp. 522–533. DOI: 10.1145/195058.195405.

[DF90] Yvo Desmedt and Yair Frankel. "Threshold Cryptosystems". In: *CRYPTO'89*. Ed. by Gilles Brassard. Vol. 435. LNCS. Springer, Heidelberg, Aug. 1990, pp. 307–315. DOI: 10.1007/0-387-34805-0_28.

[DG03] Mario Di Raimondo and Rosario Gennaro. "Provably Secure Threshold Password-Authenticated Key Exchange". In: *EUROCRYPT 2003*. Ed. by Eli Biham. Vol. 2656. LNCS. Springer, Heidelberg, May 2003, pp. 507–523. DOI: 10.1007/3-540-39200-9_32.

[DG17] Nico Döttling and Sanjam Garg. "Identity-Based Encryption from the Diffie-Hellman Assumption". In: *CRYPTO 2017, Part I*. Ed. by Jonathan Katz and Hovav Shacham. Vol. 10401. LNCS. Springer, Heidelberg, Aug. 2017, pp. 537–569. DOI: 10.1007/978-3-319-63688-7_18.

[DGHM18] Nico Döttling, Sanjam Garg, Mohammad Hajiabadi, and Daniel Masny. "New Constructions of Identity-Based and Key-Dependent Message Secure Encryption Schemes". In: *PKC 2018, Part I*. Ed. by Michel Abdalla and Ricardo Dahab. Vol. 10769. LNCS. Springer, Heidelberg, Mar. 2018, pp. 3–31. DOI: 10.1007/978-3-319-76578-5_1.

[DK01] Ivan Damgård and Maciej Koprowski. "Practical Threshold RSA Signatures without a Trusted Dealer". In: *EUROCRYPT 2001*. Ed. by Birgit Pfitzmann. Vol. 2045. LNCS. Springer, Heidelberg, May 2001, pp. 152–165. DOI: 10.1007/3-540-44987-6_10.

[DKAN] Daniel Dinu, Dmitry Khovratovich, Jean-Philippe Aumasson, and Samuel Neves. *Argon2*. https://github.com/P-H-C/phc-winner-argon2. Github Repository: Accessed on May 16, 2019.

[DS16] Léo Ducas and Damien Stehlé. "Sanitization of FHE Ciphertexts". In: *EUROCRYPT 2016, Part I*. Ed. by Marc Fischlin and Jean-Sébastien Coron. Vol. 9665. LNCS. Springer, Heidelberg, May 2016, pp. 294–310. DOI: 10.1007/978-3-662-49890-3_12.

[Facebook] *Facebook Login*. https://developers.facebook.com/docs/facebook-login. Accessed on May 16, 2019.

[FIPR05] Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. "Keyword Search and Oblivious Pseudorandom Functions". In: *TCC 2005*. Ed. by Joe Kilian. Vol. 3378. LNCS. Springer, Heidelberg, Feb. 2005, pp. 303–324. DOI: 10.1007/978-3-540-30576-7_17.

[Fis99] Marc Fischlin. "Pseudorandom Function Tribe Ensembles Based on One-Way Permutations: Improvements and Applications". In: *EUROCRYPT'99*. Ed. by Jacques Stern. Vol. 1592. LNCS. Springer, Heidelberg, May 1999, pp. 432–445. DOI: 10.1007/3-540-48910-X_30.

[FLS90] Uriel Feige, Dror Lapidot, and Adi Shamir. "Multiple Non-Interactive Zero Knowledge Proofs Based on a Single Random String (Extended Abstract)". In: *31st FOCS*. IEEE Computer Society Press, Oct. 1990, pp. 308–317. DOI: 10.1109/FSCS.1990.89549.

[Gan95] Ravi Ganesan. "Yaksha: augmenting Kerberos with public key cryptography". In: *1995 Symposium on Network and Distributed System Security, (S)NDSS '95, San Diego, California, February 16-17, 1995*. 1995, pp. 132–143. DOI: 10.1109/NDSS.1995.390639. URL: https://doi.org/10.1109/NDSS.1995.390639.

[Gen09] Craig Gentry. "Fully homomorphic encryption using ideal lattices". In: *41st ACM STOC*. Ed. by Michael Mitzenmacher. ACM Press, May 2009, pp. 169–178. DOI: 10.1145/1536414.1536440.

[GGH+13] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. "Candidate Indistinguishability Obfuscation and Functional Encryption for all Circuits". In: *54th FOCS*. IEEE Computer Society Press, Oct. 2013, pp. 40–49. DOI: 10.1109/FOCS.2013.13.

[GGH13] Sanjam Garg, Craig Gentry, and Shai Halevi. "Candidate Multilinear Maps from Ideal Lattices". In: *EUROCRYPT 2013*. Ed. by Thomas Johansson and Phong Q. Nguyen. Vol. 7881. LNCS. Springer, Heidelberg, May 2013, pp. 1–17. DOI: 10.1007/978-3-642-38348-9_1.

[GGMP16] Sanjam Garg, Divya Gupta, Peihan Miao, and Omkant Pandey. "Secure Multiparty RAM Computation in Constant Rounds". In: *TCC 2016-B, Part I*. Ed. by Martin Hirt and Adam D. Smith. Vol. 9985. LNCS. Springer, Heidelberg, Oct. 2016, pp. 491–520. DOI: 10.1007/978-3-662-53641-4_19.

[GGN16] Rosario Gennaro, Steven Goldfeder, and Arvind Narayanan. "Threshold-Optimal DSA/ECDSA Signatures and an Application to Bitcoin Wallet Security". In: *ACNS 16*. Ed. by Mark Manulis, Ahmad-Reza Sadeghi, and Steve Schneider. Vol. 9696. LNCS. Springer, Heidelberg, June 2016, pp. 156–174. DOI: 10.1007/978-3-319-39555-5_9.

[GGSW13] Sanjam Garg, Craig Gentry, Amit Sahai, and Brent Waters. "Witness encryption and its applications". In: *45th ACM STOC*. Ed. by Dan Boneh, Tim Roughgarden, and Joan Feigenbaum. ACM Press, June 2013, pp. 467–476. DOI: 10.1145/2488608.2488667.

[GHKR08] Rosario Gennaro, Shai Halevi, Hugo Krawczyk, and Tal Rabin. "Threshold RSA for Dynamic and Ad-Hoc Groups". In: *EUROCRYPT 2008*. Ed. by Nigel P. Smart. Vol. 4965. LNCS. Springer, Heidelberg, Apr. 2008, pp. 88–107. DOI: 10.1007/978-3-540-78967-3_6.

[GHL+14] Craig Gentry, Shai Halevi, Steve Lu, Rafail Ostrovsky, Mariana Raykova, and Daniel Wichs. "Garbled RAM Revisited". In: *EUROCRYPT 2014*. Ed. by Phong Q. Nguyen and Elisabeth Oswald. Vol. 8441. LNCS. Springer, Heidelberg, May 2014, pp. 405–422. DOI: 10.1007/978-3-642-55220-5_23.

[GHRW14] Craig Gentry, Shai Halevi, Mariana Raykova, and Daniel Wichs. "Outsourcing Private RAM Computation". In: *55th FOCS*. IEEE Computer Society Press, Oct. 2014, pp. 404–413. DOI: 10.1109/FOCS.2014.50.

[GHV10] Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. "i-Hop Homomorphic Encryption and Rerandomizable Yao Circuits". In: *CRYPTO 2010*. Ed. by Tal Rabin. Vol. 6223. LNCS. Springer, Heidelberg, Aug. 2010, pp. 155–172. DOI: 10.1007/978-3-642-14623-7_9.

[GJKR96] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. "Robust Threshold DSS Signatures". In: *EUROCRYPT'96*. Ed. by Ueli M. Maurer. Vol. 1070. LNCS. Springer, Heidelberg, May 1996, pp. 354–371. DOI: 10.1007/3-540-68339-9_31.

[GK10] Adam Groce and Jonathan Katz. "A new framework for efficient password-based authenticated key exchange". In: *ACM CCS 2010*. Ed. by Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov. ACM Press, Oct. 2010, pp. 516–525. DOI: 10.1145/1866307.1866365.

[GKK+12] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. "Secure two-party computation in sublinear (amortized) time". In: *ACM CCS 2012*. Ed. by Ting Yu, George Danezis, and Virgil D. Gligor. ACM Press, Oct. 2012, pp. 513–524. DOI: 10.1145/2382196.2382251.

[GKP+13] Shafi Goldwasser, Yael Tauman Kalai, Raluca A. Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. "How to Run Turing Machines on Encrypted Data". In: *CRYPTO 2013, Part II*. Ed. by Ran Canetti and Juan A. Garay. Vol. 8043. LNCS. Springer, Heidelberg, Aug. 2013, pp. 536–553. DOI: 10.1007/978-3-642-40084-1_30.

[GLO15] Sanjam Garg, Steve Lu, and Rafail Ostrovsky. "Black-Box Garbled RAM". In: *56th FOCS*. Ed. by Venkatesan Guruswami. IEEE Computer Society Press, Oct. 2015, pp. 210–229. DOI: 10.1109/FOCS.2015.22.

[GLOS15] Sanjam Garg, Steve Lu, Rafail Ostrovsky, and Alessandra Scafuro. "Garbled RAM From One-Way Functions". In: *47th ACM STOC*. Ed. by Rocco A. Servedio and Ronitt Rubinfeld. ACM Press, June 2015, pp. 449–458. DOI: 10.1145/2746539.2746593.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. "How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority". In: *19th ACM STOC*. Ed. by Alfred Aho. ACM Press, May 1987, pp. 218–229. DOI: 10.1145/28395.28420.

[GO96] Oded Goldreich and Rafail Ostrovsky. "Software Protection and Simulation on Oblivious RAMs". In: *J. ACM* 43.3 (1996), pp. 431–473.

[Gol87] Oded Goldreich. "Towards a Theory of Software Protection and Simulation by Oblivious RAMs". In: *19th ACM STOC*. Ed. by Alfred Aho. ACM Press, May 1987, pp. 182–194. DOI: 10.1145/28395.28416.

[Google] *Google Identity Platform – OpenID Connect*. https://developers.google.com/identity/protocols/OpenIDConnect. Accessed on May 16, 2019.

[GOS06] Jens Groth, Rafail Ostrovsky, and Amit Sahai. "Non-interactive Zaps and New Techniques for NIZK". In: *CRYPTO 2006*. Ed. by Cynthia Dwork. Vol. 4117. LNCS. Springer, Heidelberg, Aug. 2006, pp. 97–111. DOI: 10.1007/11818175_6.

[GOS18] Sanjam Garg, Rafail Ostrovsky, and Akshayaram Srinivasan. "Adaptive Garbled RAM from Laconic Oblivious Transfer". In: *CRYPTO 2018, Part III*. Ed. by Hovav Shacham and Alexandra Boldyreva. Vol. 10993. LNCS. Springer, Heidelberg, Aug. 2018, pp. 515–544. DOI: 10.1007/978-3-319-96878-0_18.

[GS18] Sanjam Garg and Akshayaram Srinivasan. "Adaptively Secure Garbling with Near Optimal Online Complexity". In: *EUROCRYPT 2018, Part II*. Ed. by Jesper Buus Nielsen and Vincent Rijmen. Vol. 10821. LNCS. Springer, Heidelberg, Apr. 2018, pp. 535–565. DOI: 10.1007/978-3-319-78375-8_18.

[GSW13] Craig Gentry, Amit Sahai, and Brent Waters. "Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based". In: *CRYPTO 2013, Part I*. Ed. by Ran Canetti and Juan A. Garay. Vol. 8042. LNCS. Springer, Heidelberg, Aug. 2013, pp. 75–92. DOI: 10.1007/978-3-642-40041-4_5.

[GT11] Kristian Gjosteen and Oystein Thuen. "Password-Based Signatures". In: *Public Key Infrastructures, Services and Applications - 8th European Workshop, EuroPKI 2011, Leuven, Belgium, September 15-16, 2011, Revised Selected Papers*. 2011, pp. 17–33. DOI: 10.1007/978-3-642-29804-2_2. URL: https://doi.org/10.1007/978-3-642-29804-2_2.

[HAP18] Yotam Harchol, Ittai Abraham, and Benny Pinkas. "Distributed SSH Key Management with Proactive RSA Threshold Signatures". In: *ACNS 18*. Ed. by Bart Preneel and Frederik Vercauteren. Vol. 10892. LNCS. Springer, Heidelberg, July 2018, pp. 22–43. DOI: 10.1007/978-3-319-93387-0_2.

[HK12] Shai Halevi and Yael Tauman Kalai. "Smooth Projective Hashing and Two-Message Oblivious Transfer". In: *Journal of Cryptology* 25.1 (Jan. 2012), pp. 158–193. DOI: 10.1007/s00145-010-9092-8.

[HW15] Pavel Hubacek and Daniel Wichs. "On the Communication Complexity of Secure Function Evaluation with Long Output". In: *ITCS 2015*. Ed. by Tim Roughgarden. ACM, Jan. 2015, pp. 163–172. DOI: 10.1145/2688073.2688105.

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. "Extending Oblivious Transfers Efficiently". In: *CRYPTO 2003*. Ed. by Dan Boneh. Vol. 2729. LNCS. Springer, Heidelberg, Aug. 2003, pp. 145–161. DOI: 10.1007/978-3-540-45146-4_9.

[IKO+11] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, Manoj Prabhakaran, and Amit Sahai. "Efficient Non-interactive Secure Computation". In: *EUROCRYPT 2011*. Ed. by Kenneth G. Paterson. Vol. 6632. LNCS. Springer, Heidelberg, May 2011, pp. 406–425. DOI: 10.1007/978-3-642-20465-4_23.

[IP07] Yuval Ishai and Anat Paskin. "Evaluating Branching Programs on Encrypted Data". In: *TCC 2007*. Ed. by Salil P. Vadhan. Vol. 4392. LNCS. Springer, Heidelberg, Feb. 2007, pp. 575–594. DOI: 10.1007/978-3-540-70936-7_31.

[IPS08] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. "Founding Cryptography on Oblivious Transfer - Efficiently". In: *CRYPTO 2008*. Ed. by David Wagner. Vol. 5157. LNCS. Springer, Heidelberg, Aug. 2008, pp. 572–591. DOI: 10.1007/978-3-540-85174-5_32.

[IR90] Russell Impagliazzo and Steven Rudich. "Limits on the Provable Consequences of One-way Permutations". In: *CRYPTO'88*. Ed. by Shafi Goldwasser. Vol. 403. LNCS. Springer, Heidelberg, Aug. 1990, pp. 8–26. DOI: 10.1007/0-387-34799-2_2.

[JKK14] Stanislaw Jarecki, Aggelos Kiayias, and Hugo Krawczyk. "Round-Optimal Password-Protected Secret Sharing and T-PAKE in the Password-Only Model". In: *ASIACRYPT 2014, Part II*. Ed. by Palash Sarkar and Tetsu Iwata. Vol. 8874. LNCS. Springer, Heidelberg, Dec. 2014, pp. 233–253. DOI: 10.1007/978-3-662-45608-8_13.

[JKKX16] Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. "Highly-efficient and composable password-protected secret sharing (or: How to protect your bitcoin wallet online)". In: *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2016, pp. 276–291.

[JKKX17] Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. "TOPPSS: Cost-Minimal Password-Protected Secret Sharing Based on Threshold OPRF". In: *ACNS 17*. Ed. by Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi. Vol. 10355. LNCS. Springer, Heidelberg, July 2017, pp. 39–58. DOI: 10.1007/978-3-319-61204-1_3.

[JWT] *JSON Web Tokens*. https://jwt.io/. Accessed on May 16, 2019.

[KBC97] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. *HMAC: Keyed-Hashing for Message Authentication*. IETF Internet Request for Comments 2104. Feb. 1997.

[Kerberos] *Kerberos: The Network Authentication Protocol*. https://web.mit.edu/kerberos/. Accessed on May 16, 2019.

[KK13] Vladimir Kolesnikov and Ranjit Kumaresan. "Improved OT Extension for Transferring Short Secrets". In: *CRYPTO 2013, Part II*. Ed. by Ran Canetti and Juan A. Garay. Vol. 8043. LNCS. Springer, Heidelberg, Aug. 2013, pp. 54–70. DOI: 10.1007/978-3-642-40084-1_4.

[KLW15] Venkata Koppula, Allison Bishop Lewko, and Brent Waters. "Indistinguishability Obfuscation for Turing Machines with Unbounded Memory". In: *47th ACM STOC*. Ed. by Rocco A. Servedio and Ronitt Rubinfeld. ACM Press, June 2015, pp. 419–428. DOI: 10.1145/2746539.2746614.

[KMTG05] Jonathan Katz, Philip D. MacKenzie, Gelareh Taban, and Virgil D. Gligor. "Two-Server Password-Only Authenticated Key Exchange". In: *ACNS 05*. Ed. by John Ioannidis, Angelos Keromytis, and Moti Yung. Vol. 3531. LNCS. Springer, Heidelberg, June 2005, pp. 1–16. DOI: 10.1007/11496137_1.

[KOY01] Jonathan Katz, Rafail Ostrovsky, and Moti Yung. "Efficient Password-Authenticated Key Exchange Using Human-Memorable Passwords". In: *EUROCRYPT 2001*. Ed. by Birgit Pfitzmann. Vol. 2045. LNCS. Springer, Heidelberg, May 2001, pp. 475–494. DOI: 10.1007/3-540-44987-6_29.

[KOY03] Jonathan Katz, Rafail Ostrovsky, and Moti Yung. "Forward Secrecy in Password-Only Key Exchange Protocols". In: *SCN 02*. Ed. by Stelvio Cimato, Clemente Galdi, and Giuseppe Persiano. Vol. 2576. LNCS. Springer, Heidelberg, Sept. 2003, pp. 29–44. DOI: 10.1007/3-540-36413-7_3.

[KV11] Jonathan Katz and Vinod Vaikuntanathan. "Round-Optimal Password-Based Authenticated Key Exchange". In: *TCC 2011*. Ed. by Yuval Ishai. Vol. 6597. LNCS. Springer, Heidelberg, Mar. 2011, pp. 293–310. DOI: 10.1007/978-3-642-19571-6_18.

[LNO13] Yehuda Lindell, Kobbi Nissim, and Claudio Orlandi. "Hiding the Input-Size in Secure Two-Party Computation". In: *ASIACRYPT 2013, Part II*. Ed. by Kazue Sako and Palash Sarkar. Vol. 8270. LNCS. Springer, Heidelberg, Dec. 2013, pp. 421–440. DOI: 10.1007/978-3-642-42045-0_22.

[LO13] Steve Lu and Rafail Ostrovsky. "How to Garble RAM Programs". In: *EUROCRYPT 2013*. Ed. by Thomas Johansson and Phong Q. Nguyen. Vol. 7881. LNCS. Springer, Heidelberg, May 2013, pp. 719–734. DOI: 10.1007/978-3-642-38348-9_42.

[LP09] Yehuda Lindell and Benny Pinkas. "A Proof of Security of Yao's Protocol for Two-Party Computation". In: *Journal of Cryptology* 22.2 (Apr. 2009), pp. 161–188. DOI: 10.1007/s00145-008-9036-8.

[LQR+19] Alex Lombardi, Willy Quach, Ron D Rothblum, Daniel Wichs, and David J Wu. "New constructions of reusable designated-verifier NIZKs". In: *CRYPTO 2019*. Ed. by Hovav Shacham and Alexandra Boldyreva. LNCS. Springer, Heidelberg, Aug. 2019.

[MPS+03] Keith M. Martin, Josef Pieprzyk, Reihaneh Safavi-Naini, Huaxiong Wang, and Peter R. Wild. "Threshold MACs". In: *ICISC 02*. Ed. by Pil Joong Lee and Chae Hoon Lim. Vol. 2587. LNCS. Springer, Heidelberg, Nov. 2003, pp. 237–252.

[MR01] Philip D. MacKenzie and Michael K. Reiter. "Networked Cryptographic Devices Resilient to Capture". In: *2001 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2001, pp. 12–25. DOI: 10.1109/SECPRI.2001.924284.

[MRK03] Silvio Micali, Michael O. Rabin, and Joe Kilian. "Zero-Knowledge Sets". In: *44th FOCS*. IEEE Computer Society Press, Oct. 2003, pp. 80–91. DOI: 10.1109/SFCS.2003.1238183.

[MSJ02] Philip D. MacKenzie, Thomas Shrimpton, and Markus Jakobsson. "Threshold Password-Authenticated Key Exchange". In: *CRYPTO 2002*. Ed. by Moti Yung. Vol. 2442. LNCS. Springer, Heidelberg, Aug. 2002, pp. 385–400. DOI: 10.1007/3-540-45708-9_25.

[NP01] Moni Naor and Benny Pinkas. "Efficient Oblivious Transfer Protocols". In: *12th SODA*. Ed. by S. Rao Kosaraju. ACM-SIAM, Jan. 2001, pp. 448–457.

[NPR99] Moni Naor, Benny Pinkas, and Omer Reingold. "Distributed Pseudo-random Functions and KDCs". In: *EUROCRYPT'99*. Ed. by Jacques Stern. Vol. 1592. LNCS. Springer, Heidelberg, May 1999, pp. 327–346. DOI: 10.1007/3-540-48910-X_23.

[OAuth] *The OAuth 2.0 Authorization Framework: Bearer Token Usage*. https://tools.ietf.org/html/rfc6750. Accessed on May 16, 2019.

[OpenID] *The OpenID Connect*. http://openid.net/connect/.

[OPP14] Rafail Ostrovsky, Anat Paskin-Cherniavsky, and Beni Paskin-Cherniavsky. "Maliciously Circuit-Private FHE". In: *CRYPTO 2014, Part I*. Ed. by Juan A. Garay and Rosario Gennaro. Vol. 8616. LNCS. Springer, Heidelberg, Aug. 2014, pp. 536–553. DOI: 10.1007/978-3-662-44371-2_30.

[OPWW15] Tatsuaki Okamoto, Krzysztof Pietrzak, Brent Waters, and Daniel Wichs. "New Realizations of Somewhere Statistically Binding Hashing and Positional Accumulators". In: *ASIACRYPT 2015, Part I*. Ed. by Tetsu Iwata and Jung Hee Cheon. Vol. 9452. LNCS. Springer, Heidelberg, Nov. 2015, pp. 121–145. DOI: 10.1007/978-3-662-48797-6_6.

[OS97] Rafail Ostrovsky and Victor Shoup. "Private Information Storage (Extended Abstract)". In: *29th ACM STOC*. ACM Press, May 1997, pp. 294–303. DOI: 10.1145/258533.258606.

[Ost90] Rafail Ostrovsky. "Efficient Computation on Oblivious RAMs". In: *22nd ACM STOC*. ACM Press, May 1990, pp. 514–523. DOI: 10.1145/100216.100289.

[Ost92] Rafail Ostrovsky. "Software Protection and Simulation On Oblivious RAMs". PhD thesis. Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1992.

[PHC] *Password Hashing Competition*. https://password-hashing.net/. Accessed on May 16, 2019.

[PS10] Kenneth G. Paterson and Douglas Stebila. "One-Time-Password-Authenticated Key Exchange". In: *ACISP 10*. Ed. by Ron Steinfeld and Philip Hawkes. Vol. 6168. LNCS. Springer, Heidelberg, July 2010, pp. 264–281.

[QWW18] Willy Quach, Hoeteck Wee, and Daniel Wichs. "Laconic Function Evaluation and Applications". In: *59th FOCS*. Ed. by Mikkel Thorup. IEEE Computer Society Press, Oct. 2018, pp. 859–870. DOI: 10.1109/FOCS.2018.00086.

[Rab81] Michael O. Rabin. *How To Exchange Secrets with Oblivious Transfer*. Harvard University, Cambridge, Massachusetts, USA, 1981.

[RB89] Tal Rabin and Michael Ben-Or. "Verifiable Secret Sharing and Multiparty Protocols with Honest Majority (Extended Abstract)". In: *21st ACM STOC*. ACM Press, May 1989, pp. 73–85. DOI: 10.1145/73007.73014.

[SAML] *SAML Toolkits*. https://developers.onelogin.com/saml. Accessed on May 16, 2019.

[Sho00] Victor Shoup. "Practical Threshold Signatures". In: *EUROCRYPT 2000*. Ed. by Bart Preneel. Vol. 1807. LNCS. Springer, Heidelberg, May 2000, pp. 207–220. DOI: 10.1007/3-540-45539-6_15.

[Vault] *Vault by Hashicorp*. https://www.vaultproject.io/docs/internals/token.html. Accessed on May 16, 2019.

[Vil12] Jorge Luis Villar. "Optimal Reductions of Some Decisional Problems to the Rank Problem". In: *ASIACRYPT 2012*. Ed. by Xiaoyun Wang and Kazue Sako. Vol. 7658. LNCS. Springer, Heidelberg, Dec. 2012, pp. 80–97. DOI: 10.1007/978-3-642-34961-4_7.

[WHC+14] Xiao Shaun Wang, Yan Huang, T.-H. Hubert Chan, abhi shelat, and Elaine Shi. "SCORAM: Oblivious RAM for Secure Computation". In: *ACM CCS 2014*. Ed. by Gail-Joon Ahn, Moti Yung, and Ninghui Li. ACM Press, Nov. 2014, pp. 191–202. DOI: 10.1145/2660267.2660365.

[WW15] Ding Wang and Ping Wang. "Offline Dictionary Attack on Password Authentication Schemes Using Smart Cards". In: *ISC 2013*. Ed. by Yvo Desmedt. Vol. 7807. LNCS. Springer, Heidelberg, Nov. 2015, pp. 221–237. DOI: 10.1007/978-3-319-27659-5_16.

[XS03] Shouhuai Xu and Ravi S. Sandhu. "Two Efficient and Provably Secure Schemes for Server-Assisted Threshold Signatures". In: *CT-RSA 2003*. Ed. by Marc Joye. Vol. 2612. LNCS. Springer, Heidelberg, Apr. 2003, pp. 355–372. DOI: 10.1007/3-540-36563-X_25.

[Yao82] Andrew Chi-Chih Yao. "Protocols for Secure Computations (Extended Abstract)". In: *23rd FOCS*. IEEE Computer Society Press, Nov. 1982, pp. 160–164. DOI: 10.1109/SFCS.1982.38.

[Yao86] Andrew Chi-Chih Yao. "How to Generate and Exchange Secrets (Extended Abstract)". In: *27th FOCS*. IEEE Computer Society Press, Oct. 1986, pp. 162–167. DOI: 10.1109/SFCS.1986.25.

[YHCL15] Xun Yi, Feng Hao, Liqun Chen, and Joseph K. Liu. "Practical Threshold Password-Authenticated Secret Sharing Protocol". In: *ESORICS 2015, Part I*. Ed. by Günther Pernul, Peter Y. A. Ryan, and Edgar R. Weippl. Vol. 9326. LNCS. Springer, Heidelberg, Sept. 2015, pp. 347–365. DOI: 10.1007/978-3-319-24174-6_18.