Abstract of "Cryptography for Efficiency: New Directions in Authenticated Data Structures" by Charalampos Papamanthou, Ph.D., Brown University, May 2011.

Cloud computing has emerged as an important new computational and storage medium and is increasingly being adopted both by companies and individuals as a means of reducing operational and maintenance costs. However, remotely-stored sensitive data may be lost or modified and third-party computations may not be performed correctly due to errors, opportunistic behavior, or malicious attacks. Thus, while the cloud is an attractive alternative to local trusted computational resources, users need integrity guarantees in order to fully adopt this new paradigm. Specifically, they need to be assured that uploaded data has not been altered and outsourced computations have been performed correctly.

Tackling the above problems requires the design of protocols that, on the one hand, are provably secure and at the same time remain highly efficient, otherwise the main purpose of adopting cloud computing, namely efficiency and scalability, is defeated. It is therefore essential that expertise in cryptography and efficient algorithmics be combined to achieve these goals.

This thesis studies techniques allowing the efficient verification of data integrity and computations correctness in such adversarial environments. Towards this end, several new authenticated data structures for fundamental algorithmics and computation problems, e.g., hash table queries and set operations, are proposed. The main novelty of this work lies in employing advanced *cryptography* such as lattices and bilinear maps, towards achieving high *efficiency*, departing from traditional hash-based primitives. As such, the proposed techniques lead to efficient solutions that introduce minimal asymptotic overhead and at the same time enable highly-desirable features such as optimal verification mechanisms and parallel authenticated data structures algorithms. The small asymptotic overhead does translate into significant practical savings, yielding efficient protocols and system prototypes.

Cryptography for Efficiency: New Directions in Authenticated Data Structures

by

Charalampos Papamanthou

B.Sc., Applied Informatics, University of Macedonia, 2003

M.Sc., Computer Science, University of Crete, 2005

M.Sc., Computer Science, Brown University, 2007

A dissertation submitted in partial fulfillment of the

requirements for the Degree of Doctor of Philosophy

in the Department of Computer Science at Brown University

Providence, Rhode Island

May 2011

This dissertation by Charalampos Papamanthou is accepted in its present form by the Department of Computer Science as satisfying the dissertation requirement for the degree of Doctor of Philosophy.

Date _____ _____

Roberto Tamassia, Director

Recommended to the Graduate Council

Date _____ _____

Michael T. Goodrich, Reader
University of California, Irvine

Date _____ _____

Anna Lysyanskaya, Reader

Date _____ _____

Franco P. Preparata, Reader

Approved by the Graduate Council

Date _____ _____

Peter M. Weber
Dean of the Graduate School

Vita

Charalampos Papamanthou was born in Trikala, Greece, 29 years ago. Right after graduation from high school, he began his college studies in Thessaloniki, Greece, receiving his bachelor's degree in Applied Informatics from the University of Macedonia in 2003. He then traveled south to pursue a master's degree in Computer Science at the beautiful island of Crete. There, and under the Mediterranean sun, he also did research at the Foundation for Research and Technology Hellas. Upon completion of his studies at the University of Crete in 2005, he decided to cross the Atlantic and move to Providence, Rhode Island, in order to attend Brown University for graduate school. At Brown, he received both his master's and doctoral degrees in Computer Science in 2007 and 2011 respectively. He was also the recipient of the Kanellakis and the van Dam fellowships. While at graduate school, he spent two summers at the West Coast, interning at Intel Research and Microsoft Research. His research interests are in computer security, applied cryptography and in the design and analysis of algorithms. Beginning summer 2011, he will be joining the University of California at Berkeley to work as a postdoctoral researcher at the Computer Science Division.

Preface

Cloud computing has emerged as an important new computational and storage medium and is increasingly being adopted both by companies and individuals as a means of reducing operational and maintenance costs. However, remotely-stored sensitive data may be lost or modified and third-party computations may not be performed correctly due to errors, opportunistic behavior, or malicious attacks. Thus, while the cloud is an attractive alternative to local trusted computational resources, users need integrity guarantees in order to fully adopt this new paradigm. Specifically, they need to be assured that uploaded data has not been altered and outsourced computations have been performed correctly.

Tackling the above problems requires the design of protocols that, on the one hand, are provably secure and at the same time remain highly efficient, otherwise the main purpose of adopting cloud computing, namely efficiency and scalability, is defeated. It is therefore essential that expertise in cryptography and efficient algorithmics be combined to achieve these goals.

This thesis studies techniques allowing the efficient verification of data integrity and computations correctness in such adversarial environments. Towards this end, several new authenticated data structures for fundamental algorithmics and computation problems, e.g., hash table queries and set operations, are proposed. The main novelty of this work lies in employing advanced *cryptography* such as lattices and bilinear maps, towards achieving high *efficiency*, departing from traditional hash-based primitives. As such, the proposed

techniques lead to efficient solutions that introduce minimal asymptotic overhead and at the same time enable highly-desirable features such as optimal verification mechanisms and parallel authenticated data structures algorithms. The small asymptotic overhead does translate into significant practical savings, yielding efficient protocols and system prototypes.

Acknowledgments

Many individuals contributed to the outcome of this beautiful educational journey at Brown University.

First and foremost, I deeply thank my thesis advisor, Roberto Tamassia, who guided me through the challenging path of graduate school. Roberto's vast experience in research, combined with his kindness, smile and sincerity, taught me how to produce high-quality work with a positive attitude, always being precise, objective and very self-critical. His efficient quest for perfection, his work ethic, as well as his constructive feedback were vital in shaping not only my research philosophy, but also my daily presence and interactions in an academic environment. Finally, Roberto's advice on personal matters and academics has been really invaluable and was always promptly and generously provided, whenever needed. I could not have hoped for a better advisor.

Second, I am grateful to Franco P. Preparata, with whom I closely collaborated during my first two years at Brown. Franco was the first faculty member I met as soon as I arrived in Providence, back in 2005. Having known Franco for six years now, I am still amazed by his seemingly endless knowledge of Computer Science, his high integrity, and his loyalty to his colleagues. I thank him for the so many technical and political discussions we had, his meticulously prepared lectures on parallel algorithms and computational biology, and for the provably correct advice he would always provide at the right time. Also, I would like to thank his wife, Rosa Maria, for inviting me multiple times for dinner at their place.

Admittedly these have been the most original and tasteful Italian dinners ever!

I would also like to thank the other members of my committee, Michael T. Goodrich and Anna Lysyanskaya. Michael has been a great collaborator, always encouraging new ideas and a diverse research agenda. He provided excellent feedback on the final text of this thesis. Anna taught me foundations of cryptography, through an engaging introductory class and through the crypto reading group. Her presence in the department and my interactions with her greatly influenced the research path of this dissertation. Also, many thanks to Nikos Triandopoulos, who, apart from a close friend, has been a reliable colleague, always eager to carefully listen to all my ideas and concerns. Many results in this dissertation have been the outcome of a great deal of fruitful discussions and long technical meetings with him. Finally, I would like to thank Alptekin Küpçü, C. Chris Erway, Bernardo Palazzi, Alexander Heitzmann, Olya Ohrimenko and Danfeng Yao for the work we did together on topics related to this thesis, as well as Petros Maniatis and Seny Kamara for being my internship mentors at Intel Research and Microsoft Research respectively.

The Brown CS faculty, and in particular the professors I interacted with mostly, namely, John Savage, Claire Mathieu, Philip Klein, Tom Doeppner, Pascal Van Hentenryck, Rodrigo Fonseca and Eli Upfal, have been extraordinary. Their persistent dedication to high-quality research and teaching nurtured an inspiring and challenging environment for every graduate student in the department. Also, everyday life in the CIT would not have come easy had it not been for the Brown CS Astaff and Tstaff. Thank you Janet, Lauren, Jane, Genie, Dorinda, Max, Phirum and Jeff!

Back in Greece, my professors and friends at UoM and UoC provided just the right academic environment. My undergraduate mentors, Konstantinos Paparrizos and Nikolaos Samaras, introduced me to research. My advisor at the University of Crete, Ioannis (Yanni) G. Tollis, helped me transform from an excited undergraduate student to an ambitious and focused graduate student. Working with Yanni was a great experience, and I am grateful to him for inspiring me to do research on exciting problems. Finally, many thanks to my friends

and colleagues from my university years in Greece, Dimitris Xinidis, Manos Papaggelis, Adam Arvelakis, Alexandros Stamatakis, Pavlos Pavlidis, Christos Sgaras and Kostas Tzouvaras, for keeping in touch and for sharing their exciting stories with me while I was far away.

Life in Providence would not have been as fun without the good times spent with the following people: Socrate, Dimitri, Yanni, Ari, Pari, Misha, Foteini, Yorgo, Maria, Anastasia, Aggeliki, Olga, Panagioti, Katerina, Basili, Saki, Aparna, Menia, Sophocle, Wenjin, Radu and Doria, thank you all guys! The Papavasiliou family in Attleboro made sure America felt like home, and of course, a big thank you belongs to my childhood buddies Pete, Achilleas, Ilias and Manos, as well as to all the members of my extended family, for being a continuous source of encouragement. Finally, many thanks to Vasili, Petro and Michali, for an amazing summer in Seattle (and for taking care of me while on crutches).

The research performed in this thesis was supported by the Center for Geometric Computing, the Plastech Professorship of Computer Science, the van Dam fellowship and the Kanellakis fellowship at Brown University, the National Science Foundation, NetApp and IAM Technology. It has been an honor for me to be a Kanellakis fellow and I would like to wholeheartedly thank the Kanellakis family for their generous support.

Last but not least, I am grateful to my parents Yianni and Gioula and to my brother Christo, for their unconditional love and support during all these years, as well as for everything they sacrificed for my upbringing and education. Finally, to honor her memory, I dedicate this thesis to my late grandmother Artemisia, for the values she imparted to me about life with her simple, but deep in meaning, sayings.

Στη μνήμη της γιαγιάς μου, Αρτεμισίας Χριστοδούλου.

To the memory of my grandmother, Artemisia Christodoulou.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

During the last few years, *cloud computing* has emerged as an important new computational and storage medium [55]. In fact, remote data storage (e.g., Amazon S3) and outsourcing of computation (e.g., Google docs) have become a major everyday phenomenon. An increasingly large number of companies and individuals have adopted cloud computing as a means of reducing operational and maintenance costs.

However, the cloud is not a panacea. Quoting from an article that was published in 2009 at techcrunch.com [2],

> "*...T-Mobile and Danger, the Microsoft-owned subsidiary that makes the Sidekick, has just announced that they've likely lost all user data that was being stored on Microsoft's servers due to a server failure...*".

Therefore, and beyond the control of its owner, remotely stored sensitive data may be lost, modified or accessed by unauthorized entities. Additionally, third-party (i.e., cloud) computations may not be performed correctly, due to errors, opportunistic behavior or malicious attacks. All these cases imply that, while the cloud is an attractive alternative to local trusted computational resources, there is a series of security threats that needs to be addressed in order for this paradigm to be fully adopted by the users. For example, *integrity* and *privacy* guarantees are highly needed: Specifically, users need to be assured that remote

data and computations have not been altered and no cloud data has leaked.

Tackling the above problems requires the design of protocols and the development of prototypes that, on the one hand, are *provably secure* and at the same time remain *highly efficient*, otherwise the main purpose of adopting cloud computing, i.e., efficiency and scalability, is defeated. In other words, the provable security added to a cloud service should not lead to major performance penalties and the induced overhead should be negligible compared to the actual computational resources needed by the application when executing in an insecure environment. It is essential that expertise in cryptography and efficient algorithmics be combined to achieve these goals.

This thesis addresses the first aspect of cloud security, namely the verification of *cloud data integrity* and *cloud computations correctness*. It is an extended and formal study of *authenticated data structures* [105], which comprise systematic and efficient methods for cryptographically checking the integrity of *dynamic* structured data—stored at adversarial environments—and queries executed on it. Namely, given that a data structure is stored at some untrusted entity and some computation can be performed over it by this untrusted entity (e.g., output *yes* if x is contained in a dictionary D or output the shortest path from node v to node u in a graph G), an authenticated data structure provides the cryptographic machinery and the algorithms for deciding, without having access to the data structure itself but only to some constant reliable (and possibly secret) space, whether the returned answer is correct or not.

First, an authenticated data structure should be *secure*: A computationally-bounded adversary should not be able to produce a valid proof for a false answer, under a well-accepted computational assumption. Second, it should be *efficient*: Its algorithms should have low complexity, ideally not adding too much overhead. Successfully combining both these goals comprises a challenging task, substantially depending on the underlying cryptography. Under this premise, the main novelty of this thesis lies in employing

advanced cryptography for constructing highly efficient authenticated data structures

departing in this way from widely employed *hash-based* constructions (e.g., Merkle trees [77]) which traditionally use collision-resistant hash functions (e.g., SHA-2 [85]) as a black box and enforce in this way certain complexity lower bounds [106]. Specifically, the coupling of certain cryptographic primitives, such as accumulators [13], lattices [3] and bilinear maps [60], with suitable data structuring and algorithms techniques, such as hash tables [29] and search trees [47], is explored and exploited, leading to efficient solutions that introduce minimal (and sometimes zero) asymptotic overhead. The small asymptotic overhead *does* translate into significant *practical* savings, yielding protocols that compare favorably in practice with existing work. The security of our constructions is based on computational assumptions widely established and accepted by the cryptography community.

1.1 Thesis motivation

The problem of efficiently verifying the integrity of structured data stored at untrusted resources has been an active research area for almost two decades, beginning with the seminal paper of Merkle [77], where the well-known—and widely used in practice since then—*hash trees* were introduced. Being an alternative to plain *digital signatures*, hash trees comprise an efficient way to provide integrity proofs for structured data (e.g., an array or a dictionary) stored at computationally-bounded adversarial sites—certain costs, compared to digital signatures techniques, decrease from linear to logarithmic. The problem was later formalized by Blum et al. [15] who introduced *memory checking*, i.e., mechanisms for reliably reading and writing memory cells, when the memory is not to be trusted. Later on, and after Naor and Nissim dynamized Merkle's solution [81], it became clear that verification mechanisms for more complicated structures (e.g., supporting dynamic operations) and more general query types (e.g., a verifying the output of an algorithm) were needed. Both these verification tasks could be achieved via memory checking techniques, since every data update or computation may be viewed as writing and reading bits from memory respectively. However, and as it

usually happens when directly employing fundamental primitives (e.g., zero-knowledge [45], oblivious RAM [86]) in cryptography for solving more complicated problems, *inefficiency* is a major issue. This motivated the study of *authenticated data structures* [105], a model where untrusted parties answer queries on a dynamic data structure providing a proof of validity of each answer to the user, in an efficient way.

So far in the literature, the *vast* majority of algorithms and techniques for authenticated data structures (e.g., [4, 6, 48, 53, 72, 74, 75, 77, 81, 88, 104, 112]) have traditionally relied on cryptographic hashing. In particular, *collision-resistance*, a fundamental property required for data integrity, is achieved through the use of black box generic functions, such as SHA-2 (these functions are believed to be collision-resistant in practice but there exists no formal proof for this). However, this black box property imposes certain complexity limitations through existing lower bounds [106]. For instance, for a dynamic dictionary of size n, $\Omega(\log n)$ proof complexities are inevitable, since the internal function of these primitives is not exploitable in any meaningful way. Aiming at more efficient solutions, this thesis studies the effect of *cryptography* on the *efficiency* of various authenticated data structures, by exploring certain algebraic properties of advanced cryptographic primitives that traditional black box generic hash functions lack. Combined with suitable data structures and algorithms, these properties comprise a deciding factor in deriving asymptotically better (even optimal) authenticated data structures, resulting in more scalable protocols and applications.

1.2 Thesis outline

The thesis outline is as follows. We begin with Chapter 2 where we present basic definitions that we extensively use in the thesis, such as the *data structure scheme* definition (Definition 2.2) and the *authenticated data structure scheme* definition (Definition 2.3), along with its *correctness* and *security* definitions (Definition 2.5 and 2.4 respectively). Moreover in Chapter 2 we show how, given any authenticated data structure scheme as a black box, we

can derive a *three-party* authenticated data structures protocol (Theorem 2.1) or a *two-party* authenticated data structures protocol (Theorem 2.2), both traditionally used to describe authenticated data structure solutions in the literature (e.g., see [75] for a three-party protocol and [92] for a two-party protocol). This black box approach not only eases the presentation of our results in subsequent chapters of the thesis but also helps us avoid repeating notions related to protocols in each chapter. Each of the remaining chapters of the thesis (Chapters 3 through 6) comprises a study of the application of a certain cryptographic primitive for solving a specific authenticated data structure problem, efficiently:

In Chapter 3, we design authenticated data structures for set membership queries on hash tables, using cryptographic primitives called *accumulators* [23, 83] and applying them in a novel hierarchical way over the stored data. We provide the first construction for authenticating a hash table with *constant query* cost and *sublinear update* cost, strictly improving upon previous methods, addressing and answering an open problem posed in [81]. The algebraic property we take advantage of in this construction is the *commutativity* of the RSA exponentiation function, which enables fast updates of cryptographic digests whenever partial information included in the digest changes, offering *incrementality at no cost* and at the same time allowing for succinct, constant-size proofs—notice that not all functions achieving incrementality at no cost [11] offer efficient proof complexity. The main results of this chapter are given in Theorems 3.2 and 3.4. A preliminary version of this chapter appears in [90].

In Chapter 4, we initially design a new authenticated data structure for a *dynamic table* with n entries. We present the first dynamic authenticated table that is *update-optimal* and is based on lattices, a mathematical object that found many applications in cryptography during the last decade. In particular, the update complexity of the authenticated table we design is $O(1)$, improving in this way the "a priori" $O(\log n)$ update bounds of previous constructions, such as the Merkle tree. To achieve this result, we establish and exploit a property that we call *repeated linearity* of lattice-based hash functions. We secondly observe

that the repeated linearity of the used lattice-based cryptographic primitive lends itself to a natural notion of parallelism: As such, we describe *parallel* versions of our authenticated data structure algorithms, yielding the first parallel *online memory checker* [15] with $O(1)$ query complexity using $O(\log n)$ checkers in the CREW model and without using a secret key setting, i.e., there is only need for small *reliable* but not *secret* memory. Theorem 4.4 describes the basic (parallel) authenticated data structure and Theorem 4.6 gives the application of the presented lattice-based authenticated table in parallel online memory checking. A preliminary version of this chapter appears in [93].

In Chapter 5, we study the problem of verifying outsourced set operations over a dynamic collection of sets. Based on the convenient use of the *bilinear-map* primitive, which proved to be a very useful tool in cryptography after its first appearance in the literature within a cryptographic context [60], we are able to construct the first *operation-sensitive* scheme for verifying set operations, such as *union* and *intersection* (see Theorem 5.1): Operation-sensitivity is a strong property that enables us to achieve verification costs (proof and verification complexity) proportional to the size of the answer, and not to the time taken to produce the answer, a property that could not be achieved otherwise (e.g., with traditional hash-based techniques), and is obviously highly desirable. In this chapter, we also address applications of our techniques for verification of keyword searches on outsourced document collections (e.g., inverted-index queries) and queries in outsourced databases (e.g., equi-join queries). Since set intersection is heavily used in these applications, we obtain new authenticated data structures that compare favorably to existing approaches. This chapter closes an open problem, that of operation-sensitivity of set operations, posed in [33]. A preliminary version of this chapter appears in [94].

In Chapter 6, we observe that no optimal authenticated data structure (i.e., an authenticated data structure that adds no extra asymptotic overhead to the respective plain data

structure) is known to date[1]. However, by assuming the existence of *multilinear form* generators [91], a cryptographic primitive proposed by Silverberg and Boneh in 2003 [19], the construction of which, however, remains an open problem, we introduce the first optimal authenticated dictionary data structure. The presented authenticated dictionary, described in Theorem 6.1, enjoys proofs of constant size, i.e., asymptotically equal to the size of the answer. We close this chapter with Theorem 6.2, showing a reduction connecting the existence of optimal authenticated dictionaries with the existence of multilinear form generators. A preliminary version of this chapter appears in [91].

We conclude in Chapter 7 by commenting on some open problems and future work.

[1]Note that the lattice-based authenticated structure of Chapter 4 is only update-optimal whereas the authenticated data structure of Chapter 5 achieves optimal verification costs only.

Chapter 2

Preliminaries and related work

This chapter presents preliminary definitions and results that we are using in the rest of the thesis as well as an extended study of authenticated data structures related work.

2.1 Definitions

In the remainder of the thesis, we denote with $k \in \mathbb{N}$ the *security parameter*. We begin with the definition of a *negligible function*, extensively used to express our security arguments:

Definition 2.1 (Negligible function) *Let $f : \mathbb{N} \to \mathbb{R}$. We say that $f(k)$ is $\mathsf{neg}(k)$ iff for any nonzero polynomial $p(k)$ there exits N such that for all $k > N$ it is $f(k) < 1/p(k)$.*

Typical examples of functions that are $\mathsf{neg}(k)$ are the functions $\frac{1}{2^k}$ and $\frac{\mathsf{poly}(k)}{2^k}$.

2.1.1 Data structures and authenticated data structures

To formally describe our authenticated data structures solutions in Chapters 3, 4, 5 and 6, we give definitions for a *data structure scheme* and the respective *authenticated data structure scheme*. Similar definitions have appeared in the work of Tamassia and Triandopoulos [107]. To avoid unnecessary complications, we do not define an abstract *data type* and we instead

define a *data structure* scheme directly. We use the notation

$$\{O_1, O_2, \ldots, O_o\} \leftarrow \mathsf{alg}(I_1, I_2, \ldots, I_i)\,,$$

to denote that algorithm alg has i inputs $I_1, I_2, \ldots, I_i$ and o outputs $O_1, O_2, \ldots, O_o$. Whenever an input I or an output O appears as $(I)^*$ or $(O)^*$ (e.g., algorithms $\mathsf{update}()$ and $\mathsf{verify}()$ in Definition 2.3), this means that I or O are not *required* as inputs or outputs of the algorithm but might appear depending on the implemented scheme.

Definition 2.2 (Data structure scheme) *Let D be* any *data structure supporting a set of updates $\mathbf{U}$ and a set of queries $\mathbf{Q}$. Denote with D_h the state of the data structure D at time h, where $h \geq 0$ is an integer and D_0 is the initial state of the data structure D. A* data structure scheme $\mathcal{D}(\mathbf{U}, \mathbf{Q})$ *is a collection of the following three polynomial-time algorithms* $\{$**update**, **query**, **check**$\}$*:*

1. $D_{h+1} \leftarrow$ **update**(u, D_h)*: On input an update $u \in \mathbf{U}$ and the data structure D_h, this algorithm outputs the updated data structure D_{h+1};*

2. $\alpha(q) \leftarrow$ **query**(q, D_h)*: On input a query $q \in \mathbf{Q}$ and the data structure D_h, this algorithm outputs the answer $\alpha(q)$ to query q;*

3. $\{\mathsf{accept}, \mathsf{reject}\} \leftarrow$ **check**(q, α, D_h)*: On input a query $q \in \mathbf{Q}$, an answer α and the data structure D_h, this algorithm outputs* accept *if α is a correct answer for query q on data structure D_h. Else it outputs* reject*.*

For example, consider a data structure scheme for the *dictionary* data structure (see Section 6), implemented with a red-black tree [29]. The **query**() algorithm performs a binary search while the **update**() algorithm performs the relevant rotations needed for re-balancing the structure. We note here that in Definition 2.2, script letter $\mathcal{D}$ in the notation $\mathcal{D}(\mathbf{U}, \mathbf{Q})$ implies that $\mathcal{D}(\mathbf{U}, \mathbf{Q})$ is a data structure scheme for data structure D (non-script letter) which (the data structure D) supports the set updates $\mathbf{U}$ and the set of queries $\mathbf{Q}$.

Definition 2.3 (Authenticated data structure scheme) *Let $\mathcal{D}(\mathbf{U}, \mathbf{Q})$ be a data structure scheme defined by the collection of algorithms* {**update**, **query**, **check**}. *An* authenticated data structure scheme $\mathcal{A}$ *for the data structure scheme $\mathcal{D}(\mathbf{U}, \mathbf{Q})$ is a collection of the following six polynomial-time algorithms* {genkey, setup, update, refresh, query, verify}*:*

1. $\{\mathsf{sk}, \mathsf{pk}\} \leftarrow \mathsf{genkey}(1^k)$*: This algorithm outputs the secret key* sk *and the public key* pk*, given the security parameter k;*

2. $\{\mathsf{auth}(D_0), d_0\} \leftarrow \mathsf{setup}(D_0, \mathsf{sk}, \mathsf{pk})$*: This algorithm computes the authenticated data structure* $\mathsf{auth}(D_0)$ *and the respective digest d_0 of* $\mathsf{auth}(D_0)$*, given a data structure D_0, the secret key* sk *and the public key* pk*;*[1]

3. $\{D_{h+1}, (\mathsf{auth}(D_{h+1}))^*, d_{h+1}, \mathsf{upd}\} \leftarrow \mathsf{update}(u, D_h, (\mathsf{auth}(D_h))^*, d_h, \mathsf{sk}, \mathsf{pk})$*: This algorithm takes as input an update $u \in \mathbf{U}$, a data structure D_h,* possibly *an authenticated data structure* $\mathsf{auth}(D_h)$*, the digest d_h of* $\mathsf{auth}(D_h)$ *and both the secret and the public keys* sk *and* pk*. It outputs the data structure $D_{h+1} \leftarrow$* **update**(u, D_h)*,* possibly *the authenticated data structure* $\mathsf{auth}(D_{h+1})$*, the digest d_{h+1} of* $\mathsf{auth}(D_{h+1})$ *and some relative information* upd*;*[2]

4. $\{D_{h+1}, \mathsf{auth}(D_{h+1}), d_{h+1}\} \leftarrow \mathsf{refresh}(u, D_h, \mathsf{auth}(D_h), d_h, \mathsf{upd}, \mathsf{pk})$*: This algorithm takes as input an update $u \in \mathbf{U}$, a data structure D_h, an authenticated data structure* $\mathsf{auth}(D_h)$*, the digest d_h of* $\mathsf{auth}(D_h)$*, the information* upd *computed by algorithm* update() *and* only *the public key* pk*. It outputs the data structure $D_{h+1} \leftarrow$* **update**(u, D_h)*, the authenticated data structure* $\mathsf{auth}(D_{h+1})$ *and the digest d_{h+1} of* $\mathsf{auth}(D_{h+1})$*;*[3]

[1] The digest d_0 of the authenticated data structure $\mathsf{auth}(D_0)$ is a collision-resistant representation of D_0, e.g., the roothash of a Merkle tree [77]. It is *usually* of constant size.

[2] Note that this algorithm is only required to output the new digest d_{h+1} and the new data structure D_{h+1}. Outputting the new authenticated data structure $\mathsf{auth}(D_{h+1})$ is not a requirement of the algorithm—this will be important in improving the complexity of this algorithm is some schemes. Also, the secret key is required for execution.

[3] Note here that the secret key is *not* used for execution. However, for *correct* inputs, the output digest d_{h+1} is the same as in algorithm update().

5. $\{\Pi(q), \alpha(q)\} \leftarrow \mathsf{query}(q, D_h, \mathsf{auth}(D_h), \mathsf{pk})$*: On input a query* $q \in \mathbf{Q}$*, a data structure* D_h*, an authenticated data structure* $\mathsf{auth}(D_h)$ *and the public key* pk*, this algorithm returns the answer* $\alpha(q) \leftarrow \mathbf{query}(q, D_h)$ *to the query* q*, along with a respective proof* $\Pi(q)$*;*

6. $\{\mathsf{accept}, \mathsf{reject}\} \leftarrow \mathsf{verify}(q, \alpha, \Pi, d_h, (\mathsf{sk})^*, \mathsf{pk})$*: This algorithm takes as input a query* $q \in \mathbf{Q}$*, an answer* α*, a proof* Π*, the digest* d_h *of* $\mathsf{auth}(D_h)$*,* possibly *the secret key* sk *and the public key* pk *and outputs either* accept *or* reject*.*

There are two properties that an authenticated data structure scheme should satisfy, i.e., *correctness* and *security* (intuition follows from signature schemes definitions, e.g., see Camenisch and Lysyanskaya [24]): Roughly speaking, the correctness property requires that, for every query $q \in \mathbf{Q}$, if a proof $\Pi(q)$ is computed by algorithm $\mathsf{query}()$ (i.e., faithfully), then $\mathsf{verify}()$, on input $\Pi(q)$ and a *correct* answer $\alpha(q)$, should always accept, as long as the digest d is updated through algorithm $\mathsf{refresh}()$ (i.e., it is the correct one). The security property requires that a computationally-bounded adversary, i.e., an adversary that has access to polynomially-bounded resources (time and space) in the security parameter k, should not be able (except with negligible probability) to produce verifying proofs Π for *incorrect* answers α corresponding to queries $q \in \mathbf{Q}$ on an authenticated data structure $\mathsf{auth}(D)$ whose digest is updated through adversarially chosen *oracle* calls to algorithm $\mathsf{update}()$—this is why we require $\mathsf{update}()$ have access to the secret key.

Definition 2.4 (Correctness of authenticated data structure scheme) *Let* $\mathcal{D}(\mathbf{U}, \mathbf{Q})$ *be a data structure scheme defined by the collection of algorithms* $\{\mathbf{update}, \mathbf{query}, \mathbf{check}\}$ *and let also* $\mathcal{A}$ *be an authenticated data structure scheme for* $\mathcal{D}(\mathbf{U}, \mathbf{Q})$ *defined by the collection of algorithms* $\{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$*. We say that the authenticated data structure scheme* $\mathcal{A}$ *is correct if, for all* $k \in \mathbb{N}$*, for all* $\{\mathsf{sk}, \mathsf{pk}\}$ *output by algorithm* $\mathsf{genkey}()$*, for all* $D_h, \mathsf{auth}(D_h), d_h$ *output by one invocation of algorithm* $\mathsf{setup}()$ *followed by polynomially-many invocations of algorithm* $\mathsf{refresh}()$*, where* $h \geq 0$*, for all*

queries $q \in \mathbf{Q}$ *and for all* $\Pi(q), \alpha(q)$ *output by algorithm* $\mathsf{query}(q, D_h, \mathsf{auth}(D_h), \mathsf{pk})$*, with all but negligible probability, whenever algorithm* $\mathbf{check}(q, \alpha(q), D_h)$ *accepts, so does algorithm* $\mathsf{verify}(q, \Pi(q), \alpha(q), d_h, (\mathsf{sk})^*, \mathsf{pk})$*.*

Definition 2.5 (Security of authenticated data structure scheme) *Let* $\mathcal{D}(\mathbf{U}, \mathbf{Q})$ *be a data structure scheme defined by the collection of algorithms* $\{\mathbf{update}, \mathbf{query}, \mathbf{check}\}$*,* $\mathcal{A}$ *be an authenticated data structure scheme for* $\mathcal{D}(\mathbf{U}, \mathbf{Q})$ *defined by the collection of algorithms* $\{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$*,* k *be the security parameter and* $\{\mathsf{sk}, \mathsf{pk}\} \leftarrow \mathsf{genkey}(1^k)$*. Denote with* Adv *a polynomially-bounded adversary that is only given the public key* pk*. The adversary has unlimited access to all algorithms of* $\mathcal{A}$*, except for algorithms* $\mathsf{setup}()$*,* $\mathsf{update}()$ *and* possibly *algorithm* $\mathsf{verify}()$*, to which he has only oracle access. The adversary picks an initial state of the data structure* D_0 *and computes* $\mathsf{auth}(D_0), d_0$ *through oracle access to algorithm* $\mathsf{setup}()$*. Then, for* $i = 0, \ldots, h = \mathsf{poly}(k)$*,* Adv *issues an update* $u_i \in \mathbf{U}$ *in the data structure* D_i *and outputs* D_{i+1}*,* possibly *the authenticated data structure* $\mathsf{auth}(D_{i+1})$ *and* d_{i+1} *through oracle access to algorithm* $\mathsf{update}()$*. Finally the adversary enters the attack stage where he picks an index* $0 \leq t \leq h+1$*, a query* $q \in \mathbf{Q}$*, an answer* α *and a proof* Π*. We say that the authenticated data structure scheme* $\mathcal{A}$ *is secure if for all* $k \in \mathbb{N}$*, for all* $\{\mathsf{sk}, \mathsf{pk}\}$ *output by algorithm* $\mathsf{genkey}()$*, and for all polynomially-bounded adversaries* Adv *it is*

$$\Pr\left[\begin{array}{ll} \{q, \Pi, \alpha, t\} \leftarrow \mathsf{Adv}(1^k, \mathsf{pk}); & \mathsf{accept} \leftarrow \mathsf{verify}(q, \alpha, \Pi, d_t, (\mathsf{sk})^*, \mathsf{pk}); \\ & \mathsf{reject} \leftarrow \mathbf{check}(q, \alpha, D_t). \end{array}\right] \leq \nu(k)\,,$$

where $\nu(k)$ *is* $\mathsf{neg}(k)$*.*

2.1.2 Complexity model

To explicitly measure complexity of various algorithms with respect to the number of primitive cryptographic operations, without considering the dependency on the security parameter, we adopt the complexity model used in memory checking [15, 35], which has been only implicitly used in the literature of authenticated data structures:

Definition 2.6 (Access complexity) *The* access complexity *of an algorithm is defined as the* number of memory accesses *this algorithm performs on the (authenticated) data structure stored in an indexed memory of n cells, in order for the algorithm to complete its execution.*

Here, "access complexity" is used instead of "query complexity" used in memory checking [15, 35] to avoid ambiguity when referring to algorithm $\mathsf{query}()$ of the authenticated data structure scheme. We also require that each memory cell can store up to $O(\mathsf{poly}(\log n))$ bits, a word size used in Blum's original memory checking work [15] but also in subsequent work [35]. For example, a Merkle tree [77] has $O(\log n)$ update access complexity since the update algorithm needs to read and write $O(\log n)$ memory cells of the authenticated data structure, each cell storing exactly one hash value.

Definition 2.7 (Group complexity) *The* group complexity *of a data collection (e.g., proof group complexity or authenticated data structure group complexity) is defined as the number of elementary data objects (e.g., hash values or elements in $\mathbb{Z}_p$) contained in that object.*

Whenever it is clear from the context, we omit the terms "access" and "group". Also, concerning the above definitions, we note that since the size of the problem n, in a cryptographic setting, has to be polynomially-bounded by the security parameter k, i.e., $n = o(2^k)$, the $O(.)$ notation appearing in the following chapters expresses asymptotic results for values of $n \to 2^k$, and not for values of $n \to \infty$, as is mathematically implied by the $O(.)$ notation [29].

Second, we observe that access complexity captures the notion of "time" and group complexity captures the notion of "space". However, access and group complexities are in principle smaller than time and space complexities. This is because time and space complexities are counting number of bits and are always functions of the security parameter k. Being in a cryptographic setting however, the security parameter is always $\Omega(\log n)$ and therefore time and space complexities are always $\Omega(\log n)$—however, access and group complexities can be $O(1)$.

2.1.3 Optimality and public verifiability

We now give the definition of an *optimal* authenticated data structure scheme. Intuitively, an optimal authenticated data structure scheme $\mathcal{A}$ for a data structure scheme $\mathcal{D}(\mathbf{U}, \mathbf{Q})$ should not add *any* asymptotic overhead to the complexity of the algorithms of the data structure scheme $\mathcal{D}(\mathbf{U}, \mathbf{Q})$:

Definition 2.8 (Optimal authenticated data structure scheme) *Suppose $\mathcal{D}(\mathbf{U}, \mathbf{Q})$ is a data structure scheme defined by the collection of algorithms* {**update**, **query**, **check**} *and $\mathcal{A}$ is a correct and secure authenticated data structure scheme for $\mathcal{D}(\mathbf{U}, \mathbf{Q})$ defined by the collection of algorithms* {genkey, setup, update, refresh, query, verify}*. The authenticated data structure scheme $\mathcal{A}$ is* optimal *if all of the following are true:*

1. *(Space optimality) For all integers $h \geq 0$, the group complexity of the authenticated data structure* $\mathsf{auth}(D_h)$ *is no more than the group complexity of the data structure D_h, i.e.,*

$$|\mathsf{auth}(D_h)| = O(|D_h|)\,;$$

2. *(Update optimality) For all updates $u \in \mathbf{U}$, the sum of the access complexity of* update() *plus the group complexity of* upd *(output by* update()*) plus the access complexity of* refresh() *is no more than the access complexity of* **update**()*, i.e.,*

$$|\mathsf{update}()| + |\mathsf{upd}| + |\mathsf{refresh}()| = O(|\mathbf{update}()|)\,;$$

3. *(Query optimality) For all queries $q \in \mathbf{Q}$, the access complexity of* query() *is no more than the access complexity of* **query**()*, i.e.,*

$$|\mathsf{query}()| = O(|\mathbf{query}()|)\,;$$

4. *(Proof and verification optimality) Let $\alpha(q)$ and $\Pi(q)$ be the answer and the proof output by* query()*, on input a query $q \in \mathbf{Q}$. Then for all queries $q \in \mathbf{Q}$:*

- *The group complexity of the proof* $\Pi(q)$ *is no more than the group complexity of the query* q *plus the group complexity of the answer* $\alpha(q)$*, i.e.,*

$$|\Pi(q)| = O(|q| + |\alpha(q)|)\,;$$

- *The access complexity of* $\mathsf{verify}()$ *is no more than the group complexity of the query* q *plus the group complexity of the answer* $\alpha(q)$*, i.e.,*

$$|\mathsf{verify}()| = O(|q| + |\alpha(q)|)\,.$$

We note here that constructing an optimal authenticated data structure scheme appears to be a difficult task. In Chapter 5 we construct an authenticated data structure scheme that is *almost* optimal (update costs are increased by a logarithmic factor), whereas in Chapter 6 we construct an optimal authenticated data structure scheme for a dictionary data structure that is however based on a cryptographic primitive that is not known to exist yet.

Definition 2.9 (Publicly-verifiable authenticated data structure scheme) *Suppose* $\mathcal{D}(\mathbf{U}, \mathbf{Q})$ *is a data structure scheme and* $\mathcal{A}$ *is a authenticated data structure scheme for* $\mathcal{D}(\mathbf{U}, \mathbf{Q})$ *defined by the collection of algorithms* $\{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$*. Let also* k *be the security parameter and* $\{\mathsf{sk}, \mathsf{pk}\} \leftarrow \mathsf{genkey}(1^k)$*. The authenticated data structure scheme* $\mathcal{A}$ *is* publicly-verifiable *if algorithm* $\mathsf{verify}()$ does not *require the secret key* sk *as an input.*

2.2 Protocols and applications

Let $\mathcal{A}$ be a (publicly-verifiable) authenticated data structure scheme for a data structure scheme $\mathcal{D}(\mathbf{U}, \mathbf{Q})$ and let also $\mathcal{Q}$ be a correct and secure public-key signature scheme (e.g., see Camenisch and Lysyanskaya [24]). We now describe how an authenticated data structure scheme $\mathcal{A}$ may be used in various protocols, such as a three-party authenticated data

structures protocol (e.g., see Tamassia [105] and Martel et al. [75]) or a two-party authenticated data structures protocol (e.g., see Papamanthou and Tamassia [92]). For the protocols description we use the following notation:

- $[\mathcal{R}]_t$: program. Program program is *executed* by party $\mathcal{R}$ at time t;
- $[\mathcal{R} \rightarrow \mathcal{G}]_t$: data. Data data is *sent* from party $\mathcal{R}$ to party $\mathcal{G}$ at time t.

2.2.1 Three-party authenticated data structures protocol

A typical setting where an authenticated data structure scheme can be employed involves three participating entities, usually referred to as *three-party* model [105] (see Figure 2.1): A trusted party called *source*, owns a data structure, that is replicated along with some cryptographic information to one or more untrusted parties, called *servers*. *Clients* issue data structure queries to the servers and wish to *publicly* verify the answers received by the servers, based only on the trust they have in the source. This trust is conveyed through a time-stamped signature on the *digest* of the data structure (a collision resistant succinct representation of the data structure, e.g., the roothash of a Merkle tree—see Definition 2.3), that is made available by the source. During an update, the source needs just to compute

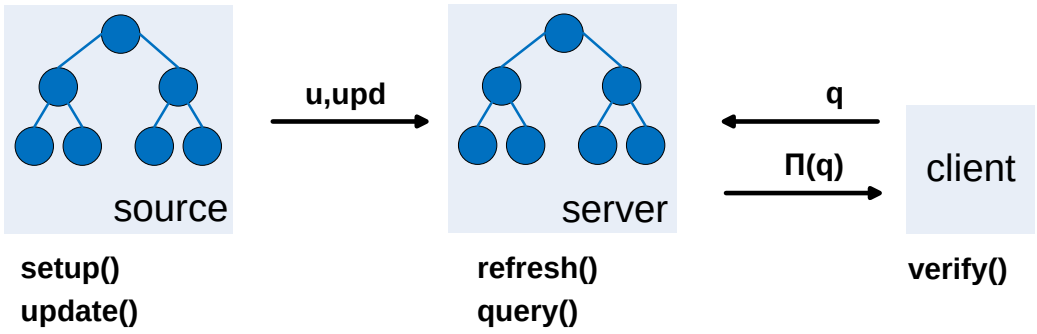

Figure 2.1: The three-party authenticated data structures protocol. During the update phase, the source sends an update $u \in \mathbf{U}$ to the server along with the respective update information upd output by update(). During the query phase, the client sends a query $q \in \mathbf{Q}$ to the server and the server runs algorithm query() to output the proof $\Pi(q)$ for the respective answer.

the new digest, whereas the server needs to update the authenticated data structure as a

whole. We describe this protocol formally below:

Protocol 2.1 *A three-party authenticated data structures protocol involves three participating entities: A* trusted *source,* $\mathcal{T}$*, that has access both to the public and the secret keys, an* untrusted *server,* $\mathcal{S}$*, that has access only to the public keys, and a client,* $\mathcal{C}$*, that has access only to the public keys. Let* k *be the security parameter,* $\mathcal{A} = \{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$ *be a correct and secure publicly-verifiable authenticated data structure scheme for a data structure scheme* $\mathcal{D}(\mathbf{U}, \mathbf{Q})$ *and* $\mathcal{Q} = \{\mathcal{GENKEY}, \mathcal{SIGN}, \mathcal{VERIFY}\}$ *be a correct and secure public-key signature scheme. The protocol has four phases:*

1. Setup phase*: Source* $\mathcal{T}$ *owns the data structure* D *(described by data structure scheme* $\mathcal{D}(\mathbf{U}, \mathbf{Q})$*) at time* t_0*. The setup phase consists of the following steps:*

   (a) $[\mathcal{T}]_{t_0}$*:* $\{\mathcal{SK}, \mathcal{PK}\} \leftarrow \mathcal{GENKEY}(1^k)$*; (*signature scheme keys generation*)*

   (b) $[\mathcal{T}]_{t_0}$*:* $\{\mathsf{sk}, \mathsf{pk}\} \leftarrow \mathsf{genkey}(1^k)$*; (*authenticated data structure scheme keys generation*)*

   (c) $[\mathcal{T}]_{t_0}$*:* $\{\mathsf{auth}(D), d\} \leftarrow \mathsf{setup}(D, \mathsf{sk}, \mathsf{pk})$*; (*computing the authenticated data structure*)*

   (d) $[\mathcal{T}]_{t_0}$*: Store* $D, \mathsf{auth}(D), d$*; (source stores necessary information)*

   (e) $[\mathcal{T} \rightarrow \mathcal{S}]_{t_0}$*:* $D, \mathsf{auth}(D), d$*; (*outsourcing*)*

   (f) $[\mathcal{S}]_{t_0}$*: Store* $D, \mathsf{auth}(D), d$*. (server stores necessary information)*

2. Update phase*: Let* $u \in \mathbf{U}$ *be an update issued by source* $\mathcal{T}$ *on data structure* D *at time* t_τ*. Let* D' *be the updated data structure. The update phase consists of the the following steps:*

   (a) $[\mathcal{T}]_{t_\tau}$*:* $\{D', (\mathsf{auth}(D'))^*, d', \mathsf{upd}\} \leftarrow \mathsf{update}(u, D, (\mathsf{auth}(D))^*, d, \mathsf{sk}, \mathsf{pk})$*;* $D = D'$*,* $(\mathsf{auth}(D))^* = (\mathsf{auth}(D'))^*$*,* $d = d'$*; (*new digest generation*)*

   (b) $[\mathcal{T} \rightarrow \mathcal{S}]_{t_\tau}$*:* u*,* upd*; (*sending relative update information*)*

(c) $[\mathcal{S}]_{t_\tau}$: $\{D', \mathsf{auth}(D'), d'\} \leftarrow \mathsf{refresh}(u, D, \mathsf{auth}(D), d, \mathsf{upd}, \mathsf{pk})$; $D = D'$, $\mathsf{auth}(D) = \mathsf{auth}(D')$, $d = d'$. (authenticated data structure update at the server)

3. Periodical signing: *Let $t_\kappa = \kappa\Delta t$ for $\kappa = 0, 1, 2, \ldots$ be certain timestamps in time (with time difference Δt), where t_0 is defined as the time of algorithm* setup() *execution (see Item 1c). The periodical signing is performed independently from all the other phases:*

(a) $[\mathcal{T}]_{t_\kappa}$ *for* $\kappa = 0, 1, 2, \ldots$: $\mathsf{sgn}_\kappa \leftarrow \mathcal{SIGN}(\mathcal{SK}, d||t_\kappa)$; (signature computation on most recent digest d)

(b) $[\mathcal{T} \rightarrow \mathcal{S}]_{t_\kappa}$ *for* $\kappa = 0, 1, 2, \ldots$: sgn_κ, t_κ; (sending signature and timestamp)

(c) $[\mathcal{S}]_{t_\kappa}$ *for* $\kappa = 0, 1, 2, \ldots$: $\mathsf{sgn} = \mathsf{sgn}_\kappa$ *and* $t = t_\kappa$. (storing most recent signature and timestamp)

4. Query phase: *The query phase proceeds as follows, between the client $\mathcal{C}$ and the server $\mathcal{S}$. Let T be the time of the query $q \in \mathbf{Q}$:*

(a) $[\mathcal{C} \rightarrow \mathcal{S}]_T$: q; (sending the query)

(b) $[\mathcal{S}]_T$: $\{\Pi(q), \alpha(q)\} \leftarrow \mathsf{query}(q, D, \mathsf{auth}(D), \mathsf{pk})$; (answer and proof computation)

(c) $[\mathcal{C} \leftarrow \mathcal{S}]_T$: $\alpha(q), \Pi(q), d, t, \mathsf{sgn}$; (sending answer, proof, most recent digest, most recent timestamp and signature on the last two by the source)

(d) $[\mathcal{C}]_T$: *Output* ACCEPT *if and only if:* (verification of answer)

i. $T - t < \Delta t$; (t is the most recent timestamp)

ii. $\mathcal{ACCEPT} \leftarrow \mathcal{VERIFY}(\mathsf{sgn}, d||t, \mathcal{PK})$; ($d$ is the most recent digest);

iii. $\mathsf{accept} \leftarrow \mathsf{verify}(q, \alpha(q), \Pi(q), d, \mathsf{pk})$.

else output REJECT.

We can now state the main theorem of this section:

Theorem 2.1 *Let* $\mathcal{A} = \{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$ *be a correct and secure publicly-verifiable authenticated data structure scheme for a data structure scheme* $\mathcal{D}(\mathbf{U}, \mathbf{Q})$

and k be the security parameter. Let $g(n)$, $s(n)$, $u(n)$, $r(n)$, $q(n)$ and $v(n)$ be the access *complexities of algorithms* genkey, setup, update, refresh, query, verify *respectively. Let also $i(n)$, $p(n)$ and $\alpha(n)$ be the* group *complexity of the information* upd *(output by algorithm* update()*), of the proof $\Pi(q)$ (output by algorithm* query()*) and of the answer $\alpha(q)$ (output by algorithm* query()*) respectively and $f(n)$ be the* group *complexity of the authenticated data structure* auth(D) *(output by algorithm* setup()*). Then there exists a three-party authenticated data structures protocol involving a trusted source $\mathcal{T}$, an untrusted server $\mathcal{S}$ and a client $\mathcal{C}$ for verifying queries $q \in \mathbf{Q}$ and at the same time supporting updates $u \in \mathbf{U}$ such that:*

1. *The setup at the source $\mathcal{T}$ has $O(s(n))$ access complexity;*
2. *The update at the source $\mathcal{T}$ has $O(u(n))$ access complexity;*
3. *The space needed at the source $\mathcal{T}$ has $O(f(n))$ group complexity;*
4. *The communication between the source $\mathcal{T}$ and the server $\mathcal{S}$ has $O(i(n))$ group complexity;*
5. *The update at the server $\mathcal{S}$ has $O(r(n))$ access complexity;*
6. *The query at the server $\mathcal{S}$ has $O(q(n))$ access complexity;*
7. *The space needed at the server $\mathcal{S}$ has $O(f(n))$ group complexity;*
8. *The communication between the server $\mathcal{S}$ and the client $\mathcal{C}$ has $O(p(n) + \alpha(n))$ group complexity;*
9. *The verification at the client $\mathcal{C}$ has $O(v(n))$ access complexity;*
10. *For a query $q \in \mathbf{Q}$ sent by the client $\mathcal{C}$ to the server $\mathcal{S}$ at any time (even after updates), let α be an answer and let π be a proof returned by the server $\mathcal{S}$. With probability $\Omega(1 - \mathsf{neg}(k))$, the client $\mathcal{C}$ accepts the answer α if and only if α is correct.*

Proof: (**Complexity**) The protocol in question is Protocol 2.1. In the setup phase the access complexity of Item 1a and Item 1b is $O(1)$ (they are not accessing the data structure at all). The access complexity of Item 1c is $O(s(n))$ since it involves one call to algorithm $\mathsf{setup}()$. Therefore the total setup access complexity at the source is $O(s(n))$. The update at the source involves one call to algorithm $\mathsf{update}()$ (see Item 2a), therefore it has $O(u(n))$ access complexity. Both the source and the server store the authenticated data structure $\mathsf{auth}(D)$, therefore their space has $O(f(n))$ group complexity. The communication between the source and the server has $O(i(n))$ group complexity, since information u and upd are sent (see Item 2b) and also periodically (i.e., every Δt time units) a signature on the timestamped digest is sent (see Item 3c), which has $O(1)$ group complexity. The update at the server has $O(r(n))$ access complexity since it involves one call to algorithm $\mathsf{refresh}()$ (see Item 2c). Computing a proof at the server (i.e., query at the server) has $O(q(n))$ access complexity since it involves one call to algorithm $\mathsf{query}()$ (see Item 4b). The communication between the server and the client has $O(p(n) + \alpha(n))$ group complexity since it involves sending off the proof and the answer, a digest, a signature and a timestamp (see Item 4c). Finally the verification at the client has $O(v(n))$ access complexity since it involves checking the Relations 4(d)i ($O(1)$ complexity), 4(d)ii ($O(1)$ complexity) and 4(d)iii ($O(v(n))$ complexity since it involves one call to algorithm $\mathsf{verify}()$).

(**Security**) Let now T be the time of a query $q \in \mathbf{Q}$. The client sends query q to the server (Item 4a). The server replies with an answer α, a proof π, a digest d, a timestamp t, such that $T - t < \Delta t$, and a signature sgn (Item 4c), computed by the trusted source (Item 3a). If server $\mathcal{S}$ follows the protocol, it will output a correct answer α and a correct digest d (output by $\mathsf{refresh}()$). Since a *correct* signature scheme and a *correct* authenticated data structure scheme are used, it follows that the client will accept with all but negligible probability (see Definition 2.4). Suppose now that the server does not follow the protocol and that the client accepts α, while α is an incorrect answer. In this case, the following three events have to be true:

- $\mathcal{E}_1$: $\mathcal{ACCEPT} \leftarrow \mathcal{VERIFY}(\mathsf{sgn}, d||t, \mathcal{PK})$. This event can be partitioned into the following events:

  1. $\mathcal{E}_{10}$: Digest d is not the correct digest, i.e., the one signed by the source at time t;
  2. $\mathcal{E}_{11}$: Digest d is the correct digest, i.e., the one signed by the source at time t;

- $\mathcal{E}_2$: $\mathsf{accept} \leftarrow \mathsf{verify}(q, \alpha, \pi, d, \mathsf{pk})$;

- $\mathcal{F}$: α is an incorrect answer to query q.

Therefore the probability in question is the probability $\Pr[\mathcal{E}_1 \cap \mathcal{E}_2 \cap \mathcal{F}]$. By using some probability calculus this is bounded by

$$\Pr[\mathcal{E}_2|\mathcal{E}_{11} \cap \mathcal{F}] + \Pr[\mathcal{E}_{10}] = P_1 + P_2 \,.$$

However, since a *secure* authenticated data structure scheme $\mathcal{A}$ is used, P_1 is $\mathsf{neg}(k)$. Finally, since a *secure* (i.e., unforgeable) signature scheme is used, P_2 is also $\mathsf{neg}(k)$. This concludes the security proof for the three-party protocol. □

2.2.2 Two-party authenticated data structures protocol

We now continue with the description of the two-party authenticated data structures protocol. This protocol involves a trusted client and an untrusted server (see Figure 2.2). The client can store constant space and cannot usually perform expensive computations (e.g., iPhone). This model is close to the model of *outsourced verifiable computation* [5, 28, 41], which has recently appeared in the literature. The main differences with the three-party protocol are the following:

1. The client performs the updates, sends the queries to the server and verifies the answers;
2. The client stores some state of *constant* size only and cannot store the authenticated data structure;

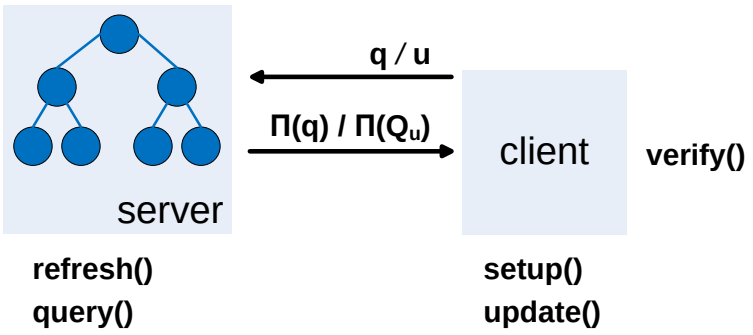

Figure 2.2: The two-party authenticated data structures protocol. During the query phase, the client sends a query $q \in \mathbf{Q}$ to the server and the server runs algorithm $\mathsf{query}()$ to output the proof $\Pi(q)$ for the answer. During the update phase, the client sends to the server an update $u \in \mathbf{U}$, which relates to a certain set of queries $Q_u \subseteq \mathbf{Q}$. Then the server computes the set of proofs $\Pi(Q_u)$. This set of proofs will be used by function $z(.)$ of Assumption 2.1, which will output $\delta_u(D_h)$ and $\delta_u(\mathsf{auth}(D_h))$, which are subsequently input to algorithm $\mathsf{update}()$.

3. The authenticated data structure scheme used need not be publicly-verifiable.

Before we continue with the description of the protocol, we give a necessary definition, inspired by the definition of t-heavy locations in the work on memory checking lower bounds of Dwork et al. [35]. This definition will allow us to formally characterize the parts of the authenticated data structure that are accessed during an update. This characterization will be important for formalizing the two-party protocol:

Definition 2.10 (Heavy locations of an update) *Let $\mathcal{A}$ be an authenticated data structure scheme for a data structure scheme $\mathcal{D}(\mathbf{U}, \mathbf{Q})$ defined by the collection of algorithms $\{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$, k be the security parameter and $\{\mathsf{sk}, \mathsf{pk}\} \leftarrow \mathsf{genkey}(1^k)$. Let $u \in \mathbf{U}$ be an update that updates the data structure from D to D' and the authenticated data structure from $\mathsf{auth}(D)$ to $\mathsf{auth}(D')$. We denote with $\delta_u(D) \subseteq D$ and $\delta_u(\mathsf{auth}(D)) \subseteq \mathsf{auth}(D)$ the set of memory locations of D and $\mathsf{auth}(D)$ respectively that are accessed by algorithm $\mathsf{update}()$ during update u. Analogously, we denote with $\delta_u(D') \subseteq D'$ and $\delta_u(\mathsf{auth}(D')) \subseteq \mathsf{auth}(D')$ the set of memory locations of D' and $\mathsf{auth}(D')$ respectively that*

are altered by algorithm $\mathsf{update}()$ *during update* u*. Namely, we have the following equivalence*

$$\begin{aligned} \{D', \mathsf{auth}(D'), d', \mathsf{upd}\} &\leftarrow \mathsf{update}(u, D, \mathsf{auth}(D), d, \mathsf{sk}, \mathsf{pk}) \\ &\Leftrightarrow \\ \{\delta_u(D'), \delta_u(\mathsf{auth}(D')), d', \mathsf{upd}\} &\leftarrow \mathsf{update}(u, \delta_u(D), \delta_u(\mathsf{auth}(D)), d, \mathsf{sk}, \mathsf{pk})\,. \end{aligned}$$

The above definition implies that, in order for $\mathsf{update}()$ to execute, it needs to have access only to the parts of the (authenticated) data structure required for execution, and not to the whole (authenticated) data structure. This is essential for the two-party model, since the client does cannot store all the data locally.

An assumption for formalizing the two-party protocol

Let $\mathcal{A} = \{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$ be an authenticated data structure scheme for a data structure scheme $\mathcal{D}(\mathbf{U}, \mathbf{Q})$. Let $Q' \subseteq \mathbf{Q}$ be a set of queries and $\Pi(Q')$ and $\alpha(Q')$ be the sets of proofs and answers respectively output by $\mathsf{query}(q', D_h, \mathsf{auth}(D_h), \mathsf{pk})$ for all queries $q' \in Q'$. We make the following assumption in order to achieve a generalized application of an authenticated data structure scheme in the two-party protocol: For every update $u \in \mathbf{U}$, there exists a *set* of data structure queries $Q_u \subseteq \mathbf{Q}$ such that the set of memory locations of D_h and the set of memory locations of $\mathsf{auth}(D_h)$ accessed by algorithm $\mathsf{update}()$ during update u (i.e., $\delta_u(D_h)$ and $\delta_u(\mathsf{auth}(D_h))$) is a function of $Q_u, \alpha(Q_u), \Pi(Q_u)$. We state this formally now:

Assumption 2.1 (Update query) *Let* $\mathcal{A} = \{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$ *be an authenticated data structure scheme for data structure scheme* $\mathcal{D}(\mathbf{U}, \mathbf{Q})$*. For every update* $u \in \mathbf{U}$ *on data structure* D_h*, there exists a set of queries* $Q_u \subseteq \mathbf{Q}$ *such that if* $\{\Pi(Q_u), \alpha(Q_u)\} \leftarrow \mathsf{query}(Q_u, D_h, \mathsf{auth}(D_h), \mathsf{pk})$*, then it is*

$$\{\delta_u(\mathsf{auth}(D_h)), \delta_u(D_h)\} = z(Q_u, \alpha(Q_u), \Pi(Q_u))\,,$$

for some well-defined function $z(.)$*, computable with complexity* $O(|Q_u| v(n))$*, where* $v(n)$ *is the access complexity of* $\mathsf{verify}()$*.*

The following example gives some intuition about Assumption 2.1.

Example 2.1 Let us consider the case of a Merkle tree [77] that is used to verify the contents of an n-index array $\mathbf{A}$. Let also u be the update "set $\mathbf{A}[i] = x$" and let $\mathbf{A}[i] = y$ before the update u takes place. The respective set of queries Q_u consists of one query q_u, namely the query "return the contents of cell i". Let $\Pi(q_u)$ be a verifying proof (a logarithmic-sized chain of hash values) and let $\alpha(q_u)$ be the correct answer, i.e., $\alpha(q_u) = y$. Note that $\delta_u(D_h) = \mathbf{A}[i] = y$ and $\delta_u(\mathsf{auth}(D_h))$ is the path of hash values that is computed during the verification. Namely, function $z(.)$ in this case is the Merkle tree verification algorithm. Moreover, $z(.)$ has $O(\log n)$ complexity, equal to the complexity of the verification algorithm. Thus we conclude that for the Merkle tree authenticated data structure (as well as for the similar authenticated data structure scheme presented in Chapter 4), Assumption 2.1 is true and therefore we can implement the authenticated data structure scheme in the two-party model in a generic way (a similar implementation for skip lists is described in [92]).

Protocol 2.2 *A two-party authenticated data structures protocol involves two participating entities: A* trusted *client* $\mathcal{C}$ *(that has access both to the public and the secret keys) and an* untrusted *server* $\mathcal{S}$ *(that has access only to the public keys). Let* k *be the security parameter and* $\mathcal{A} = \{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$ *be a correct and secure authenticated data structure scheme for a data structure scheme* $\mathcal{D}(\mathbf{U}, \mathbf{Q})$*. The protocol has four phases:*

1. Setup phase*: Client* $\mathcal{C}$ *owns the data structure* D *at time* t_0*. The setup phase consists of the following steps:*

   (a) $[\mathcal{C}]_{t_0}$*:* $\{\mathsf{sk}, \mathsf{pk}\} \leftarrow \mathsf{genkey}(1^k)$*; (*authenticated data structure scheme keys generation*)*

   (b) $[\mathcal{C}]_{t_0}$*:* $\{\mathsf{auth}(D), d\} \leftarrow \mathsf{setup}(D, \mathsf{sk}, \mathsf{pk})$*; (*authenticated data structure generation*)*

   (c) $[\mathcal{C} \rightarrow \mathcal{S}]_{t_0}$*:* $D, \mathsf{auth}(D), d$*; (*outsourcing*)*

(d) $[\mathcal{C}]_{t_0}$*: Delete* D *and* $\mathsf{auth}(D)$*. Store* d*. (*storing necessary information only*)*

2. Update phase*: Let* $u \in \mathbf{U}$ *be any update issued by client* $\mathcal{C}$ *on data structure* D *at time* t_τ*. Let* D' *be the updated data structure. The update phase consists of the the following steps:*

   (a) $[\mathcal{C} \rightarrow \mathcal{S}]_{t_\tau}$*:* u*; (*sending the update*)*

   (b) $[\mathcal{S}]_{t_\tau}$*:* $\{\Pi(Q_u), \alpha(Q_u)\} \leftarrow \mathsf{query}(Q_u, D_h, \mathsf{auth}(D_h), \mathsf{pk})$ *(*computing proofs and answers for the set of queries Q_u related to update u—see Assumption 2.1*)*

   (c) $[\mathcal{C} \leftarrow \mathcal{S}]_{t_\tau}$*:* $\Pi(Q_u), \alpha(Q_u)$*; (*sending the data required for the update*)*

   (d) $[\mathcal{C}]_{t_\tau}$*: Execute the following steps:*

      i. If $\mathsf{reject} \leftarrow \mathsf{verify}(Q_u, \alpha(Q_u), \Pi(q_u), d, (\mathsf{sk})^*, \mathsf{pk})$*, output* REJECT*;*

      ii. $\{\delta_u(\mathsf{auth}(D_h)), \delta_u(D_h)\} = z(Q_u, \alpha(Q_u), \Pi(Q_u)$*; (use of function* $z(.)$*, see Assumption 2.1)*

      iii. $\{\delta_u(D'), (\delta_u(\mathsf{auth}(D')))^*, d', \mathsf{upd}\} \leftarrow \mathsf{update}(u, \delta_u(D), (\delta_u(\mathsf{auth}(D)))^*, d, \mathsf{sk}, \mathsf{pk})$*;* $d = d'$*; (*generating and storing the new digest by running update() only on the heavy locations of the update—see Definition 2.10*)*

      iv. If $\mathsf{upd} \neq \emptyset$ *go to Item 2e; Else go to Item 2f; (*not always need for a second round*)*

   (e) $[\mathcal{C} \rightarrow \mathcal{S}]_{t_\tau}$*:* upd*; (*sending relative update information*)*

   (f) $[\mathcal{S}]_{t_\tau}$*:* $\{D', \mathsf{auth}(D'), d'\} \leftarrow \mathsf{refresh}(u, D, \mathsf{auth}(D), d, \mathsf{upd}, \mathsf{pk})$*;* $D = D'$*,* $\mathsf{auth}(D) = \mathsf{auth}(D')$*,* $d = d'$*. (*authenticated data structure update*)*

3. Query phase*: The query phase proceeds as follows, between the client* $\mathcal{C}$ *and the server* $\mathcal{S}$*. Let* T *be the time of the query* $q \in \mathbf{Q}$*:*

   (a) $[\mathcal{C} \rightarrow \mathcal{S}]_T$*:* q*; (*sending the query*)*

   (b) $[\mathcal{S}]_T$*:* $\{\Pi(q), \alpha(q)\} \leftarrow \mathsf{query}(q, D, \mathsf{auth}(D), \mathsf{pk})$*; (*answer and proof computation*)*

(c) $[\mathcal{C} \leftarrow \mathcal{S}]_T$*:* $\alpha(q), \Pi(q)$*; (*sending answer and proof*)*

(d) $[\mathcal{C}]_T$*: Output* ACCEPT *if and only if* accept $\leftarrow$ verify$(q, \alpha(q), \Pi(q), d, (\mathsf{sk})^*, \mathsf{pk})$ *else output* REJECT*. (*verification of answer*)*

Theorem 2.2 *Let* $\mathcal{A} = \{$genkey, setup, update, refresh, query, verify$\}$ *be a correct and secure authenticated data structure scheme (not necessarily publicly-verifiable) for a data structure scheme* $\mathcal{D}(\mathbf{U}, \mathbf{Q})$ *and* k *be the security parameter. Suppose Assumption 2.1 holds for* $\mathcal{A}$*, such that for each update* $u \in \mathbf{U}$ *there exists a respective set of queries* Q_u*, as defined in Assumption 2.1. Let* $g(n)$*,* $s(n)$*,* $u(n)$*,* $r(n)$*,* $q(n)$ *and* $v(n)$ *be the* access *complexities of algorithms* genkey*,* setup*,* update*,* refresh*,* query*,* verify *respectively. Let also* $i(n)$*,* $p(n)$ *and* $\alpha(n)$ *be the* group *complexity of the information* upd *(output by algorithm* update$())$*, of the proof* $\Pi(q)$ *(output by algorithm* query$())$ *and of the answer* $\alpha(q)$ *(output by algorithm* query$())$ *respectively and* $f(n)$ *be the* group *complexity of the authenticated data structure* auth(D) *(output by algorithm* setup$())$*. Then there exists a two-party authenticated data structures protocol involving a trusted client* $\mathcal{C}$ *and an untrusted server* $\mathcal{S}$ *for verifying queries* $q \in \mathbf{Q}$ *and at the same time supporting updates* $u \in \mathbf{U}$ *such that:*

1. *The protocol is non-interactive if the information* upd *output by* update() *is empty; Otherwise it requires one round of interaction;*

2. *The setup at the client* $\mathcal{C}$ *has* $O(s(n))$ *access complexity;*

3. *The update at the client* $\mathcal{C}$ *has* $O(|Q_u|v(n) + u(n))$ *access complexity;*

4. *The verification at the client* $\mathcal{C}$ *has* $O(v(n))$ *access complexity;*

5. *The space needed at the client* $\mathcal{C}$ *has* $O(1)$ *group complexity;*

6. *The communication between the client* $\mathcal{C}$ *and the server* $\mathcal{S}$ *has* $O(|Q_u|(p(n) + \alpha(n)) + i(n))$ *group complexity during updates and* $O(p(n) + \alpha(n))$ *group complexity during queries;*

7. *The update at the server $\mathcal{S}$ has $O(|Q_u|q(n) + r(n))$ access complexity;*

8. *The query at the server $\mathcal{S}$ has $O(q(n))$ access complexity;*

9. *The space needed at the server $\mathcal{S}$ has $O(f(n))$ group complexity;*

10. *For a query $q \in \mathbf{Q}$ sent by the client $\mathcal{C}$ to the server $\mathcal{S}$ at any time (even after updates), let α be an answer and let π be a proof returned by the server $\mathcal{S}$. With probability $\Omega(1 - \mathsf{neg}(k))$, the client $\mathcal{C}$ accepts the answer α if and only if α is correct.*

Proof: Note that Item 2e, which introduces one round of interaction, is executed only when $\mathsf{upd} = \emptyset$. Therefore Item 1 of Theorem 2.2 holds.

(**Complexity**) The protocol in question is Protocol 2.2. In the setup phase the access complexity of Item 1a is $O(1)$ (it is not accessing the data structure at all). The access complexity of Item 1b is $O(s(n))$ since it involves one call to algorithm $\mathsf{setup}()$. Therefore the total setup access complexity at the client is $O(s(n))$. The update at the client involves one verification of $\alpha(Q_u)$ (see Item 2(d)i, which has $O(|Q_u|v(n))$ complexity), the application of function $z(.)$ to output $\delta_u(D_h)$ and $\delta_u(\mathsf{auth}(D_h))$ respectively (see Item 2(d)ii)—this, by Assumption 2.1 has $O(|Q_u|v(n))$ complexity— and one call to algorithm $\mathsf{update}()$ (see Item 2(d)iii). Therefore the total complexity is $O(|Q_u|v(n) + u(n))$. The server stores the authenticated data structure $\mathsf{auth}(D)$, therefore its space has $O(f(n))$ group complexity. The client stores only the digest d, therefore it needs space of $O(1)$ group complexity. The communication between the client and the server has

- $O(|Q_u|(p(n)+\alpha(n))+i(n))$ group complexity during an update u, since $\Pi(Q_u)$ (Item 2c), $\alpha(Q_u)$ (Item 2c) and information upd (Item 2e) are exchanged between the two parties;

- $O(p(n) + \alpha(n))$ group complexity during queries since it involves sending off a proof and an answer for one query (see Item 4c).

The update at the server has $O(|Q_u|q(n)+r(n))$ access complexity since it involves computing $\alpha(Q_u)$ and $\Pi(q_u)$ (Item 2b) and one call to algorithm $\mathsf{refresh}()$ (see Item 2f). Computing a

proof at the server (i.e., query at the server) has $O(q(n))$ access complexity since it involves one call to algorithm query() (see Item 3b). Finally the verification at the client has $O(v(n))$ access complexity since it involves one call to algorithm verify() (Item 3d), which has $O(v(n))$ complexity.

(**Security**) Let now T be the time of a query $q \in \mathbf{Q}$. The client sends query q to the server (Item 3a). The server replies with an answer α and a proof π (Item 3b). If server $\mathcal{S}$ follows the protocol, it will output a correct answer α. Since a *correct* authenticated data structure scheme is used, it follows that the client will accept with all but negligible probability (see Definition 2.4). Suppose now that the server does not follow the protocol and that the client accepts α, while α is an incorrect answer. Let $\mathcal{E}$ be that event. Note that $\mathcal{E}$ can be written as

$$\mathcal{E} = \mathcal{E} \cap [(\text{digest } d \text{ is correct}) \cup (\text{digest } d \text{ is not correct})] \ .$$

Therefore

$$\begin{aligned} \Pr[\mathcal{E}] &\leq \Pr[\mathcal{E} \cap (\text{digest } d \text{ is correct})] + \Pr[\mathcal{E} \cap (\text{digest } d \text{ is not correct})] \\ &\leq \Pr[\mathcal{E} \cap (\text{digest } d \text{ is correct})] + \Pr[\text{digest } d \text{ is not correct}] \\ &\leq \Pr[\mathcal{E} | (\text{digest } d \text{ is correct})] + \Pr[\text{digest } d \text{ is not correct}] \\ &= P_1 + P_2 \, . \end{aligned}$$

However, since a *secure* authenticated data structure scheme $\mathcal{A}$ is used, P_1 is $\mathsf{neg}(k)$. Also by Lemma 2.1, P_2 is also $\mathsf{neg}(k)$. This completes the proof. □

Lemma 2.1 *Let k be the security parameter. The digest d used by every verification at Item 3d of Protocol 2.2 is correct with probability $\Omega(1 - \mathsf{neg}(k))$, even in the presence of updates.*

Proof: Let u_i, for some $i \geq 0$, be the first update where the protocol rejects at Item 2(d)i (if there is such an update). We prove the lemma by induction on the number of updates before

update u_i. Before the execution of the first update u_0 issued by the client, the lemma holds since the digest d_0 is output by $\mathsf{setup}()$ in Item 1b, run by the trusted client. Therefore d_0 is correct with probability 1. Suppose the lemma holds for the time period before the execution of the update u_{i-1} at Item 2(d)iii, namely the digest d_{i-1}, used by every verification before update u_{i-1}, is correct with probability $\Omega(1-\mathsf{neg}(k))$. Let $\alpha(Q_{u_{i-1}})$ and $\Pi(Q_{u_{i-1}})$ be the answers and the proofs related to the update u_{i-1}, derived from Assumption 2.1. Since the protocol does not reject during update u_{i-1}, the verification of $\alpha(Q_{u_{i-1}})$ and $\Pi(Q_{u_{i-1}})$ at Item 2(d)i is successful. Therefore, since d_{i-1} is correct with probability $\Omega(1-\mathsf{neg}(k))$ (by inductive hypothesis), the values $\alpha(Q_{u_{i-1}})$ and $\Pi(Q_{u_{i-1}})$ are also correct with probability $\Omega(1-\mathsf{neg}(k))$, since a secure authenticated data structure scheme is used (see Definition 2.5). This implies that $\delta_{u_{i-1}}(\mathsf{auth}(D_{i-1})), \delta_{u_{i-1}}(D_{i-1})$, output by $z(Q_{u_{i-1}}, \alpha(Q_{u_{i-1}}), \Pi(Q_{u_{i-1}}))$ at Item 2(d)ii are correct as well with overwhelming probability. Therefore $\mathsf{update}()$ outputs the correct digest d_i with probability $\Omega(1-\mathsf{neg}(k))$, since it executes on correct data $\delta_{u_{i-1}}(\mathsf{auth}(D_{i-1}))$ and $\delta_{u_{i-1}}(D_{i-1})$ at Item 2(d)iii. Thus the digest d_i, used by every verification before update u_i, is correct with probability $\Omega(1-\mathsf{neg}(k))$. This completes the proof. □

2.3 Related work

In this section we present a general literature review related to authenticated data structures [105] and more broadly to methods developed for checking the integrity of data and computations stored and executed by adversarial parties. In the description of the related work, we denote with n the size of the data structure (e.g., number of elements stored in the dictionary). Note that in the subsequent chapters of the thesis, there are references to literature work that is more related to the algorithmic or cryptographic problem that is studied in the specific chapter.

2.3.1 Generic collision-resistant hashing

Since the appearance of Merkle's seminal paper on hash trees [77], many works in authenticated data structures literature have used generic collision-resistant hashing to realize efficient integrity checking mechanisms. Generic collision-resistant hashing refers to the use of certain cryptographic hash functions, such a MD-5 and SHA-2, as a black box, i.e., without exploiting the internal (algebraic) structure of the algorithms implementing them. These constructions have been known to be very efficient in practice. However, due to the heuristic nature of their security arguments, various attacks on them (e.g., recent attack on SHA-1 [111] and MD-5 [103]) have appeared over the years.

Nevertheless, several constructions based on generic collision-resistant hashing have been developed for the verification of various queries on dynamic data structures. After Naor and Nissim dynamized Merkle's solution [81], providing the first dynamic authenticated dictionary, Goodrich and Tamassia [48] presented another efficient realization of a dynamic authenticated dictionary with skip lists, which was enhanced with persistence by Anagnostopoulos et al. [4] and was subsequently tested by Goodrich et al. [50] and implemented in a two-party protocol by Papamanthou and Tamassia [92], finding many applications in authenticated file systems (e.g., see Goodrich et al. [46]) and authenticated outsourced storage (e.g., see Heitzmann et al. [56]). Martel et al. [75] presented several new authenticated data structures for more complicated computations (e.g., database queries), supporting efficient I/O algorithms as well. Distributed authenticated hash tables were introduced by Tamassia and Triandopoulos [104] and the verification of more complicated queries, such as connectivity queries on graphs and fractional cascading, is presented by Goodrich et al. [53]. All the above authenticated data structures achieve $O(\log n)$ complexity costs, which are shown to be optimal [106] (only for methods using generic collision-resistant hashing as a black box). The database community has also employed generic collision-resistant hashing extensively in order to verify various structures and queries related to systems and applications such as I/O-efficient search trees [66], queries on streams [67], database join queries [112], shortest

path computations [72] and set operations [33]. Other types of queries such as 2-dimensional range search have also been investigated [6].

2.3.2 More advanced cryptography

Although the majority of authenticated data structures developed in the literature are based on generic collision-resistant hashing, there have been some solutions for verifying queries in various settings using other cryptographic primitives, such as one-way accumulators. One-way accumulators, that were introduced by Benaloh and de Mare [13], are based on the RSA exponentiation function and comprise an efficient way of securely compressing multiple inputs into one *succinct* representation, so that efficient proofs of membership can be computed. Implemented with an RSA accumulator, they satisfy quasi-commutativity, a useful property that common generic collision-resistant functions lack, which allows for efficient updates and convenient preprocessing. Refinements of the RSA accumulator are also given by Baric and Pfitzmann [10], where except for one-wayness, collision resistance is achieved, and also by Gennaro et al. [42] and by Sander et al. [101]. Dynamic accumulators (along with protocols for zero-knowledge proofs) were introduced by Camenisch and Lysyanskaya [23].

A first application of accumulators in the authenticated data structures model was made by Goodrich et al. [51]; in this work, and in favor of constant complexity proofs, general $O(n^{\epsilon})$ bounds are derived for various complexity measures such as query and update complexity (as opposed to logarithmic bounds of methods using generic collision-resistant hashing). An authenticated data structure [52] that combines hierarchical hashing with the scheme of [51] appeared later, and a similar hybrid authentication scheme was developed by Nuckolls [84].

Accumulators using other cryptographic primitives (groups admitting bilinear pairings) the security of which is based on other assumptions (hardness of strong Diffie-Hellman problem) are presented by Nguyen [83] and Camenisch et al. [22]. However, updates in the work of Nguyen [83] are inefficient when the trapdoor information is not known: individual pre-computed witnesses can be updated with constant complexity, thus incurring a linear total

cost for updating all the witnesses after an update in the set. Also, the accumulator by Camenisch et al. [22] requires space proportional to the number of elements *ever* accumulated in the set (book-keeping information of considerable size is needed), or otherwise important constraints on the range of the accumulated values are required. Efficient dynamic accumulators for non-membership proofs are presented by Li et al. [68]. Accumulators for batch updates are presented by Wang et al. [110] and accumulator-like expressions to authenticate static sets under the *provable data possession model* are presented by Ateniese et al. [7, 37]. The work by Sander et al. [100] studies efficient algorithms for accumulators with unknown trapdoor information. Finally in the work of lower bounds by Dwork et al. [35], and simultaneously with work performed by Papamanthou et al. [90], logarithmic lower bounds as well as constructions achieving query-update cost trade-offs have been studied in the memory-checking model. Tree hierarchies are authenticated (with access-control enabled) by Atallah et al. [6], and by using bilinear pairings.

A study of an extensive suite of various authenticated data structures problems can be found in the PhD theses of Triandopoulos [108] and Crosby [30]. Applications of authenticated data structures in distributed systems integrity are presented in the PhD thesis of Maniatis [73].

2.3.3 Relation to memory checking

Authenticated data structures are closely related to the memory checking model, which was originally defined by Blum et al. [15] and has consequently been studied in several works [35, 82]. Both authenticated data structures and memory checking have as goal to verify the operation of some functionality that is offered by an untrusted and possibly malicious server, i.e., to design cryptographic protocols that can efficiently verify the correctness of the corresponding provided functionality. In memory checking, the functionality offered by a memory array is being verified, namely read and write operations in a one-dimensional table with indices $1, \ldots, n$. A read operation returns the value that is stored at a given index

$j \in [1, n]$ and a write operation involves changing the content of a given index $j \in [1, n]$ to a new given value (similar to the authenticated data structure introduced in Chapter 4); a memory-checking protocol verifies that a value that is read from any given index j is the last value that was written to that index j. The celebrating result by Blum et al. [15] states that this fundamental read-write functionality on n memory cells can be verified by reading $O(\log n)$ special values that are stored at some additional unreliable memory cells of total size $O(n)$. In authenticated data structures, the functionality offered by a *data structure* (e.g., heap, dynamic trees, hash table, dictionary, inverted index, fractional cascading, etc.) is being verified, namely query and update operations defined over a structured and dynamic data collection. For example, for the case of dynamic trees [53], a query can be "is node v a child of u?" and an update can be "move subtree T to node v".

It is important to note that since any ordinary data structure (of size n) is implemented in the RAM model and since every operation is simulated by reading and writing bits in memory, it is consequently true that every data structure can be authenticated using the memory checking model by individually verifying (with $O(\log n)$ complexity) every elementary read or write operation needed during a query or update in the data structure. This reduction immediately gives raise to a very reasonable question: Is it useful to work with the more abstract model of authenticated data structures?

The answer is yes, and the main reason is related to *efficiency*. The reduction of authenticated data structures to memory checking is in practice highly inefficient. Firstly, the verification of any read or write operation introduces a logarithmic (in the size of the data structure) multiplicative factor in the complexity of the verification protocol. Secondly and more importantly, verifying the functionality of a data structure through memory checking corresponds to verifying the entire execution of the query or the update algorithm that is defined for the data structure in study, which is in general unnecessary because what is needed to be verified in the result of such a query or update algorithm and not its entire execution. The best way to demonstrate how inefficient such a reduction to memory checking can be is

through a concrete and very illustrative example. Consider the verification of *range search queries*. To verify a range search query by solely relying on the known memory-checking techniques, we need to verify the entire search procedure in an underlying data structure of range searching, e.g., the range search tree, a procedure which requires $O(\log n + k)$ reads of memory locations, where n is the size of the data set and k is the number of the elements belonging to the queried range. Given the logarithmic overhead introduced by the memory checking (by using for example Blum's memory checker [15]), the total verification cost is $O(\log^2 n + k \log n)$. Instead, by using an implementation of an authenticated dictionary (e.g., a Merkle tree), the complexity to authenticate range search queries is only $O(\log n + k)$. Even better as it has been shown by Tamassia and Triandopoulos [107], by using optimal data certification techniques in combination with optimal authenticated dictionaries, it is possible to authenticate range search queries, optimally, in $O(k)$ complexity using proofs of size only $O(\log k)$. Therefore, authenticated data structures can significantly reduce the complexity for the authentication of *complicated* queries, by combining cryptography with algorithmics, which is the basis of constructing efficient authenticated data structures solutions. Additionally, apart from efficiency, in the authenticated data structures model, communication complexity *does* matter. We are interested in minimizing the *size* of the proof that the untrusted server computes for the verification of a query. However in memory checking, bandwidth does not come into place since query complexity is the main complexity measure that is studied. Overall, authenticated data structures provide a more powerful, more refined and more expressive model for studying the verification of computations that take place during data management and data querying.

Finally, we note that memory checking solutions have been traditionally constructed with cryptographic primitives that bear very weak assumptions (e.g., existence of one-way functions [15]). However, authenticated data structures do employ stronger (e.g., strong RSA assumption [51]) assumptions, still widely acceptable and widely used by the cryptography community.

Chapter 3

Accumulators for authenticated hash tables

In this chapter we describe our first authenticated data structure scheme for a *hash table* data structure. We use *cryptographic accumulators* [23] as our basic cryptographic primitive to verify standard hash table queries. Specifically, our main results (Theorem 3.2 and Theorem 3.4) show how to use two different accumulator schemes [23, 83] in a hierarchical way (see Figure 3.1) over the set and the underlying hash table, in order to achieve the verification of both *membership* and *non-membership* queries.

In the presented fully-dynamic schemes, communication and verification complexities are constant, the query complexity is constant and the update complexity is *sublinear*, realizing the first authenticated hash table with this performance. Our schemes (denoted with $\mathcal{RHT}$ and $\mathcal{BHT}$ in Table 3.1) strictly improve, in terms of complexity, upon previous schemes based on accumulators and other cryptographic primitives (for a detailed comparison, see Table 3.1). Their security is based on two widely accepted assumptions, the strong RSA assumption [10] and the bilinear q-strong Diffie-Hellman assumption [16]. Finally, to meet the needs of different data-access patterns, we extend our schemes to achieve a reverse performance, i.e., sublinear query cost, but constant update cost.

Table 3.1: In this table, we exhibit a detailed comparison of asymptotic access and group complexities of various authenticated data structure schemes in the literature with the complexities of our schemes. The underlying data structure scheme is for a hash table storing n elements. All the authenticated data structure schemes compared are defined by algorithms {genkey, setup, update, refresh, query, verify} (see Definition 2.3). Parameter $0 < \epsilon < 1$ is a constant, "D. Log" stands for "Discrete Logarithm", "Generic CR" stands for "Generic Collision Resistance" and "B. q-DH" stands for "Bilinear q-strong Diffie-Hellman". In all constructions the authenticated data structure has group complexity (i.e., size) $O(n)$ and genkey() has $O(1)$ complexity. $\Pi(q)$ denotes the proof for a query q and upd is the update information output by algorithm update(). Our schemes are denoted with $\mathcal{RHT}$ (RSA-based authenticated hash table) and $\mathcal{BHT}$ (bilinear-map-based authenticated hash table). The "one-star" notation * denotes an *expected* complexity, the "two-star" notation ** denotes an *expected amortized* complexity, whereas the "plus" notation $^+$ denotes an *amortized* complexity. All schemes in the table are publicly-verifiable.

	[15, 48, 75, 81]	[11]	[23, 101]	[51]	$\mathcal{RHT}$	[83]	$\mathcal{BHT}$
setup()	n	n	n	n	n	n	n
update()	$\log n$	1	1	n^ϵ	1^{**}	1	1^{**}
refresh()	$\log n$	1	$n \log n$	n^ϵ	$n^\epsilon \log n \;/\; 1^{**}$	n	$n^\epsilon \;/\; 1^{**}$
query()	$\log n$	n	1	n^ϵ	$1 \;/\; n^\epsilon{}^*$	1	$1 \;/\; n^\epsilon \log n^*$
verify()	$\log n$	n	1	1	1^*	1	1
proof $\Pi(q)$	$\log n$	n	1	1	1^*	1	1
info. upd	1	1	1	n^ϵ	1	1	1^+
assumption	Generic CR	D. Log	Strong RSA			B. q-DH	

3.1 Preliminaries

In this section we describe some algorithmic and cryptographic primitives and other useful concepts that are used in our approach.

3.1.1 Hash tables

The main functionality of a hash table data structure $\mathbf{T}(\mathcal{X})$ is to support optimal complexity look-ups of elements that belong to a general dynamic set $\mathcal{X}$ (i.e., not necessarily ordered). Elements can be inserted or deleted from $\mathcal{X}$. The elements in $\mathcal{X}$ are drawn from a universe $\mathcal{U}$.

The data structure scheme. The data structure scheme $\{\textbf{query}(), \textbf{update}(), \textbf{check}()\}$ as defined in Definition 2.2 for a hash table $\mathbf{T}(\mathcal{X})$ is as follows:

1. $\{\mathsf{true}, \mathsf{false}\} \leftarrow \textbf{query}(x, \mathbf{T}(\mathcal{X}))$: Given an element $x \in \mathcal{U}$, return true if $x \in \mathcal{X}$ or false otherwise;

2. $\mathbf{T}(\mathcal{X}') \leftarrow \textbf{update}(x, \mathbf{T}(\mathcal{X}))$: Given an element $x \in \mathcal{U}$ such that $x \notin \mathcal{X}$, *insert* element x into $\mathcal{X}$ and output $\mathbf{T}(\mathcal{X}')$; Given an element $x \in \mathcal{U}$ such that $x \in \mathcal{X}$, *delete* element x from $\mathcal{X}$ and output $\mathbf{T}(\mathcal{X}')$;

3. $\{\mathsf{accept}, \mathsf{reject}\} \leftarrow \textbf{check}(x, b \in \{\mathsf{true}, \mathsf{false}\}, \mathbf{T}(\mathcal{X}))$: If $x \in \mathcal{X}$ (or $x \notin \mathcal{X}$) and $b = \mathsf{false}$ (or $b = \mathsf{true}$), return reject. Else return accept.

Note that answering a hash table query can be implemented to have $O(1)$ expected complexity (see Theorem 3.1) and that both insertions and deletions in a hash table can be implemented to have $O(1)$ expected amortized complexity (see Theorem 3.1).

Implementation. Different ways of implementing hash tables have been extensively studied (e.g., [34, 58, 62, 69, 80]). Here we use a simple approach for the implementation of the plain hash table: Suppose we wish to store n elements from a universe $\mathcal{U}$ in a data structure

so that we can have expected constant look-up complexity. For totally ordered universes and by searching based on comparisons, it is well known that an $\Omega(\log n)$ lower bound is in place. Essential for the construction of a hash table—and for achieving better efficiency than $\Omega(\log n)$—is a two-universal hash function:

Definition 3.1 (Two-universal hash function [26]) *A* two-universal hash function $H : \mathcal{U} \to \{1, \ldots, m\}$, *randomly selected from a family of two-universal hash functions* $\mathcal{H}$, *is a function such that for any two elements* $e_1, e_2 \in \mathcal{U}$, *it is*

$$\Pr\left[H(e_1) = H(e_2)\right] \leq \frac{1}{m}\,. \tag{3.1}$$

By using a two-universal hash function, hash tables can be constructed as follows.

- Set up an one-dimensional table $\mathbf{T}[1 \ldots m]$ where $m = O(n)$;
- Pick a *two-universal hash function* $H : \mathcal{U} \to \{1, \ldots, m\}$ as defined in Definition 3.1;
- Store element e in slot $\mathbf{T}[H(e)]$ of the table.

The probabilistic property that holds for hash function h implies that for any slot of the table, the expected number of elements mapped to it is $O(1)$. Also, if h can be computed in $O(1)$ time, looking-up an element has expected constant complexity.

But the above property of hash tables comes at some cost. The expected constant-complexity look-up holds when the number of elements stored in the hash table does not change, i.e., when the hash table is static. In particular, because of insertions, the number of elements stored in a slot may grow and we cannot assume anymore that is expected to be constant. A different problem arises in the presence of deletions as the number n of elements may become much smaller than the size m of the hash table. Thus, we may no longer assume that the hash table uses $O(n)$ space.

In order to deal with updates, we periodically update the size of the hash table by a constant factor (e.g., doubling or halving its size). This is an expensive operation since we

have to rehash all the elements. Therefore, there might be one update (over a course of $O(n)$ updates) that has $O(n)$ rather than $O(1)$ complexity. Thus, hash tables for dynamic sets typically have expected $O(1)$ query complexity and $O(1)$ expected *amortized* update complexity. Methods that vary the size of the hash table for the sake of maintaining $O(1)$ expected query complexity, fall into the general category of *dynamic hashing*. The above discussion is summarized in the following theorem:

Theorem 3.1 (Dynamic hashing [29]) *For a set of size n, dynamic hashing can be implemented to use $O(n)$ space and have $O(1)$ expected query complexity for (non-)membership queries and $O(1)$ expected amortized complexity for elements insertions or deletions.*

3.1.2 The RSA accumulator

We now give an overview of the RSA accumulator, which will be used for the construction of our first solution, i.e., the construction of the authenticated data structure scheme $\mathcal{RHT}$.

Prime representatives. For security and correctness reasons that will soon become clear, in our construction we extensively use the notion of *prime representatives* of elements. Initially introduced by Baric and Pfitzmann [10], prime representatives provide a solution whenever it is necessary to map general elements to prime numbers. In particular, one can map a k-bit element e_i to a $3k$-bit prime x_i using a two-universal hash function (see Definition 3.1). In our context, we are using a two-universal hash function $h : A \rightarrow B$, which is different than the one (i.e., $H(.)$) we use to map elements to buckets, and where set A is the set of $3k$-bit boolean vectors and B is the set of k-bit boolean vectors. Specifically, we use the two-universal hash function

$$h(x) = \mathbf{F}x\,,$$

where $\mathbf{F}$ is a $k \times 3k$ boolean matrix. Since the linear system $h(x) = \mathbf{F}x$ has more than one solution, one k-bit element is mapped to more than one $3k$-bit elements. We are interested

in finding only one such solution which is prime; this can be computed efficiently according to the following result:

Lemma 3.1 (Prime representatives [42, 51]) *Let $\mathcal{H}$ be a two-universal family of functions mapping $\{0,1\}^{3k}$ to $\{0,1\}^k$ and let $h \in \mathcal{H}$. For any element $e_i \in \{0,1\}^k$, we can compute with high probability a prime $x_i \in \{0,1\}^{3k}$ such that $h(x_i) = e_i$ by sampling $O(k^2)$ times from the set of inverses $h^{-1}(e_i)$.*

By Lemma 3.1, we have that computing prime representatives has expected constant complexity, i.e., independent of n. Also, solving the $k \times 3k$ linear system in order to compute the set of inverses has polynomial complexity in k by using standard methods (e.g., Gaussian elimination). Finally, we note that, in our context, prime representatives are computed and stored only once. Indeed, using the above method multiple times for computing the prime representative of the same element will not yield the same prime as output, for Lemma 3.1 describes a randomized process. From now on, given a k-bit element x, we denote with $r(x)$ the $3k$-bit prime representative that is computed as described by Lemma 3.1.

Description of the RSA accumulator. We now give an overview of the *RSA accumulator* [10, 13, 23, 68], which provides an efficient technique to produce a short (computational) proof that a certain element is (or is not) a member of a set. The RSA accumulator works as follows. Suppose we have the set of k-bit elements $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$. Let N be a k'-bit RSA modulus ($k' > 3k$), namely $N = pq$, where p, q are strong primes [23]. We can represent $\mathcal{X}$ compactly and securely with an *accumulation* value $\mathsf{acc}(\mathcal{X})$, which is a k'-bit integer, as follows

$$\mathsf{acc}(\mathcal{X}) = g^{r(x_1)r(x_2)\ldots r(x_n)} \mod N\,,$$

where $g \in \mathbb{QR}_N$ and $r(x_i)$ is a $3k$-bit prime representative, computed using a two-universal hash function $h(.)$. Note that the RSA modulus N, the exponentiation base g and the two-universal hash function comprise the public key pk, i.e., information that is available to the

adversary (the factorization of N is kept secret). Subject to the accumulation $\mathsf{acc}(\mathcal{X})$, every element x in set $\mathcal{X}$ has a *membership witness* $(\mathsf{W}_x, r(x), x)$, where

$$\mathsf{W}_x = g^{\prod_{x_j \in \mathcal{X} : x_j \neq x} r(x_j)} \mod N\,. \tag{3.2}$$

Membership of x in $\mathcal{X}$ is verified by means of the following tests:

1. Checking that $r(x)$ is a prime number;
2. Checking that $h(r(x)) = x$;
3. Computing $\mathsf{W}_x^{r(x)} \mod N$ and checking that this equals $\mathsf{acc}(\mathcal{X})$.

Moreover, subject to the accumulation $\mathsf{acc}(\mathcal{X})$, every element $x \notin \mathcal{X}$ has a *non-membership witness* as well [68], namely the integer values $(\mathsf{A}_x, \mathsf{B}_x, r(x), x)$ such that

$$\left(\prod_{i=1}^{n} r(x_i)\right) \mathsf{A}_x + r(x)\mathsf{B}_x = 1\,. \tag{3.3}$$

Note that A_x and B_x can be computed by running the extended Euclidean algorithm [102] on $r(x_1)r(x_2)\ldots r(x_n)$ and $r(x)$. Given the accumulation value $\mathsf{acc}(\mathcal{X})$ and the non-membership witness $(\mathsf{A}_x, \mathsf{B}_x, r(x), x)$, non-membership of x in $\mathcal{X}$ can be verified by means of the following tests:

1. Checking that $r(x)$ is a prime number;
2. Checking that $h(r(x)) = x$;
3. Computing $\mathsf{acc}(\mathcal{X})^{\mathsf{A}_x} g^{x\mathsf{B}_x} \mod N$ and checking that this equals g.

We finally note that the representation $\mathsf{acc}(\mathcal{X})$ has the crucial property that any computationally bounded adversary Adv who does not know $\phi(N)$ cannot find another set of elements $\mathcal{X}' \neq \mathcal{X}$ such that $\mathsf{acc}(\mathcal{X}') = \mathsf{acc}(\mathcal{X})$, unless Adv breaks the *the factoring assumption* [10]. However, in order to achieve some more advanced security goals we need, we are going to use a stronger assumption:

Assumption 3.1 (Strong RSA assumption) *Let k be the security parameter. Given a k-bit RSA modulus N and a random element $x \in \mathbb{Z}_N^*$, there is no polynomial-time algorithm that outputs $y > 1$ and a such that $a^y = x \mod N$, except with negligible probability $\mathsf{neg}(k)$.*

The security of our RSA-based solution is based on the following result. To assist the reader, we also recall the proof of the security results for membership proofs [10] and for non-membership proofs [68].

Lemma 3.2 (Security of the RSA accumulator [10, 68]) *Let k be the security parameter, h be a two-universal hash function mapping $3k$-bit integers to k-bit integers, N be a $(3k+1)$-bit RSA modulus and $g \in \mathbb{QR}_N$. Given N, g, a set of k-bit elements $\mathcal{X}$ and h, suppose there is a polynomial-time algorithm for one of the tasks below (or both):*

- *It outputs $x \notin \mathcal{X}$, W and prime r such that $h(r) = x$ and $\mathsf{W}^r = \mathsf{acc}(\mathcal{X}) \mod N$;*
- *It outputs $x \in \mathcal{X}$, A, B and prime r such that $h(r) = x$ and $\mathsf{acc}(\mathcal{X})^{\mathsf{A}} g^{r\mathsf{B}} = g \mod N$.*

Then there is a polynomial-time algorithm for breaking the strong RSA assumption.

Proof: Let $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ and let $x \notin \mathcal{X}$. For the membership proof, suppose there is an algorithm that finds W, r and x such that r is a prime number, $h(r) = x$ and

$$\mathsf{W}^r = g^{r(x_1)r(x_2)\ldots r(x_n)} \mod N\,.$$

Since $x \notin \mathcal{X}$, by construction of the prime representatives, it is $r \notin \{r(x_1), r(x_2), \ldots, r(x_n)\}$ (recall that $h(r) = x$). Let now $e = r$ and $R = r(x_1)r(x_2)\ldots r(x_n)$. The algorithm can now compute the e-th root of g as follows: It computes $a, b \in \mathbb{Z}$ such that $aR + br = 1$ by using the extended Euclidean algorithm, since r is a prime and $r \notin \{r(x_1), r(x_2), \ldots, r(x_n)\}$. Let now $y = \mathsf{W}^a g^b \mod N$. It is

$$y^e = \mathsf{W}^{ar} g^{br} = g^{aR+br} = g \mod N\,.$$

Therefore the algorithm can be used for breaking the strong RSA assumption. For the non-membership proof case, since $x \in \mathcal{X}$ the algorithm can output the e-th root of g as

$y = \mathsf{W}_x^{\mathsf{A}} g^{\mathsf{B}}$, where W_x is the membership witness defined in Relation 3.2. Then

$$y^e = \mathsf{W}_x^{e\mathsf{A}} g^{e\mathsf{B}} = \mathsf{acc}(\mathcal{X})^{\mathsf{A}} g^{r\mathsf{B}} = g \mod N\,.$$

This completes the proof. □

3.1.3 The bilinear-map accumulator

We next give an overview of the bilinear-map accumulator [83] which will be used for the construction of our second solution, i.e., the construction of the authenticated data structure scheme $\mathcal{BHT}$.

Bilinear pairings. Before presenting the bilinear-map accumulator we describe some basic terminology and definitions about *bilinear pairings*. Let $\mathbb{G}_1, \mathbb{G}_2$ be two cyclic multiplicative groups of prime order p, generated by g_1 and g_2 and for which there exists an isomorphism $\psi : \mathbb{G}_2 \rightarrow \mathbb{G}_1$ such that $\psi(g_2) = g_1$. Let also $\mathcal{G}$ be a cyclic multiplicative group with the same order p and $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathcal{G}$ be a bilinear pairing with the following properties:

1. Bilinearity: $e(P^a, Q^b) = e(P, Q)^{ab}$ for all $P \in \mathbb{G}_1$, $Q \in \mathbb{G}_2$ and $a, b \in \mathbb{Z}_p$;
2. Non-degeneracy: $e(g_1, g_2) \neq 1$;
3. Computability: There is an efficient algorithm to compute $e(P, Q)$ for all $P \in \mathbb{G}_1$ and $Q \in \mathbb{G}_2$.

In our setting we have $\mathbb{G}_1 = \mathbb{G}_2 = \mathbb{G}$ and $g_1 = g_2 = g$. A *bilinear pairing instance generator* is a probabilistic polynomial-time algorithm that takes as input the security parameter 1^k and outputs a uniformly random tuple $\mathsf{t} = (p, \mathbb{G}, \mathcal{G}, e, g)$ of bilinear pairings parameters.

Here we have to make an important observation: Groups $\mathbb{G}$ and $\mathcal{G}$ are generic. That is, their elements are not simple integers and doing operations between elements can be complicated. E.g., group elements of $\mathbb{G}$ and $\mathcal{G}$ (for which there exist efficient constructions of a bilinear map $e(.,.)$) are usually points on an elliptic curve. Also the operations in the

exponent of elements of $\mathbb{G}$ and $\mathcal{G}$ are performed modulo p, since this is the order of both groups $\mathbb{G}$ and $\mathcal{G}$. A simplified exposition of these groups and their arithmetic is given in the book of Katz and Lindell [61].

Description of the bilinear-map accumulator. Similarly with the RSA accumulator, the bilinear-map accumulator [32, 83] comprises an efficient way to provide short proofs of (non-)membership for elements that (do not) belong to a set. The bilinear-map accumulator works as follows. Let $s \in \mathbb{Z}_p^*$ is a randomly chosen value that constitutes the trapdoor in the scheme (in the same way that $\phi(N)$ was the trapdoor in the RSA accumulator). The accumulator accumulates elements in $\mathbb{Z}_p^* - \{-s\}$ (where p is a k-bit prime) and the accumulated value is an element in $\mathbb{G}$. Given a set of n elements $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ the accumulation value $\mathsf{acc}(\mathcal{X})$ is defined as

$$\mathsf{acc}(\mathcal{X}) = g^{(x_1+s)(x_2+s)\ldots(x_n+s)} ,$$

where g is a generator of group $\mathbb{G}$ of prime order p. We note here that $\mathsf{acc}(\mathcal{X})$ can be constructed by only using $\mathcal{X}$ and $g, g^s, g^{s^2}, \ldots, g^{s^q}$, where $q \geq |\mathcal{X}|$, by using polynomial interpolation (see Lemma 3.15). The proof of membership for an element x that belongs to set $\mathcal{X}$ will be the witness (W_x, x) where

$$\mathsf{W}_x = g^{\prod_{x_j \in \mathcal{X}: x_j \neq x} (x_j+s)} . \tag{3.4}$$

Accordingly, a verifier can test set membership for x by computing $e(\mathsf{W}_x, g^s g^x)$ and checking that this equals $e(\mathsf{acc}(\mathcal{X}), g)$.

Moreover, subject to the accumulation $\mathsf{acc}(\mathcal{X})$, every element $x \notin \mathcal{X}$ has a *non-membership witness* [32], namely the elements in $(\mathsf{A}_x = g^{\alpha(s)}, \mathsf{B}_x = g^{\beta(s)}, x)$ such that

$$\left[\prod_{i=1}^{n}(x_i + s)\right] \alpha(s) + (x + s)\beta(s) = 1 . \tag{3.5}$$

Note that $\alpha(s)$ and $\beta(s)$ are polynomials that can be computed by running the extended Euclidean algorithm on polynomials $(x_1 + s)(x_2 + s)\ldots(x_n + s)$ and $(x + s)$. Given the

accumulation value $\mathsf{acc}(\mathcal{X})$ and the witnesses A_x and B_x, non-membership of x in $\mathcal{X}$ can be verified by computing $e(\mathsf{acc}(\mathcal{X}), \mathsf{A}_x)e(g^s g^x, \mathsf{B}_x)$ and checking that this equals $e(g, g)$.

Proving the security of the bilinear-map accumulator requires the *bilinear* q-strong Diffie-Hellman assumption, a *slightly* stronger assumption than the q-strong Diffie-Hellman assumption [16][1], that can be stated as follows:

Assumption 3.2 (Bilinear q-strong Diffie-Hellman assumption) *Let k be the security parameter and let $(p, \mathbb{G}, \mathcal{G}, e, g)$ be a uniformly randomly generated tuple of bilinear pairings parameters. Given the elements $g, g^s, \ldots, g^{s^q} \in \mathbb{G}$ for some s chosen at random from $\mathbb{Z}_p^*$, where $q = \mathsf{poly}(k)$, there is no polynomial-time algorithm that can output the pair $(a, e(g, g)^{1/(s+a)}) \in \mathbb{Z}_p \times \mathcal{G}$ except with negligible probability $\mathsf{neg}(k)$.*

Lemma 3.3 (Security of the bilinear-map accumulator [32, 83]) *Let k be the security parameter and let $\mathsf{t} = (p, \mathbb{G}, \mathcal{G}, e, g)$ be a uniformly randomly generated tuple of bilinear pairings parameters. Given the elements $g, g^s, \ldots, g^{s^q} \in \mathbb{G}$ for some s chosen at random from $\mathbb{Z}_p^*$ and a set of k-bit elements $\mathcal{X}$ ($q \geq |\mathcal{X}|$), suppose there is a polynomial-time algorithm for one of the tasks below (or both):*

- *It outputs $x \notin \mathcal{X}$ and W such that $e(\mathsf{W}, g^s g^x) = e(\mathsf{acc}(\mathcal{X}), g)$;*
- *It outputs $x \in \mathcal{X}$, A and B such that $e(\mathsf{acc}(\mathcal{X}), \mathsf{A})e(g^s g^x, \mathsf{B}) = e(g, g)$.*

Then there is a polynomial-time algorithm for breaking the bilinear q-strong Diffie-Hellman assumption.

Proof: Let $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ and let $x \notin \mathcal{X}$. Suppose there is an algorithm that finds W such that $e(\mathsf{W}, g^s g^x) = e(\mathsf{acc}(\mathcal{X}), g)$. This implies

$$e(\mathsf{W}, g)^{s+x} = e(g, g)^{(s+x_1)(s+x_2)\ldots(s+x_n)}.$$

[1] However, proving just *collision resistance* of the accumulator requires the plain q-strong Diffie-Hellman assumption [83].

Note now that the quantity

$$\Pi_n = (s + x_1)(s + x_2) \dots (s + x_n)$$

can be viewed as a polynomial in s of degree n. Since $x \notin \mathcal{X}$, we have that $(s + x)$ does not divide Π_n and therefore values c and P can be computed such that $\Pi_n = c + P(s + x)$. Therefore the algorithm can output $(x, e(g, g)^{1/(s+x)})$ as

$$\left(x, \left[e(\mathsf{W}, g)e(g, g)^{-P}\right]^{c^{-1}}\right) .$$

For a non-membership proof, note that we can output $e(g, g)^{1/(s+x)}$ as

$$e(\mathsf{W}_x, \mathsf{A})e(g, \mathsf{B}) ,$$

since $x \in \mathcal{X}$ and $e(\mathsf{acc}(\mathcal{X}), \mathsf{A})e(g^s g^x, \mathsf{B}) = e(g, g)$, where W_x is a membership witness given in Relation 3.4. Therefore the bilinear q-strong Diffie-Hellmann assumption can be broken in both cases. □

Size of non-membership witnesses. We note here that although non-membership witnesses are constructed in the same fashion in both instantiations of the accumulator schemes, their sizes differ considerably. In the RSA accumulator case, the integers A_x and B_x (see Relation 3.3) can have size proportional to the number of the elements in $\mathcal{X}$. In the bilinear-map accumulator case, A_x and B_x (see Relation 3.5) are always *two* group elements in $\mathbb{G}$ (this takes advantage of the bilinear map $e(.,.)$, which is not known to exist for RSA groups), therefore their size never depends on the elements collection $\mathcal{X}$. This observation is very important and will contribute to significant complexity improvements in Chapter 5.

We now continue with some necessary algorithmic and definitional framework. We present an algorithmic construction called *accumulation tree* that will be used in both our constructions.

3.1.4 The accumulation tree

Let $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ be a set of elements. Given a constant $\epsilon < 1$ such that $0 < \epsilon < 1$, the *accumulation tree* of $\mathcal{X}$, denoted with $T(\epsilon)$, is a rooted tree with n leaves defined as follows:

1. The leaves of $T(\epsilon)$ store the elements $x_1, x_2, \ldots, x_n$;
2. $T(\epsilon)$ consists of exactly $l = \lceil \frac{1}{\epsilon} \rceil$ levels;
3. All the leaves are at the same level;
4. Every node of $T(\epsilon)$ has $O(n^{\epsilon})$ children;
5. Level i in the tree contains $O(n^{1-i\epsilon})$ nodes, where the leaves are at level 0 and the root is at level l.

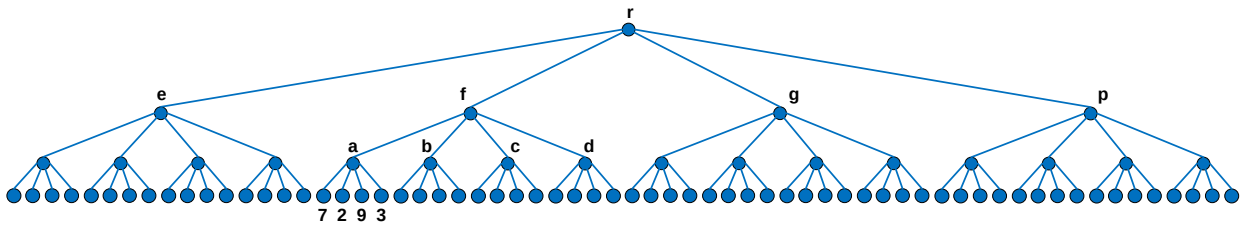

Figure 3.1: The accumulation tree of a set of 64 elements for $\epsilon = \frac{1}{3}$: every internal node has $4 = 64^{\frac{1}{\epsilon}}$ children, there are $3 = \frac{1}{\epsilon}$ levels in total, and there are $64^{1-i/3}$ nodes at level $i = 0, 1, 2, 3$.

We note that the levels of the accumulation tree are numbered from the leaves to the root of the tree, i.e., the leaves have level 0, their parents level 1 and finally the root has level l. The structure of the accumulation tree, which for a set of 64 elements is shown in Figure 3.1, resembles that of normal "flat" search trees, in particular, the structure of a B-tree [29]. However there are some differences: First, every internal node of the accumulation tree, instead of having a constant upper bound on its degree, it has a bound that is a function of the number of its leaves, n; also, its depth is always maintained to be constant, namely

$O\left(\frac{1}{\epsilon}\right)$. Note that it is simple to construct the accumulation tree when n^ϵ is an integer (see Figure 3.1). Else, we define the accumulation tree to be the unique tree of degree $\lceil n^\epsilon \rceil$ (by assuming a certain ordering of the leaves). This maintains the degree of internal nodes to be $O(n^\epsilon)$.

Using the accumulation tree and search keys stored at the internal nodes, one can search for an element in $O(n^\epsilon)$ time and perform updates in $O(n^\epsilon)$ *amortized* time. Indeed, as the depth of the tree is not allowed to vary, one should periodically (e.g., when the number of elements of the tree doubles) rebuild the tree spending $O(n)$ time. Actually, by using individual binary trees to index the search keys within each internal node, queries could be answered in $O(\log n)$ time and updates could be processed in $O(\log n)$ amortized time. Yet, the reason we build this flat tree is not to use it as a search structure, but rather to design an authentication structure for defining the digest of $\mathcal{X}$ that matches the *optimal querying performance* of hash tables. The idea is as follows: we wish to hierarchically employ an accumulator over the subsets (of accumulation values) defined by each internal node in the accumulation tree, so that (non)-membership proofs of size proportional to the depth of the tree (hence of constant size) are defined with respect the root digest (accumulation value of the entire set).

3.2 Scheme based on the RSA accumulator

In this section we present a secure authenticated data structure scheme for an authenticated hash table $\mathcal{RHT} = \{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$ and prove it satisfies the complexities of Table 3.1. For each algorithm, we are going to describe two constructions, i.e., the "plain" construction and the one with "precomputed witnesses".

Algorithm $\{\mathsf{sk}, \mathsf{pk}\} \leftarrow \mathsf{genkey}(1^k)$**:** The algorithm picks a constant $0 < \epsilon < 1$ and $\lceil 1/\epsilon \rceil + 1$ RSA moduli $N_i = p_i q_i$ $(i = 0, \ldots, l)$, where p_i, q_i are strong primes [23] and $l = \lceil 1/\epsilon \rceil$. The

length of the RSA moduli is defined by the recursive relations

$$|N_{i+1}| = 3|N_i| + 1\,,$$

where $|N_0| = 3k+1$ and $i = 0, \ldots, l-1$. The algorithm also picks $l+1$ public bases $g_i \in \mathbb{QR}_{N_i}$ to be used for exponentiation. Finally, given $l+1$ families of two-universal hash functions $\mathcal{H}_0, \mathcal{H}_2, \ldots, \mathcal{H}_l$, the algorithm randomly picks one function $h_i \in \mathcal{H}_i$, for $i = 0, \ldots, l$ (h_i will be used for computing prime representatives). The function h_i is such that it maps $(|N_i|-1)$-bit primes to $((|N_i|-1)/3)$-bit integers[2]. The algorithm sets $\mathsf{sk} = \{\phi(N_i) = (p_i-1)(q_i-1) : i = 0, \ldots, l\}$ and $\mathsf{pk} = \{N_i, g_i, h_i : i = 0, \ldots, l; \epsilon\}$. Note that since l is constant all RSA moduli have size that only depends on the security parameter k. Also, since $1/\epsilon$ is constant, the algorithm has access complexity $O(1)$.

3.2.1 Main authenticated data structure

Let $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ be a collection of n elements. $\mathcal{X}$ is stored in a dynamic hash table D_0 by using a two-universal hash function $H(.)$ that maps each element to a certain bucket (see Theorem 3.1). Specifically, the hash table D_0 has $m = O(n)$ buckets $L_1, L_2, \ldots, L_m$, where each bucket contains $O(1)$ elements *in expectation* (by the property of the two-universal hash function—see Relation 3.1). Let now $0 < \epsilon < 1$ be the fixed constant chosen by algorithm $\mathsf{setup}()$. We build the accumulation tree $T(\epsilon)$ on top of the buckets, i.e., every leaf of the tree corresponds to a specific bucket and *not* to an element within the bucket. Since the number of buckets is $m = O(n)$, the internal nodes of the accumulation tree have $O(n^\epsilon)$ children.

Our authenticated data structure is defined with respect to the accumulation tree as follows. We hierarchically employ the RSA accumulator over the buckets of the hash table so that to augment the accumulation tree with a collection of corresponding *accumulation*

[2]The choice of the domains and ranges of functions h_i and of the lengths of moduli N_i is due to the requirement that prime representatives should be smaller numbers than the respective moduli (see [101]). As we will see in Section 3.2.5, using ideas from [10] it is possible to avoid the increasing size of the RSA moduli and instead use only one size for all N_i's. By doing so, however, we are forced to prove security in the random oracle model (using cryptographic hash functions), which is fine for practical applications.

values. That is, assuming the setup parameters are in place, for any node v in the accumulation tree we define its accumulation value $\chi(v)$ recursively along the tree structure, as a function of the accumulation value of its children (in a similar way as in a Merkle tree). We describe algorithm $\mathsf{setup}()$ in detail below:

Algorithm $\{\mathsf{auth}(D_0), d_0\} \leftarrow \mathsf{setup}(D_0, \mathsf{sk}, \mathsf{pk})$**:** The algorithm builds the accumulation tree $T(\epsilon)$ on top of the m buckets $L_1, L_2, \ldots, L_m$. For every leaf node v in tree $T(\epsilon)$ that lies at level 0 and corresponds to a bucket L_j, the algorithm sets

$$\chi(v) = g_0^{\prod_{x \in L_j} r_0(x)} \mod N_0 \in \mathbb{Z}^*_{N_0}\,. \tag{3.6}$$

For every non-leaf node v in $T(\epsilon)$ that lies at level $1 \leq i \leq l$, the algorithm sets:

$$\chi(v) = g_i^{\prod_{u \in \mathcal{N}(v)} r_i(\chi(u))} \mod N_i \in \mathbb{Z}^*_{N_i}\,, \tag{3.7}$$

where $r_i(a)$ is a prime representative of a computed using function h_i, $\mathcal{N}(v)$ is the set of children of node v (when node v refers to a bucket, i.e., it is a leaf, we define as v's children to be the elements contained in the bucket) and $g_i \in \mathbb{QR}_{N_i}$. The authenticated data structure $\mathsf{auth}(D_0)$ output by the algorithm consists of the following components:

1. The accumulation tree $T(\epsilon)$;
2. The prime representatives $r_i(\chi(v))$ that correspond to the values $\chi(v)$, such that $h_i(r_i(\chi(v))) = \chi(v)$—as used in Relations 3.6 and 3.7, for all nodes $v \in T(\epsilon)$ (at some level i).

Let r be the root of the tree T. The algorithm also outputs $d_0 = \chi(r)$, i.e., the digest of the authenticated data structure is the $\chi(.)$ value of the root of the accumulation tree.

Precomputed witnesses. In order to achieve constant-complexity queries, algorithm $\mathsf{setup}()$ can also compute *precomputed* witnesses. Namely, for every node v of the accumulation tree that lies at level $0 \leq i \leq l$, let $\mathcal{N}(v)$ be the set of its children (for a leaf node,

we consider as "children" the elements in the respective bucket). For every $j \in \mathcal{N}(v)$ the algorithm computes

$$\mathsf{W}_{j(v)} = \chi(v)^{r_i(\chi(j))^{-1}} = g_i^{\prod_{u \in \mathcal{N}(v)-\{j\}} r_i(\chi(u))} \mod N_i\,, \tag{3.8}$$

and stores $\mathsf{W}_{j(v)}$ at v. When the construction with precomputed witnesses is used, $\mathsf{auth}(D_0)$ also includes $\mathsf{W}_{j(v)}$, for all $v \in T(\epsilon)$ and all $j \in \mathcal{N}(v)$, along with $T(\epsilon)$ and $r_i(\chi(v))$.

Lemma 3.4 *Algorithm* $\mathsf{setup}()$ *of the authenticated data structure scheme* $\mathcal{RHT}$ *has* $O(n)$ *access complexity both with and without precomputed witnesses. Moreover, the authenticated data structure* $\mathsf{auth}(D_0)$ *output by* $\mathsf{setup}()$ *has* $O(n)$ *group complexity.*

Proof: For a node v that has degree d, computing $\chi(v)$ from Relation 3.7 has $O(d)$ access complexity. At level $i \geq 1$, there are $O(m^{1-i\epsilon})$ such nodes, of degree $O(m^{\epsilon})$, where m is the number of the buckets (at level 0 there are m nodes of constant degree). Since $m = O(n)$ and $T(\epsilon)$ has $O(1)$ levels, the access complexity without precomputed witnesses is $O(n)$. For a node v at level i that has degree d, computing $\mathsf{W}_{j(v)}$ for all $j \in \mathcal{N}(v)$ from Relation 3.8 has $O(d)$ access complexity: Compute $\chi(v)$ first, and then set

$$\mathsf{W}_{j(v)} = \chi(v)^{r_i(\chi(j))^{-1}} \mod N_i\,.$$

Note that the computation of the inverse in the exponent is feasible because $\mathsf{setup}()$ has access to the secret key, that contains the factorization $\phi(N_0)$ (we will see that this computational task requires more work when the factorization is not available). Therefore the access complexity of $\mathsf{setup}()$ with precomputed witnesses is also $O(n)$, since computing one such witness requires $O(1)$ work and there are $O(n)$ such witnesses. Finally, every node of $T(\epsilon)$ stores one group element (and two group elements in the precomputed witnesses case). Since the tree $T(\epsilon)$ has $O(n)$ nodes, the group complexity of $\mathsf{auth}(D_0)$ is $O(n)$. □

3.2.2 Updates

We now describe how updates can be efficiently supported in the authenticated hash table scheme, by using a rebuilding technique, appropriately adjusted from the book of Cormen

et al. [29]: Since a hash table with $m = O(n)$ buckets is used, we should expect that at some point the update algorithms will need to rebuild the table (i.e., rehash all the elements and reinsert them in a bigger or smaller hash table) *and* the related authenticated data structures. This is done according to the following definition:

Definition 3.2 *Let m be the current number of buckets of the authenticated hash table and n be the number of elements contained in the authenticated hash table after an update has been performed. Define $\alpha = \frac{n}{m}$ to be the load factor of the authenticated hash table after the update. If $\alpha = 1$ (full table) the capacity of the hash table is doubled. If $\alpha = \frac{1}{4}$ (near empty table) the capacity of the hash table is halved.*

The rebuilding method described in Definition 3.2, adjusted to our authenticated hash table construction, is essential to get the necessary amortized results of Lemmata 3.5 and 3.7, which constitutes the main complexity results of this work (for similar methods see the book of Cormen et al. [29]). We describe now algorithms update() and refresh() in detail.

Algorithm $\{D_{h+1}, \mathsf{auth}(D_{h+1}), d_{h+1}, \mathsf{upd}\} \leftarrow \mathsf{update}(u, D_h, \mathsf{auth}(D_h), d_h, \mathsf{sk}, \mathsf{pk})$**:** Let m be the current number of buckets of D_h and n be the number of elements stored in D_h, *after* the update has been performed. We distinguish two cases:

Case 1. $\frac{m}{4} < n < m$**:** In this case there is no need to rebuild the table and the update is performed as follows: Suppose the update is "insert element e". The algorithm computes the bucket $j = H(e)$ (see Relation 3.1) and inserts e in bucket j. Let v_0 be the node of $T(\epsilon)$ referring to bucket j and $r_0(e)$ be a new prime representative for element e computed using function h_0, i.e., $h_0(r_0(e)) = e$. Let $v_0, v_1, \ldots, v_l$ be the path in $T(\epsilon)$ from node v_0 to the root of the tree. The algorithm initially sets

$$\chi'(v_0) = \chi(v_0)^{r_0(e)} \mod N_0\,,$$

i.e., it updates the accumulation value that corresponds to the updated bucket. Note that

if the update is "delete element e", the algorithm sets

$$\chi'(v_0) = \chi(v_0)^{r_0(e)^{-1}} \mod N_0 \,. \tag{3.9}$$

Subsequently, for $j = 1, \ldots, l$ the algorithm sets

$$\chi'(v_j) = \chi(v_j)^{r_j(\chi'(v_{j-1}))r_j(\chi(v_{j-1}))^{-1}} \mod N_j \,, \tag{3.10}$$

where $r_j(\chi(v_{j-1}))$ is the prime representative of accumulation value $\chi(v_{j-1})$ and $r_j(\chi'(v_{j-1}))$ is a new prime representative for the updated accumulation value $\chi'(v_{j-1})$, such that

$$h_j(r_j(\chi'(v_{j-1}))) = \chi'(v_{j-1}) \,.$$

All these values are stored by the algorithm after they have been computed. The algorithm also outputs the new prime representatives $r_j(\chi'(v_{j-1}))$ $(j = 1, \ldots, l)$ as the information upd along the path from the updated bucket to the root of the tree. Information upd also includes $r_0(e)$ and $\chi'(v_l)$. Also it sets $d_{h+1} = \chi'(v_l)$, i.e., the updated digest is the updated $\chi(.)$ value of the root of $T(\epsilon)$. Finally the new authenticated data structure $\mathsf{auth}(D_{h+1})$ is computed as follows. Let $\mathsf{auth}(D_h)$ be the previous authenticated data structure that is input to the algorithm. Overwrite the values $r_j(\chi(v_{j-1}))$ $(j = 1, \ldots, l)$ with the new values $r_j(\chi'(v_{j-1}))$ $(j = 1, \ldots, l)$ and output the updated structure. The behavior of the algorithm in the *precomputed witnesses* case is the same with the difference that $\mathsf{upd} = \emptyset$.

Case 2. $n = \frac{m}{4}$ **or** $n = m$**:** In this case the hash table is rebuilt according to Definition 3.2: If $n = \frac{m}{4}$, then the algorithm builds a data structure D_{h+1} with $m/2$ buckets. Otherwise, i.e., when $n = m$, the algorithm builds a data structure D_{h+1} with $2m$ buckets. Subsequently, it outputs $\mathsf{auth}(D_{h+1})$ and d_{h+1} by calling algorithm $\mathsf{setup}(D_{h+1}, \mathsf{sk}, \mathsf{pk})$ and sets $\mathsf{upd} = \emptyset$.

Lemma 3.5 *By using the rebuilding policy of Definition 3.2, algorithm* $\mathsf{update}()$ *of the authenticated data structure scheme* $\mathcal{RHT}$ *has* $O(1)$ *expected amortized access complexity. Moreover, the update information* upd *output by* $\mathsf{update}()$ *has* $O(1)$ *group complexity.*

Proof: The $O(1)$ expected complexity bound comes from the fact that the number of operations (as long as the group elements contained in upd) that $\mathsf{update}()$ performs is always

a function of $l = 1/\epsilon = O(1)$—also the actual hash table update is performed, which has expected $O(1)$ complexity. Note that the complexity of the operations in Relations 3.9 and 3.10 is constant since sk contains the factorizations $\phi(N_i)$ and therefore inverses can be computed with the extended Euclidean algorithm. When the hash table has to be rebuilt, algorithm $\mathsf{setup}()$ is called, which has $O(n)$ access complexity (see Lemma 3.4). Therefore the result is expected amortized since we can use the rebuilding strategy in Definition 3.2 and follow the same amortized analysis from [29] (i.e., the cost of rebuilding in [29] does not increase due to the $O(n)$ complexity of $\mathsf{setup}()$). $\square$

Algorithm $\{D_{h+1}, \mathsf{auth}(D_{h+1}), d_{h+1}\} \leftarrow \mathsf{refresh}(u, D_h, \mathsf{auth}(D_h), d_h, \mathsf{upd}, \mathsf{pk})$**:** Let m be the current number of buckets of D_h and n be the number of elements stored in D_h, *after* the update has been performed. We distinguish two cases:

Case 1. $\frac{m}{4} < n < m$**:** Suppose the update is "insert element e". The algorithm computes the bucket $j = H(e)$ (see Relation 3.1) and inserts e in bucket j. Let v_0 be the node of $T(\epsilon)$ referring to bucket j. Let $v_0, v_1, \ldots, v_l$ be the path in $T(\epsilon)$ from node v_0 to the root of the tree. The algorithm, for $j = 0, \ldots, l$, sets

$$r_j(\chi(v_j)) = r_j(\chi'(v_j))\,,$$

i.e., it updates the prime representatives that correspond to the updated path by using the information upd[3]. Finally it outputs the updated hash table as D_{h+1}, the updated prime representatives $r_j(\chi(v_j))$ (along with the ones that belong to the nodes that are not updated) as $\mathsf{auth}(D_{h+1})$ and $\chi'(v_l)$ (contained in upd) as d_{h+1}.

Precomputed witnesses. When precomputed witnesses are used, the algorithm should update $\mathsf{W}_{j(v)}$ for $v = v_0, v_1, \ldots, v_l$ and for all $j \in \mathcal{N}(v)$ (see Relation 3.8). To achieve that

[3]Note that information upd is not *required* for $\mathsf{refresh}()$ to perform this task. Algorithm $\mathsf{refresh}()$ uses upd for efficiency. Namely, algorithm $\mathsf{refresh}()$ could compute the updated values $r_j(\chi(v_j))$ by performing explicit exponentiations, which would have $O(n^\epsilon)$ complexity.

efficiently, the following result from Sander et al. [101] for efficiently maintaining updated precomputed witnesses is used:

Lemma 3.6 (Computing witnesses [101]) *Suppose we are given the elements collection* $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$, *an RSA modulus* N *and* $g \in \mathbb{QR}_N$. *Without the knowledge of* $\phi(N)$, *the witnesses* $\mathsf{W}_i = g^{\prod_{j \neq i} x_j} \mod N$ *for* $i = 1, \ldots, n$ *can be computed with* $O(n \log n)$ *complexity.*

In order to compute the updated witnesses, the algorithm uses the algorithm from Sander et al. [101] that provides the above result for all nodes v_i, $0 \leq i \leq l$, and for all $j \in \mathcal{N}(v_i)$, as follows. For each v_i, it uses the result from Lemma 3.6 with inputs the *updated* elements $\{r_i(\chi(j)) : j \in \mathcal{N}(v_i)\}$, the RSA modulus N_i and the exponentiation base g_i. In this computation the updated prime representative $r_i(\chi(v_{i-1}))$, computed with $O(n^\epsilon)$ exponentiations, is used (note that $O(n^\epsilon)$ exponentiations are required since sk is not available). This computation outputs the witnesses $\mathsf{W}_{j(v_i)}$ for $j \in \mathcal{N}(v_i)$ (note that the witness $\mathsf{W}_{v_{i-1}(v_i)}$, for $i > 0$, remains the same). Also, since the algorithm for updating witnesses [101] is run on $O(1/\epsilon)$ nodes v with $|\mathcal{N}(v)| = O(n^\epsilon)$, we have, by Lemma 3.6, that the witnesses update complexity is $O(n^\epsilon \log n)$ (for the complete result see Lemma 3.7).

Case 2. $m = \frac{m}{4}$ **or** $n = m$**:** In this case the hash table is rebuilt according to Definition 3.2: If $n = \frac{m}{4}$, then the algorithm builds a data structure D_{h+1} with $m/2$ buckets. Otherwise, i.e., when $n = m$, the algorithm builds a data structure D_{h+1} with $2m$ buckets. Subsequently, it outputs $\mathsf{auth}(D_{h+1})$ and d_{h+1} by using Relations 3.6 and 3.7. In the case of precomputed witnesses, it computes the new witnesses to be included in $\mathsf{auth}(D_{h+1})$ by using Lemma 3.6 (note that $\mathsf{refresh}()$ cannot call $\mathsf{setup}()$ directly since it does not have access to the secret key sk and that is why it has to use Relations 3.6, 3.7 and Lemma 3.6).

Lemma 3.7 *By using the rebuilding policy of Definition 3.2, algorithm* $\mathsf{refresh}()$ *of the authenticated data structure scheme* $\mathcal{RHT}$ *has* $O(1)$ *expected amortized access complexity, without precomputed witnesses. With precomputed witnesses, algorithm* $\mathsf{refresh}()$ *has* $O(n^\epsilon \log n)$ *expected amortized access complexity.*

Proof: For the case when no precomputed witnesses are used, the argument is the same as in Lemma 3.5. For the case of precomputed witnesses, suppose there are currently n elements in the hash table and that the capacity of the table (i.e., number of buckets) is m. Note that, by the rebuilding policy of Definition 3.2, it is $m/4 < n < m$. As we know, each one of the m buckets stores $O(1)$ elements in expectation. When an update takes place and no rebuilding of the table is triggered, all the witnesses along the path of the update of the accumulation tree have to be updated. By using the algorithm described in Lemma 3.6, the witnesses within the bucket can be updated in expected complexity $O(1)$, since the size of the bucket is an expected value. The witnesses of the internal nodes can be updated in $O(m^{\epsilon} \log m)$ complexity and therefore the overall complexity is $O(m^{\epsilon} \log m)$ in expectation. When a rebuilding of the table is triggered then the total complexity is $O(m \log m)$, since there is a constant number of levels in the accumulation tree, processing each node has complexity $O(m^{\epsilon} \log m)$ (since the degree of any internal node is $O(m^{\epsilon})$) and the maximum number of nodes that lie in any level is $O(m^{1-\epsilon})$. Therefore, the *actual complexity* of an update is *expected* $O(m^{\epsilon} \log m)$, when no rebuilding is triggered and $O(m \log m)$ otherwise. We are interested in the expected value of the amortized complexity (expected amortized complexity) of an update. Let n_i be the number of elements contained in the hash table after update i and m_i be the number of buckets after update i. We do the analysis by defining the following potential function:

$$F_i = \begin{cases} c(2n_i - m_i) \log m_i, & \alpha_i \geq \frac{1}{2} \\ c\left(\frac{m_i}{2} - n_i\right) \log m_i, & \alpha_i < \frac{1}{2} \end{cases},$$

where $\alpha_i = \frac{n_i}{m_i}$. The amortized complexity for an update i will be equal to $\hat{\gamma}_i = \gamma_i + F_i - F_{i-1}$. Therefore $\mathbb{E}[\hat{\gamma}_i] = \mathbb{E}[\gamma_i] + F_i - F_{i-1}$, since F_i is a deterministic function. To perform the analysis more precisely we define some constants. Let c_1 be that constant such that if the update complexity C is $O(m_i^{\epsilon} \log m_i)$, it is

$$C \leq c_1 m_i^{\epsilon} \log m_i \,. \tag{3.11}$$

Also, let r_1 be that constant such that if the rebuilding complexity R is $O(n_i \log n_i)$, it is

$$R \leq r_1 n_i \log n_i \,. \tag{3.12}$$

Also we note that in all cases it holds

$$\frac{m_i}{4} \leq n_i \leq m_i \,. \tag{3.13}$$

We perform the analysis by distinguishing the following cases:

1. $\alpha_{i-1} \geq \frac{1}{2}$ (insertion). For this case, we examine the cases where the hash table is rebuilt or not. In case the hash table is not rebuilt, we have $m_{i-1} = m_i$ and $n_i = n_{i-1} + 1$. Therefore the amortized complexity will be:

$$\begin{aligned}
\mathbb{E}[\hat{\gamma}_i] &= \mathbb{E}[\gamma_i] + F_i - F_{i-1} \\
&\leq c_1 m_i^{\epsilon} \log m_i + c(2n_i - m_i - 2n_{i-1} + m_{i-1}) \log m_i \\
&= c_1 m_i^{\epsilon} \log m_i + 2c \log m_i \,.
\end{aligned}$$

   In case the hash table is rebuilt (which takes $O(n \log n)$ complexity in total) we have $m_i = 2m_{i-1}$, $n_i = n_{i-1} + 1$ and $n_{i-1} = m_{i-1}$ (which give $n_i = m_i/2 + 1 \leq m_i/2$) and the amortized complexity will be:

$$\begin{aligned}
\mathbb{E}[\hat{\gamma}_i] &= \mathbb{E}[\gamma_i] + F_i - F_{i-1} \\
&\leq r_1 n_i \log n_i + c(2n_i - m_i) \log m_i - c(2n_{i-1} - m_{i-1}) \log m_{i-1} \\
&= r_1 n_i \log n_i + c(2n_i - m_i) \log m_i - c\frac{m_i}{2} \log m_i/2 \\
&\leq r_1 \frac{m_i}{2} \log m_i/2 + 2c \log m_i - c\frac{m_i}{2} \log m_i/2 \\
&\leq 2c \log m_i
\end{aligned}$$

   for a constant c of the potential function such that $c > r_1$.

2. $\alpha_{i-1} < \frac{1}{2}$ (insertion). Note that that there is no way that the hash table is rebuilt in this case. Therefore $m_{i-1} = m_i$ and $n_i = n_{i-1} + 1$. If now $\alpha_i < \frac{1}{2}$ the amortized

complexity will be:

$$\begin{aligned}
\mathbb{E}[\hat{\gamma}_i] &= \mathbb{E}[\gamma_i] + F_i - F_{i-1} \\
&\leq c_1 m_i^\epsilon \log m_i + c(m_i/2 - n_i)\log m_i - c(m_{i-1}/2 - n_{i-1})\log m_{i-1} \\
&= c_1 m_i^\epsilon \log m_i + c(m_i/2 - n_i - m_i/2 + n_{i-1})\log m_i \\
&= c_1 m_i^\epsilon \log m_i - c \log m_i \,.
\end{aligned}$$

In case now $\alpha_i \geq \frac{1}{2}$ the amortized complexity will be:

$$\begin{aligned}
\mathbb{E}[\hat{\gamma}_i] &= \mathbb{E}[\gamma_i] + F_i - F_{i-1} \\
&\leq c_1 m_i^\epsilon \log m_i + c(2n_i - m_i)\log m_i - c(m_{i-1}/2 - n_{i-1})\log m_{i-1} \\
&= c_1 m_i^\epsilon \log m_i + c(2(n_{i-1}+1) - m_{i-1} - m_{i-1}/2 + n_{i-1})\log m_i \\
&= c_1 m_i^\epsilon \log m_i + c(3n_{i-1} - 3m_{i-1}/2 + 2)\log m_i \\
&= c_1 m_i^\epsilon \log m_i + c(3\alpha m_{i-1} - 3m_{i-1}/2 + 2)\log m_i \\
&< c_1 m_i^\epsilon \log m_i + c(3m_{i-1}/2 - 3m_{i-1}/2 + 2)\log m_i \\
&= c_1 m_i^\epsilon \log m_i + 2c \log m_i \,.
\end{aligned}$$

3. $\alpha_{i-1} < \frac{1}{2}$ (deletion). Here we have $n_i = n_{i-1} - 1$. In case the hash table does not have to be rebuilt (i.e., $\frac{1}{4} < \alpha_i < \frac{1}{2}$ and $m_i = m_{i-1}$), we have that the amortized complexity of the deletion is going to be:

$$\begin{aligned}
\mathbb{E}[\hat{\gamma}_i] &= \mathbb{E}[\gamma_i] + F_i - F_{i-1} \\
&\leq c_1 m_i^\epsilon \log m_i + c(m_i/2 - n_i)\log m_i - c(m_{i-1}/2 - n_{i-1})\log m_{i-1} \\
&= c_1 m_i^\epsilon \log m_i + c(m_i/2 - n_i - m_i/2 + n_{i-1})\log m_i \\
&= c_1 m_i^\epsilon \log m_i + c \log m_i \,.
\end{aligned}$$

In case now the hash table has to be rebuilt (which has $O(n_i \log n_i)$ complexity), we

have that $m_i = m_{i-1}/2$, $m_i = 4n_i$ and therefore the amortized complexity is:

$$\begin{aligned}
\mathbb{E}[\hat{\gamma}_i] &= \mathbb{E}[\gamma_i] + F_i - F_{i-1} \\
&\leq r_1 n_i \log n_i + c(m_i/2 - n_i)\log m_i - c(m_{i-1}/2 - n_{i-1})\log m_{i-1} \\
&\leq r_1 n_i \log n_i + c(m_i/2 - n_i)\log m_i - c(m_i - (n_i+1))\log 2m_i \\
&\leq r_1 n_i \log n_i - c(m_i/2 - 1)\log m_i - c(3n_i - 1) \\
&\leq r_1 n_i \log n_i - cm_i/2 \log m_i + c\log m_i \\
&\leq r_1 m_i \log m_i - (c/2) m_i \log m_i + c\log m_i \\
&\leq c\log m_i\,,
\end{aligned}$$

where c must also be chosen to satisfy $c > 2r_1$.

4. $\alpha_{i-1} \geq \frac{1}{2}$ (deletion). In this case we have $m_{i-1} = m_i$. If $\alpha_i \geq \frac{1}{2}$, the amortized complexity will be:

$$\begin{aligned}
\mathbb{E}[\hat{\gamma}_i] &= \mathbb{E}[\gamma_i] + F_i - F_{i-1} \\
&\leq c_1 m_i^{\epsilon} \log m_i + c(2n_i - m_i - 2n_{i-1} + m_{i-1})\log m_i \\
&\leq c_1 m_i^{\epsilon} \log m_i - 2c\log m_i\,.
\end{aligned}$$

Finally for the case that $\alpha_i < \frac{1}{2}$ we have

$$\begin{aligned}
\mathbb{E}[\hat{\gamma}_i] &= \mathbb{E}[\gamma_i] + F_i - F_{i-1} \\
&\leq c_1 m_i^{\epsilon} \log m_i + c(m_{i-1}/2 - n_i - 2n_{i-1} + m_{i-1})\log m_i \\
&= c_1 m_i^{\epsilon} \log m_i + c(3m_{i-1}/2 - (n_{i-1} - 1) - 2n_{i-1})\log m_i \\
&= c_1 m_i^{\epsilon} \log m_i + c(3m_{i-1}/2 - 3n_{i-1} + 1)\log m_i \\
&= c_1 m_i^{\epsilon} \log m_i + c(3(1/\alpha_{i-1})n_{i-1}/2 - 3n_{i-1} + 1)\log m_i \\
&\leq c_1 m_i^{\epsilon} \log m_i + c\log m_i\,.
\end{aligned}$$

Therefore we conclude that for all constants $c > 2r_1$ of the potential function, the expected value of the amortized complexity of any operation is bounded by

$$\mathbb{E}[\hat{\gamma}_i] \leq c_1 m_i^{\epsilon} \log m_i + 2c \log m_i \,.$$

By using now Relation 3.13, there is a constant r such that $\mathbb{E}[\hat{\gamma}_i] \leq r n_i^{\epsilon} \log n_i$ which implies that the expected value of the amortized complexity of any update (insertion/deletion) in an authenticated hash table containing n elements is $O(n^{\epsilon} \log n)$ for $0 < \epsilon < 1$. □

3.2.3 Queries and verification

We show now how a proof for an element $e \in \mathcal{X}$ (or an element $e \notin \mathcal{X}$) can be constructed, by using the authenticated data structure presented in the previous section. Let $H(e) = j$, i.e., the bucket that corresponds to element e is j. Let $v_0, v_1, \ldots, v_l$ be the path from the node that corresponds to bucket j to the root of $T(\epsilon)$. We add a fictitious node v_{-1} that stores element e within bucket j such that $v_{-1}, v_0, v_1, \ldots, v_l$ is the path in $T(\epsilon)$ from the node to corresponds to element e to the root of $T(\epsilon)$. We consider two cases, i.e., *membership* and *non-membership* proof:

- *Element e is contained in the hash table.* The proof is the ordered sequence $\pi_0, \pi_1, \ldots, \pi_l$, where π_i is a tuple of a prime representative and a witness that authenticates every node of the path $v_{-1}, v_0, \ldots, v_l$ from the element in question e to the root of the tree v_l. Thus, item π_i of proof $\Pi(e)$ $(i = 0, \ldots, l)$ is defined as:

$$\pi_i = \left(r_i(\chi(v_{i-1})), \mathsf{W}_{v_{i-1}(v_i)}\right) \,, \tag{3.14}$$

  where $\mathsf{W}_{v_{i-1}(v_i)}$ is defined in Relation 3.8 and $\chi(v_{-1}) = e$. For simplicity, we set $\alpha_i = r_i(\chi(v_{i-1}))$ and

$$\beta_i = \mathsf{W}_{v_{i-1}(v_i)} \,. \tag{3.15}$$

  For example in Figure 3.1, the proof for an element that belongs to the bucket of node

a (e.g., element 2) consists of the following tuples:

$$\begin{aligned}\pi_0 &= \left(r_0(2), g^{r_0(3)r_0(7)r_0(9)} \mod N_0\right) ,\\ \pi_1 &= \left(r_1(\chi(a)), g_1^{r_1(\chi(b))r_1(\chi(c))r_1(\chi(d))} \mod N_1\right) ,\\ \pi_2 &= \left(r_2(\chi(f)), g_2^{r_2(\chi(e))r_2(\chi(g))r_2(\chi(p))} \mod N_2\right) .\end{aligned}$$

- *Element e is not contained in the hash table.* Let $y_1, y_2, \ldots, y_u$ be the elements contained in bucket j (all different than e). First, output a *membership proof* (as above) for an element y_i in bucket j (note that $H(y_i) = H(e)$). Then, and by running the extended Euclidean algorithm, output a non-membership witness

$$\pi_\nu = (\mathsf{A}_e, \mathsf{B}_e, r_0(e), e) , \tag{3.16}$$

where $\mathsf{A}_e, \mathsf{B}_e$ and $r_0(e)$ are defined in Relation 3.3. Note that $\mathsf{A}_e, \mathsf{B}_e$ are integer values proving non-membership of e in the set $\{y_1, y_2, \ldots, y_u\}$.

We now describe the algorithm formally:

Algorithm $\{\Pi(q), \alpha(q)\} \leftarrow \mathsf{query}(q, D_h, \mathsf{auth}(D_h), \mathsf{pk})$**:** Let $e = q$ be the queried element. If e is *contained* in D_h, set $\Pi(q) = (\pi_0, \pi_1, \ldots, \pi_l)$, as in Relation 3.14 and output $\alpha(q) = \mathsf{true}$. If e is *not contained* in D_h, output a membership proof for some other element y_i in bucket j, such that $H(e) = H(y_i)$. Then output a non-membership proof π_ν for e in bucket j, as defined in Relation 3.16. Set $\Pi(q) = (\Pi(y_i), \pi_\nu)$ and output $\alpha(q) = \mathsf{false}$.

Lemma 3.8 *Algorithm* $\mathsf{query}()$ *of the authenticated data structure scheme* $\mathcal{RHT}$ *has* $O(n^\epsilon)$ *expected access complexity, without precomputed witnesses. With precomputed witnesses, algorithm* $\mathsf{query}()$ *has* $O(1)$ *expected access complexity. Moreover, it outputs a proof* $\Pi(q)$ *of* $O(1)$ *expected group complexity.*

Proof: (a) *Membership proof*: Without precomputed witnesses, the construction of π_0 has always $O(1)$ expected access complexity since each bucket contains $O(1)$ elements in expectation. Also, the construction of each element π_i $(i = 1, \ldots, l)$ has $O(n^\epsilon)$ access complexity

due to the degree bound of the nodes in $T(\epsilon)$. Therefore the total complexity is expected $O(n^{\epsilon})$. With precomputed witnesses, each π_i can be "read" directly from memory with $O(1)$ access complexity (not expected). Finally, the group complexity of $\Pi(q)$ for a membership proof is $O(1)$ (not expected), since one witness for each level of $T(\epsilon)$ must be provided. (b) *Non-membership proof*: A non-membership proof consists of a membership proof (thus the above arguments apply here as well) plus the proof π_ν (see Relation 3.16). Therefore in both cases (without precomputed witnesses and with precomputed witnesses), π_ν turns the complexities into *expected*, since the complexity of A_e and B_e depends on the number of the elements in the bucket in question (see observation before Section 3.1.4), which is expected $O(1)$. This completes the proof. $\square$

We now formally describe the verification algorithm. The verification algorithm will take as input a proof and an answer and will either accept or reject the answer.

Algorithm $\{\mathsf{accept}, \mathsf{reject}\} \leftarrow \mathsf{verify}(q, \alpha, \Pi, d_h, \mathsf{pk})$**:** Let the query q refer to element e, i.e., $q = e$. We distinguish two cases:

1. *Membership proof*: In this case it is $\alpha = \mathsf{true}$. The proof Π contains a membership proof for e, denoted with $\Pi(e) = \pi_0, \pi_1, \ldots, \pi_l$, where $\pi_i = (\alpha_i, \beta_i)$ for $i = 0, \ldots, l$, and where α_i are all primes. The algorithm outputs reject if one of the following is true:

   (a) $h_0(\alpha_0) \neq e$ (the prime representative of element e is not correct);

   (b) $h_i(\alpha_i) \neq \beta_{i-1}^{\alpha_{i-1}} \mod N_{i-1}$ for some $1 \leq i \leq l$ (false witness);

   (c) $d_h \neq \beta_l^{\alpha_l} \mod N_l$ (final digest mismatch).

2. *Non-membership proof*: In this case it is $\alpha = \mathsf{false}$. We recall that in this case the proof Π contains (a) a membership proof $\Pi(y) = \pi_0, \pi_1, \ldots, \pi_l$ for element $y \neq e$ such that $H(y) = H(e)$, where $\pi_i = (\alpha_i, \beta_i)$ for $i = 0, \ldots, l$ (α_i are all primes); (b) a non-membership proof for e, denoted with $\pi_\nu = (\mathsf{A}, \mathsf{B}, r, e)$, where r is a prime. The algorithm outputs reject if one of the following is true:

(a) $H(e) \neq H(y)$; (e and y do not belong in the same bucket);

(b) The membership proof for y does not verify, i.e., it is

$$\mathsf{reject} \leftarrow \mathsf{verify}(y, \mathsf{true}, \Pi(y), d_h, \mathsf{pk})\,;$$

(c) $h_0(r) \neq e$ (the prime representative r contained in π_ν for element e is not correct);

(d) $\alpha_1^{\mathsf{A}} g_0^{r\mathsf{B}} \neq g_0 \mod N_0$ (verification test for non-membership proof of e in the corresponding bucket does not succeed, see Lemma 3.2).

If all the above tests are successful, the algorithm outputs accept.

Lemma 3.9 *Algorithm* verify() *of the authenticated data structure scheme* $\mathcal{RHT}$ *has* $O(1)$ *expected access complexity.*

Proof: Processing the membership proof has $O(1)$ access complexity, since it requires processing $l = O(1)$ pairs of witnesses and prime representatives. Moreover, processing the non-membership proof has $O(1)$ expected access complexity, due to the size of the non-membership witness, that depends on the number of the elements in the bucket in question. Therefore, the expected access complexity of verify() is $O(1)$. □

3.2.4 Correctness and security

The following lemmata describe the correctness and the security of our new construction, according to Definitions 2.4 and 2.5. The security of our scheme is based on the strong RSA assumption.

Lemma 3.10 *The authenticated data structure scheme* $\mathcal{RHT} = \{$genkey, setup, update, refresh, query, verify$\}$ *is correct according to Definition 2.4.*

Proof: Let D_0 be any hash table containing n elements and having $m = O(n)$ buckets. Fix the security parameter k and output $\mathsf{pk} = \{N_i, g_i, h_i : i = 0, \ldots, l; \epsilon\}$ and $\mathsf{sk} = \{\phi(N_i) :$

$i = 0, \ldots, l\}$ by calling algorithm $\mathsf{genkey}()$. Then output an authenticated data structure $\mathsf{auth}(D_0)$ and the respective digest d_0, by calling algorithm $\mathsf{setup}()$. Pick a polynomial number of updates—namely, pick a polynomial number of elements for insertion or deletion—and update $\mathsf{auth}(D_0)$ and d_0 by calling algorithm $\mathsf{refresh}()$. Let D_h be the final hash table, $\mathsf{auth}(D_h)$ be the produced authenticated data structure and d_h be the final digest. Let e be an element that belongs (or, it should belong) to bucket j (i.e., $H(e) = j$). Output a proof $\Pi(e)$ and an answer by calling $\mathsf{query}()$. We distinguish two cases:

1. *Element e is contained in the hash table.* Then $\Pi(e)$ is a membership proof as defined in Relation 3.14. Note that π_0 contains the prime representative of e with the respective witness, therefore $\mathsf{verify}()$ does not reject at Item 1a. By the definitions of the accumulation values output by $\mathsf{setup}()$ and maintained under updates by $\mathsf{refresh}()$ (see Relations 3.6 and 3.7) and by the definition of proof element π_i in Relation 3.14 for $i = 1, \ldots, l$, $\mathsf{verify}()$ does not reject at Items 1b and 1c;

2. *Element e is not contained in the hash table.* Let $y_1, y_2, \ldots, y_u$ be the elements in bucket j, where $H(e) = j$. In this case, the non-membership proof consists of (a) a membership proof $\pi_i = (\alpha_i, \beta_i)$ for an element y contained in bucket j (which verifies due to Item 1) such that $H(e) = H(y) = j$, therefore $\mathsf{verify}()$ does not reject at either Item 2a or Item 2b and, (b) a non-membership proof $(\mathsf{A}_e, \mathsf{B}_e, r_0(e), e)$ for element e that *should* belong to bucket j. Therefore $\mathsf{verify}()$ does not reject at Item 2c, since $h_0(r_0(e)) = e$. Also it does not reject at Item 2d as

$$\alpha_1^{\mathsf{A}_e} g_0^{r_0(e)\mathsf{B}_e} = g_0^{\left(\prod_{j=1}^{u} r_0(y_j)\right)\mathsf{A}_e + r_0(e)\mathsf{B}_e} = g_0 \mod N_0 \,,$$

since, by construction it is

$$\alpha_1 = g_0^{\prod_{j=1}^{u} r_0(y_j)} \mod N_0 \,,$$

and by Relation 3.3, A_e and B_e are computed to satisfy

$$\left(\prod_{j=1}^{u} r_0(y_j)\right) \mathsf{A}_e + r_0(e)\mathsf{B}_e = 1 \,.$$

This completes the proof. □

Lemma 3.11 *The authenticated data structure scheme* $\mathcal{RHT}$ = {genkey, setup, update, refresh, query, verify} *is secure according to Definition 2.5 and under the strong RSA assumption.*

Proof: Let k be the security parameter. Output $\mathsf{pk} = \{N_i, g_i, h_i : i = 0, \ldots, l; \epsilon\}$ and $\mathsf{sk} = \{\phi(N_i) : i = 0, \ldots, l\}$ by calling algorithm genkey(). Let Adv be a polynomially-bounded adversary. Adv picks an initial collection of n elements $\mathcal{X}$, stored in hash table D_0. Adv outputs an authenticated data structure $\mathsf{auth}(D_0)$, by calling algorithm setup() through oracle access. Then Adv picks a polynomial number of updates—namely, he picks a polynomial number of elements for insertion or deletion. Let D_h be the final hash table, let the updated final element collection be $\mathcal{X}$, and let d_h be the final digest as produced by the adversary through oracle access to algorithm update(). We will compute the probability that **check**() rejects, while verify() accepts, as required by Definition 2.5. We distinguish two cases:

1. *Membership proof*: The adversary outputs a membership proof $\Pi(e) = (\pi_0, \pi_1, \ldots, \pi_l)$ ($l = \lceil \frac{1}{\epsilon} \rceil$) where $\pi_i = (\alpha_i, \beta_i)$ (see algorithm query()) for an element $e \notin \mathcal{X}$ (thus, a proof for an incorrect answer). Let $v_0, v_1, \ldots, v_l$ be a path of nodes in $T(\epsilon)$ from the bucket referring to e to the root of the tree. We define now the following events, related to the choice of the proof above made by the adversary. Our goal will be to the express the probability that $\mathsf{verify}(e, \mathsf{true}, \Pi(e), d_h, \mathsf{pk})$ accepts and $e \notin \mathcal{X}$ as a function of the following events. Note that d_h is the correct digest of the authenticated data structure:

   (a) $\mathcal{E}_{0,0}$: The value e and α_0 picked by Adv are such that $e \notin \mathcal{X}$, α_0 is prime and $h_0(\alpha_0) = e$;

   (b) $\mathcal{E}_j$: For $j = 1, \ldots, l$, the values α_j, α_{j-1} and β_{j-1} picked by Adv are such that both α_j and α_{j-1} are primes and

$$h_j(\alpha_j) = \beta_{j-1}^{\alpha_{j-1}} \mod N_{j-1} \text{ for all } 1 \leq j \leq l\,.$$

This event can be partitioned into two mutually exclusive events, i.e., $\mathcal{E}_j = \mathcal{E}_{j,0} \cup \mathcal{E}_{j,1}$ such that

- $\mathcal{E}_{j,0}$: Value $h_j(\alpha_j)$ is *not* the correctly formed digest (i.e., an accumulation of the digests of its children) of some node $v_{j-1} \in \mathcal{N}(v_j)$, as defined in Relation 3.7;
- $\mathcal{E}_{j,1}$: Value $h_j(\alpha_j)$ is the correctly formed digest of a node $v_{j-1} \in \mathcal{N}(v_j)$, as defined in Relation 3.7.

(c) $\mathcal{E}_{l+1,1}$: The values α_l and β_l picked by Adv are such that

$$\beta_l^{\alpha_l} = d_h \mod N_l \,.$$

The probability that $\mathsf{verify}()$ accepts, while $e \notin \mathcal{X}$ is the probability

$$\begin{aligned} & \Pr[\mathcal{E}_{0,0} \cap \mathcal{E}_1 \cap \mathcal{E}_2 \cap \ldots \cap \mathcal{E}_{l+1,1}] \\ = \; & \Pr[\mathcal{E}_{0,0} \cap (\mathcal{E}_{1,0} \cup \mathcal{E}_{1,1}) \cap (\mathcal{E}_{2,0} \cup \mathcal{E}_{2,1}) \cap \ldots \cap \mathcal{E}_{l+1,1}] \\ \leq \; & \Pr[\mathcal{E}_{1,1}|\mathcal{E}_{0,0}] + \Pr[\mathcal{E}_{2,1}|\mathcal{E}_{1,0}] + \Pr[\mathcal{E}_{3,1}|\mathcal{E}_{2,0}] + \ldots + \Pr[\mathcal{E}_{l+1,1}|\mathcal{E}_{l,0}] \\ = \; & \Pr[\mathcal{E}_{1,1}|\mathcal{E}_{0,0}] + \sum_{j=2}^{l+1} \Pr[\mathcal{E}_{j,1}|\mathcal{E}_{j-1,0}] \,. \end{aligned} \tag{3.17}$$

First we examine the event $\mathcal{E}_{1,1}|\mathcal{E}_{0,0}$. This event implies that the adversary has found a value $e \notin \mathcal{X}$, a prime α_0 such that $h_0(\alpha_0) = e$ and a value β_0 such that

$$\beta_0^{\alpha_0} = g_0^{\prod_{t=1,\ldots,l'} r_0(x_t)} \mod N_0 \,,$$

where $x_1, x_2, \ldots, x_{l'}$ is a subset of the set $\mathcal{X}$. Since $e \notin \mathcal{X}$, it is $e \notin \{x_1, x_2, \ldots, x_{l'}\}$. Also, since every prime representative is mapped to a *unique* element through function h_0, we conclude that it should be $\alpha_0 \notin \{r_0(x_1), r_0(x_2), \ldots, r_0(x_{l'})\}$. By Lemma 3.2 and Assumption 3.1, this probability is $\mathsf{neg}(k)$. Therefore $\Pr[\mathcal{E}_{1,1}|\mathcal{E}_{0,0}] \leq \mathsf{neg}(k)$.

For the remaining events $\mathcal{E}_{j,1}|\mathcal{E}_{j-1,0}$ $(2 \leq j \leq l+1)$, we have:

- By the one-to-one property of the function $h_{j-1}(.)$, $\mathcal{E}_{j-1,0}$ implies that value α_{j-1} is *not* the prime representative of the correctly formed digest of some node $v_{j-2} \in \mathcal{N}(v_{j-1})$, as defined in Relation 3.7, namely that

$$\alpha_{j-1} \notin \{r_{j-1}(\chi(v_t)) : v_t \in \mathcal{N}(v_{j-1})\}\,;$$

- However, the event $\mathcal{E}_{j,1}$ implies that (1) digest $h_j(\alpha_j)$ (for $j = l+1$ this is just d_h) is the correctly formed digest of node v_{j-1}; and (2)

$$\beta_{j-1}^{\alpha_{j-1}} = g_{j-1}^{\prod_{v_t \in \mathcal{N}(v_{j-1})} r_{j-1}(\chi(v_t))} \mod N_{j-1}\,.$$

  where $r_{j-1}(\chi(v_t))$ are the prime representatives of correctly formed digests of the set of neighbors of v_{j-1}.

Since $\alpha_{j-1} \notin \{r_{j-1}(\chi(v_t)) : v_t \in \mathcal{N}(v_{j-1})\}$, by Lemma 3.2 and Assumption 3.1, this probability is $\mathsf{neg}(k)$. Therefore for all $j = 1, \ldots, l+1$, $\Pr[\mathcal{E}_{j,1}|\mathcal{E}_{j-1,0}]$ is $\mathsf{neg}(k)$. Since $l = O(1)$, the total probability is also $\mathsf{neg}(k)$. This concludes the proof for the membership case.

2. *Non-membership proof*: For this case, we define the events:

   (a) $\mathcal{B}_0$: Adv finds $e \in \mathcal{X}$ and y such that $H(e) = H(y) = j$;

   (b) $\mathcal{B}_1$: Adv finds a membership proof for y, namely the proof $\pi = \pi_0, \pi_1, \ldots, \pi_l$ where $\pi_i = (\alpha_i, \beta_i)$ (where α_i are prime numbers) and $\mathsf{accept} \leftarrow \mathsf{verify}(y, \mathsf{true}, \pi, d_h, \mathsf{pk})$;

   (c) $\mathcal{B}_2$: Adv finds α_1, a non-membership proof $(\mathsf{A}, \mathsf{B}, r, e)$ for e such that r is a prime number, $h_0(r) = e$ and $\alpha_1^{\mathsf{A}} g_0^{r\mathsf{B}} = g_0 \mod N_0$. This event is partitioned into two events:

      i. $\mathcal{B}_{20} : \alpha_1 \neq \mathsf{acc}(L_j) = g_0^{\prod_{x \in L_j} r_0(x)} \mod N_0$;

      ii. $\mathcal{B}_{21} : \alpha_1 = \mathsf{acc}(L_j) = g_0^{\prod_{x \in L_j} r_0(x)} \mod N_0$.

We need to compute the probability $\Pr[\mathcal{B}_0 \cap \mathcal{B}_1 \cap \mathcal{B}_2] = \Pr[\mathcal{B}_0 \cap \mathcal{B}_1 \cap (\mathcal{B}_{20} \cup \mathcal{B}_{21})] \leq \Pr[\mathcal{B}_{20}|\mathcal{B}_1|\mathcal{B}_0] + \Pr[\mathcal{B}_{21}|\mathcal{B}_0]$. By Lemma 3.2 and Assumption 3.1, it is $\Pr[\mathcal{B}_{21}|\mathcal{B}_0] \leq$

$\nu(k)$, where $\nu(k)$ is the appropriate negligible function. Note also that we can express $\mathcal{B}_{20}|\mathcal{B}_1|\mathcal{B}_0$ as a function of the events $\mathcal{E}$ in the membership proof case. Specifically, the event $\mathcal{B}_{20}|\mathcal{B}_1|\mathcal{B}_0$ implies the event $\mathcal{E}_{1,0} \cap \mathcal{E}_2 \cap \mathcal{E}_3 \cap \ldots \cap \mathcal{E}_{l+1,1}$, the probability of which, by following the same logic as in Relations 3.17 is bounded by $\mathsf{neg}(k)$.

This concludes the proof for both the membership and the non-membership cases. □

We can now present the main result of this section.

Theorem 3.2 *Let k be the security parameter and $0 < \epsilon < 1$. Then there exists a publicly-verifiable authenticated data structure scheme $\mathcal{RHT} = \{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$ for a data structure scheme defined for dynamic hash table D storing n elements such that:*

1. *It is correct according to Definition 2.4 and secure according to Definition 2.5 and under the strong RSA assumption;*
2. *The access complexity of $\mathsf{setup}()$ is $O(n)$, outputting an authenticated data structure $\mathsf{auth}(D)$ of $O(n)$ group complexity;*
3. *The expected amortized access complexity of $\mathsf{update}()$ is $O(1)$, outputting update information upd of $O(1)$ group complexity;*
4. *The expected amortized access complexity of $\mathsf{refresh}()$ is $O(n^\epsilon \log n)$ (or $O(1)$);*
5. *The expected access complexity of $\mathsf{query}()$ is $O(1)$ (or $O(n^\epsilon)$), outputting a proof $\Pi(q)$ for a query q of $O(1)$ expected group complexity;*
6. *The expected access complexity of $\mathsf{verify}()$ is $O(1)$.*

Proof: This result follows directly from Lemmata 3.4, 3.5, 3.7, 3.8, 3.9, 3.10 and 3.11. The complexities in the brackets ($O(1)$ for $\mathsf{refresh}()$ and $O(n^\epsilon)$ for $\mathsf{query}()$) refer to the case when no precomputed witnesses are used. Note that the presented scheme is publicly verifiable since $\mathsf{verify}()$ does not take the secret key as an input. □

3.2.5 A more practical scheme

The construction we have presented ($\mathcal{RHT}$ authenticated data structure scheme) uses different RSA moduli for each level of the tree and each new RSA modulus has a bit-length that is three times longer than the bit-length of the previous-level RSA modulus. Therefore, computations corresponding to higher levels in the accumulation tree are more expensive, since they involve modular arithmetic operations over longer elements. This increase in the lengths of the RSA moduli is due to the need to compute, for the elements stored at every level in the tree, prime representatives of size that is three times as large as the size of the elements (see Lemma 3.1). Although from a theoretical point of view this is not a concern as the number of the levels of the tree is constant (i.e., $1/\epsilon$), from a practical point of view, this can be prohibitive for efficiently implementing our schemes.

To overcome this complexity overhead, we want to use the same RSA modulus for each level of the tree, and to achieve this, we present a heuristic inspired by a similar method originally used in the work of Baric and Pfitzmann [10]. Instead of using two-universal hash functions to map (general) integers to primes of increased size, the idea is to employ random oracles [12] for consistently computing primes of relatively small size. In particular, given a k-bit integer x, instead of mapping it to a $3k$-bit prime, we can map it to the value $2^t 2^b g(x) + d$, where $g(x)$ is the output of length b of a random oracle (which in practice is the output of a cryptographic hash function) at the end of which we append b zeros so that we make this number large enough, t is a value that equals to the number of bits we are shifting $2^b g(x)$ to the left, and $d = 1, 3, \ldots, 2^t - 1$ is a number we are adding so that $2^t 2^b g(x) + d$ is a prime. Note that we require that t is related to b according to Relation 3.18 of Theorem 3.3.

In the following, we denote by $q(x)$ a prime representative of x computed by the above procedure, i.e., the output of a procedure that transforms a k-bit integer into a k'-bit prime, where $k' < k$. Note that the above procedure (i.e., the computation of $q(x)$) cannot map two different integers to the same prime. This can be derived by the random oracle property, namely that for $x_1 \neq x_2$, w.h.p. it is $g(x_1) \neq g(x_2)$. This implies that the intervals

$[2^t 2^b g(x_1), 2^t 2^b g(x_1) + 2^t - 1]$ and $[2^t 2^b g(x_2), 2^t 2^b g(x_2) + 2^t - 1]$ are disjoint. Finally we show that we can make sure that with high probability we will always be able to find a prime within the specified interval.

Theorem 3.3 *Let x be a k-bit integer and let $a = 2^b g(x)$ be the output of a b-bit random oracle with b zeros appended at the end. The interval $[2^t a, 2^t a + 2^t - 1]$ contains a prime with probability at least $1 - 2^{-b}$ provided*

$$b \leq \left\lfloor \log(1 + \sqrt{2^t + 4e^{2^t - 1}}) - 1 \right\rfloor . \tag{3.18}$$

Proof: By the prime distribution theorem we have that the number of primes less than n is approximately $\frac{n}{\ln n}$. Therefore, we want to compute the probability

$$\Pr\left[\frac{2^t a + 2^t - 1}{\ln(2^t a + 2^t - 1)} - \frac{2^t a}{\ln(2^t a)} \geq 1\right] = \Pr\left[a \leq \frac{e^{2^t - 1}}{2^t}\right] ,$$

by assuming $\ln(2^t a + 2^t - 1) \simeq \ln(2^t a)$ since $a > 2^b >> 2^t$. By the random oracle property we have that

$$\Pr\left[a \leq \frac{e^{2^t - 1}}{2^t}\right] = \Pr\left[2^b g(x) \leq \frac{e^{2^t - 1}}{2^t}\right] = \frac{e^{2^t - 1}}{2^{b+t}} \frac{1}{2^b} .$$

Note that

$$\frac{e^{2^t - 1}}{2^{b+t}} \frac{1}{2^b} \geq 1 - \frac{1}{2^b} \Leftrightarrow \frac{1 - \sqrt{2^t + 4e^{2^t - 1}}}{2} \leq 2^b \leq \frac{1 + \sqrt{2^t + 4e^{2^t - 1}}}{2} ,$$

which gives $b \leq \left\lfloor \log(1 + \sqrt{2^t + 4e^{2^t - 1}}) - 1 \right\rfloor$ since b is a positive integer. This completes the proof. □

Using Theorem 3.3, we can pick the length of the output of the random oracle to ensure hitting a prime with high probability. For example, for $t = 9$ we get $b \leq 368$, which is true for most practical hash functions used today (e.g., SHA-256).

Using the above method, we can still accumulate primes in the exponent but this time without having to increase the size of the RSA moduli at any level of the tree. The only conditions we need in order to securely use the RSA accumulator are:

1. the safe accumulation of primes that map to unique integers (i.e., each accumulated prime can only represent one integer), and

2. the bit-length of accumulated primes is smaller than the bit-length of the used RSA modulus.

Thus, we can apply our new procedure for computing prime representatives to all of the constructions in Section 3.2 with one important efficiency improvement: the same RSA modulus and exponentiation bases are used at all levels of the accumulation tree. With this heuristic, we overall get the same security and complexity results as before, but now we have a more practical accumulator with security that is now based on both the strong RSA and the random oracle assumptions.

3.2.6 Protocols

Three-party protocol. By using Theorem 2.1 we can easily derive the following corollary that describes the use of the authenticated data structure scheme $\mathcal{RHT}$ of Theorem 3.2 by Protocol 2.1.

Corollary 3.1 *Let k be the security parameter and assume that the strong RSA assumption holds. Then there exists a three-party authenticated data structures protocol (see Protocol 2.1) for verifying (non)-membership queries q on a dynamic hash table storing n elements such that:*

1. *The setup at the source has $O(n)$ access complexity;*

2. *The update at the source has $O(1)$ expected amortized access complexity;*

3. *The space needed at the source has $O(n)$ group complexity;*

4. *The communication between the source and the server has $O(1)$ group complexity;*

5. *The update at the server has $O(n^\epsilon \log n)$ (or $O(1)$) expected amortized access complexity;*

6. *The query at the server has $O(1)$ (or $O(n^\epsilon)$) expected access complexity;*

7. *The space needed at the server has $O(n)$ group complexity;*

8. *The communication between the server and the client has $O(1)$ expected group complexity;*

9. *The verification at the client has $O(1)$ expected access complexity;*

10. *For a query q sent by the client to the server at any time (even after updates), let α be an answer and let π be a proof returned by the server. With probability $\Omega(1 - \mathsf{neg}(k))$, the client accepts the answer α if and only if α is correct.*

Two-party protocol. In order to be able to use the authenticated data structure scheme $\mathcal{RHT}$ of Theorem 3.2 in a black-box way with Theorem 2.2—and derive a two-party authenticated data structures protocol, we have to ensure that Assumption 2.1 holds for the authenticated data structure scheme $\mathcal{RHT}$:

Lemma 3.12 *Assumption 2.1 is true for the authenticated data structure scheme $\mathcal{RHT}$. Moreover, for every update u, $|Q_u|$ has $O(1)$ amortized complexity.*

Proof: Let an update u refer to element e, i.e., either *insert* element e to the hash table or *delete* element e from the hash table. We distinguish two cases:

1. The hash table is not rebuilt (see Definition 3.2) due to update u. In this case, the respective set of queries Q_u required for Assumption 2.1 *simply* contains one query for element e, i.e., $q_u = e$. Let $\{\Pi(e), \alpha(e)\} \leftarrow \mathsf{query}(e, D_h, \mathsf{auth}(D_h), \mathsf{pk})$.

   We now describe function $z(.)$ from Assumption 2.1. Let $q_u = e$. Function $z(.)$ first computes $\delta_u(D_h)$ as $H(e) = j$, since $\mathsf{update}()$ needs to access bucket j in order to do any

operation on element e, with $H(e) = j$[4]. To compute $\delta_u(\mathsf{auth}(D_h))$, $z(.)$ processes the proof $\Pi(q_u)$ as follows. We recall that both a membership and non-membership proof contains the ordered sequence $\pi_0, \pi_1, \ldots, \pi_l$, where π_i is a tuple of a prime representative and a witness that authenticates every digest of the path $v_0, v_1, \ldots, v_l$ from the bucket in question $H(e) = j$ to the root of the tree v_l. Thus, item π_i of proof $\Pi(e)$ $(i = 0, 1, \ldots, l)$ is defined as (see Relation 3.14):

$$\pi_i = \left(r_i(\chi(v_{i-1})), \mathsf{W}_{v_{i-1}(v_i)}\right) .$$

Note now that $\mathsf{update}()$ needs to access $\chi(v_i)$, for $i = 0, 1, \ldots, l$, in order to perform the update (see Relation 3.10). All these values are easily computed from π_i: Just set $\chi(v_i) = \mathsf{W}_{v_{i-1}(v_i)}^{r_i(\chi(v_{i-1}))}$—see Relation 3.7. Therefore the function $z(.)$ computes such exponentiations and outputs $\delta_u(\mathsf{auth}(D_h))$ with $O(1)$ complexity, same with the complexity of algorithm $\mathsf{verify}()$.

2. The hash table is rebuilt (see Definition 3.2) due to update u. Then the set of queries Q_u consists of queries for *all* the elements contained in the hash table, therefore its size is $O(n)$, where n is the number of the elements stored in the table. In this case $z(.)$ is just a call to algorithm $\mathsf{setup}()$.

Because the hash table has to be rebuilt, and in this case it is $|Q_u| = O(n)$, it follows (with a similar analysis as in Lemma 3.5) that $|Q_u|$ has $O(1)$ amortized complexity. This completes the proof. □

By Theorems 2.2 and 3.2 and Lemma 3.12, we can now state the final result for the two-party model:

Corollary 3.2 *Let k be the security parameter and assume that the strong RSA assumption holds. Then there exists a two-party authenticated data structures protocol (see Protocol 2.2)*

[4]We recall that $\mathsf{update}()$ does not update the bucket j itself, but receives the updated bucket from **update**()—see Definition 2.3.

for verifying (non)-membership queries q on a dynamic hash table storing n elements such that:

1. *When precomputed witnesses are used, the protocol is non-interactive; Otherwise, it requires one round of interaction during updates;*
2. *The setup at the client has $O(n)$ access complexity;*
3. *The update at the client has $O(1)$ expected amortized access complexity;*
4. *The verification at the client has $O(1)$ expected access complexity;*
5. *The space needed at the client has $O(1)$ group complexity;*
6. *The communication between the client and the server has $O(1)$ expected amortized group complexity during updates and $O(1)$ expected group complexity during queries;*
7. *The update at the server has $O(n^{\epsilon} \log n)$ (or $O(n^{\epsilon})$) expected amortized access complexity;*
8. *The query at the server has $O(1)$ (or $O(n^{\epsilon})$) expected access complexity;*
9. *The space needed at the server has $O(n)$ group complexity;*
10. *For a query q sent by the client to the server at any time (even after updates), let α be an answer and let π be a proof returned by the server. With probability $\Omega(1 - \mathsf{neg}(k))$, the client accepts the answer α if and only if α is correct.*

3.3 Scheme based on the bilinear-map accumulator

In this section we use the bilinear-map accumulator and present a new authenticated data structure scheme $\mathcal{BHT} = \{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$ for dynamic hash tables. We use exactly the same methodology as the one used in Section 3.2, that is, nested

invocations of accumulators in a constant-depth tree, to overall obtain similar complexity and security results with the solution presented before. Accordingly, we use the same structure in presenting and proving our results. Note however that there are significant differences (both in complexity and cryptography) that are imposed by the use of the different cryptographic primitive. For example, the underlying algebraic groups used are fundamentally different (the RSA accumulator is using $\mathbb{Z}_N$ while the bilinear-map accumulator is using groups defined over elliptic curves). This imposes certain differences in the complexity of some algorithms, which will be discussed in the following sections. We begin with algorithms $\mathsf{genkey}()$ and $\mathsf{setup}()$. Again, the underlying (plain) data structure is a dynamic hash table $\mathbf{T}(\mathcal{X})$ storing n elements $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$. As in the RSA accumulator case, the elements are distributed into m buckets $L_1, L_2, \ldots, L_m$ (using a two-universal hash function H), where $m = O(n)$.

Algorithm $\{\mathsf{sk}, \mathsf{pk}\} \leftarrow \mathsf{genkey}(1^k)$**:** The algorithm chooses a k-bit prime p, an exponentiation base g that is a generator of a multiplicative cyclic group $\mathbb{G}$ of prime order p, for which there is a bilinear map $e(.,.) : \mathbb{G} \times \mathbb{G} \rightarrow \mathcal{G}$[5]. All the above are chosen uniformly at random as indicated by Assumption 3.2, basically the algorithm has to generate the tuple $\mathsf{t} = (p, \mathbb{G}, \mathcal{G}, e, g)$. Then it randomly picks a number $s \in \mathbb{Z}_p^*$ (s is the trapdoor). An upper bound q of the total number of elements that will be accumulated is decided and the algorithm also computes the elements of $\mathbb{G}$ $g^s, g^{s^2}, \ldots, g^{s^q}$. Finally, a function that outputs the bit-description of the elements in $\mathbb{G}$, i.e., the function $h : \mathbb{G} \rightarrow \mathbb{Z}_p$ is used[6]. Note that since $\mathbb{G}$ has exactly p elements, the function maps each element in $\mathbb{G}$ to an integer in $\mathbb{Z}_p$. In order not to overload the notation, we assume that *when* the input to the function $h(.)$ is an element $x \in \mathbb{Z}_p$, it just outputs x. The algorithm outputs $s \in \mathbb{Z}_p^*$ as sk and everything else as pk.

[5]The generator g is used as the exponentiation base in all the levels of the accumulation tree $T(\epsilon)$.

[6]In this way we make sure that the output accumulated value at some level can be used as input to the next level of accumulation, since we can only accumulate elements of $\mathbb{Z}_p$ and not elements of $\mathbb{G}$

Algorithm $\{\mathsf{auth}(D_0), d_0\} \leftarrow \mathsf{setup}(D_0, \mathsf{sk}, \mathsf{pk})$**:** The algorithm builds the accumulation tree $T(\epsilon)$ on top of the m buckets $L_1, L_2, \ldots, L_m$. For every leaf node v in tree $T(\epsilon)$ that lies at level 0 and corresponds to a bucket L_j, the algorithm sets

$$\chi(v) = g^{\prod_{x \in L_j}(x+s)} \in \mathbb{G}\,, \tag{3.19}$$

while for every non-leaf node v in $T(\epsilon)$ that lies at level $1 \leq i \leq l$, the algorithm sets:

$$\chi(v) = g^{\prod_{u \in \mathcal{N}(v)}(h(\chi(u))+s)} \in \mathbb{G}\,, \tag{3.20}$$

where $h(\chi(u))$ is an element in $\mathbb{Z}_p$, computed using function $h()$. The authenticated data structure $\mathsf{auth}(D_0)$ output by the algorithm consists of the following components:

1. The accumulation tree $T(\epsilon)$;

2. For every node $v \in T(\epsilon)$ at level i, the accumulation values $\chi(v)$.

Let r be the root of the tree T. The algorithm also outputs $d_0 = \chi(r)$, i.e., the digest of the authenticated data structure is the $\chi(.)$ value of the root of the accumulation tree.

Precomputed witnesses. The precomputed witnesses in this case are defined as follows: For every $j \in \mathcal{N}(v)$ we store at node v the witness

$$\mathsf{W}_{j(v)} = g^{\prod_{u \in \mathcal{N}(v)-\{j\}}(h(\chi(u))+s)}\,. \tag{3.21}$$

When the construction with the precomputed witnesses is used, $\mathsf{auth}(D_0)$ also includes $\mathsf{W}_{j(v)}$, for all $v \in T(\epsilon)$ and all $j \in \mathcal{N}(v)$.

Lemma 3.13 *Algorithm* $\mathsf{setup}()$ *of the authenticated data structure scheme* $\mathcal{BHT}$ *has* $O(n)$ *access complexity both with and without precomputed witnesses. Moreover, the authenticated data structure* $\mathsf{auth}(D_0)$ *output by* $\mathsf{setup}()$ *has always* $O(n)$ *group complexity.*

Proof: Same with Lemma 3.4 with the difference that the efficient computation of the exponent expressions is now feasible since sk contains the trapdoor s. □

We continue with the algorithms used for updates:

Algorithm $\{D_{h+1}, \mathsf{auth}(D_{h+1}), d_{h+1}, \mathsf{upd}\} \leftarrow \mathsf{update}(u, D_h, \mathsf{auth}(D_h), d_h, \mathsf{sk}, \mathsf{pk})$**:** Let m be the current number of buckets of D_h and n be the number of elements stored in D_h, *after* the update has been performed. We distinguish two cases:

Case 1. $\frac{m}{4} < n < m$**:** In this case there is no need to rebuild the table and the update is performed as follows: Suppose the update we consider is the insertion of an element $e \in \mathbb{Z}_p$. The algorithm computes the bucket $j = H(e)$ (see Relation 3.1) and inserts e in bucket j. Let v_0 be the node of $T(\epsilon)$ referring to bucket j. Let $v_0, v_1, \ldots, v_l$ be the path in $T(\epsilon)$ from node v_0 to the root of the tree. The algorithm initially sets

$$\chi'(v_0) = \chi(v_0)^{e+s},$$

i.e., it updates the accumulation value that corresponds to the updated bucket. Note that if the update we consider is the deletion of an element e, the algorithm sets

$$\chi'(v_0) = \chi(v_0)^{(e+s)^{-1}}. \tag{3.22}$$

Subsequently, for $j = 1, \ldots, l$ the algorithm sets

$$\chi'(v_j) = \chi(v_j)^{(h(\chi'(v_{j-1}))+s)(h(\chi(v_{j-1}))+s)^{-1}}, \tag{3.23}$$

where $\chi(v_{j-1})$ is the previous accumulation value and $\chi'(v_{j-1})$ is the updated accumulation value. All these values are stored by the algorithm after they have been computed. The algorithm also outputs the new accumulation values $\chi'(v_{j-1})$ $(i = 1, \ldots, l)$ as the information upd along the path from the updated bucket to the root of the tree. Information upd also includes e and $\chi'(v_l)$. Also it sets $d_{h+1} = \chi'(v_l)$, i.e., the updated digest is the updated $\chi(.)$ value of the root of $T(\epsilon)$. Finally the new authenticated data structure $\mathsf{auth}(D_{h+1})$ is computed as follows. Let $\mathsf{auth}(D_h)$ be the previous authenticated data structure that is input to the algorithm: Overwrite the values $\chi(v_{j-1})$ $(j = 1, \ldots, l)$ with the new values $\chi'(v_{j-1})$ $(j = 1, \ldots, l)$ and output the updated structure. The behavior of the algorithm in the *precomputed witnesses* case is the same, with the difference that $\mathsf{upd} = \emptyset$.

Case 2. $m = \frac{m}{4}$ **or** $n = m$**:** In this case the hash table is rebuilt according to Definition 3.2:

If $n = \frac{m}{4}$, then the algorithm builds a data structure D_{h+1} with $m/2$ buckets. Otherwise, i.e., when $n = m$, the algorithm builds a data structure D_{h+1} with $2m$ buckets. Subsequently, it outputs $\mathsf{auth}(D_{h+1})$ and d_{h+1} by calling algorithm $\mathsf{setup}(D_{h+1}, \mathsf{sk}, \mathsf{pk})$. However, instead of setting $\mathsf{upd} = \emptyset$, it sets $\mathsf{upd} = \{\mathsf{auth}(D_{h+1}), d_{h+1}\}$.

Lemma 3.14 *By using the rebuilding policy of Definition 3.2, algorithm* $\mathsf{update}()$ *of the authenticated data structure scheme* $\mathcal{BHT}$ *has* $O(1)$ *expected amortized access complexity. Moreover, the update information* upd *output by* $\mathsf{update}()$ *has* $O(1)$ *amortized group complexity.*

Proof: Same as in Lemma 3.5. However, the group complexity of upd is *amortized*, because when the hash table is rebuilt, it contains the new authenticated data structure (of group complexity $O(n)$). □

Before presenting the remaining algorithms we provide some necessary complexity results that we are going to need. The following result is derived by using an FFT algorithm (e.g., see Preparata and Sarwate [96]) that computes the DFT in a finite field (e.g., $\mathbb{Z}_p$) for arbitrary n and with $O(n \log n)$ field operations. Note that there is no requirement for existence of an n-th root of unity in $\mathbb{Z}_p$ for the algorithm to work.

Lemma 3.15 (Polynomial interpolation with FFT [96]) *Let* $\prod_{i=1}^{n}(s+x_i) = \sum_{i=0}^{n} a_i s^i$ *be a degree-n polynomial. The coefficients* $a_n, a_{n-1}, \ldots, a_0$ *can be computed with* $O(n \log n)$ *complexity, given* $x_1, x_2, \ldots, x_n$.

Algorithm $\{D_{h+1}, \mathsf{auth}(D_{h+1}), d_{h+1}\} \leftarrow \mathsf{refresh}(u, D_h, \mathsf{auth}(D_h), d_h, \mathsf{upd}, \mathsf{pk})$**:** Let m be the current number of buckets of D_h and n be the number of elements stored in D_h, *after* the update has been performed. We distinguish two cases:

Case 1. $\frac{m}{4} < n < m$**:** Suppose the update is an insertion of element e. The algorithm computes the bucket $j = H(e)$ (see Relation 3.1) and inserts e in bucket j. Let v_0 be the node of $T(\epsilon)$ referring to bucket j. Let $v_0, v_1, \ldots, v_l$ be the path in $T(\epsilon)$ from node v_0 to the

root of the tree. The algorithm, for $j = 0, \ldots, l$, sets

$$\chi(v_j) = \chi'(v_j)\,,$$

i.e., it updates the accumulation values that correspond to the updated path by using the information upd[7]. Finally it outputs the updated hash table as D_{h+1}, the updated accumulation values $\chi(v_j)$ (along with the ones that belong to the nodes that are not updated) as $\mathsf{auth}(D_{h+1})$ and $\chi'(v_l)$ (contained in upd) as d_{h+1}.

Precomputed witnesses. When precomputed witnesses are used, the algorithm should update $\mathsf{W}_{j(v)}$ for $v = v_0, v_1, \ldots, v_l$ and for all $j \in \mathcal{N}(v)$ (see Relation 3.8). To achieve that efficiently, the following result is used (derived in part from [83]):

Lemma 3.16 (Witnesses update formulas) *Suppose we are given the elements collection $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$. Let W_i be the witness of x_i, i.e., $\mathsf{W}_i = g^{\prod_{j\neq i}(x_j+s)}$. Then the following hold:*

1. *(Element addition) If $\mathcal{X}' = \mathcal{X} \cup \{x_{n+1}\}$, then for all $i = 1, \ldots, n+1$ it is*

$$\mathsf{W}'_i = \mathsf{acc}(\mathcal{X})\mathsf{W}_i^{x_{n+1}-x_i}\,. \tag{3.24}$$

2. *(Element deletion) If $\mathcal{X}' = \mathcal{X} - \{x_j\}$, then for all $i \neq j$ it is*

$$\mathsf{W}'_i = \left(\frac{\mathsf{W}_i}{\mathsf{W}_j}\right)^{\frac{1}{x_j-x_i}}\,. \tag{3.25}$$

3. *(Element modification) If $\mathcal{X}' = \mathcal{X} - \{x_j\} \cup \{x'_j\}$, then for all $i \neq j$ it is*

$$\mathsf{W}'_i = \mathsf{W}_j \left(\frac{\mathsf{W}_i}{\mathsf{W}_j}\right)^{\frac{x'_j-x_i}{x_j-x_i}}\,. \tag{3.26}$$

   For $i = j$, it is $\mathsf{W}'_i = \mathsf{W}_i$.

[7] Note that information upd is not *required* for $\mathsf{refresh}()$ to perform this task. Algorithm $\mathsf{refresh}()$ uses upd for efficiency. Namely, algorithm $\mathsf{refresh}()$ could compute the updated values $\chi(v_j)$ by doing polynomial interpolation, which would have $O(n^\epsilon \log n)$ complexity (see Lemma 3.15).

Proof: Relations 3.24 and 3.25 are given in the initial work of Nguyen [83]. Relation 3.26 is derived as a corollary of Relations 3.24 and 3.25. Indeed, for all $i \neq j$:

$$\begin{aligned}
\mathsf{W}_j \left(\frac{\mathsf{W}_i}{\mathsf{W}_j}\right)^{\frac{x'_j - x_i}{x_j - x_i}} &= g^{\prod_{x \in \mathcal{X} - \{x_j\}}(x+s)} \left(\frac{g^{\prod_{x \in \mathcal{X} - \{x_i\}}(x+s)}}{g^{\prod_{x \in \mathcal{X} - \{x_j\}}(x+s)}}\right)^{\frac{x'_j - x_i}{x_j - x_i}} \\
&= g^{\prod_{x \in \mathcal{X} - \{x_j\}}(x+s)} \left(g^{(x_j - x_i)\prod_{x \in \mathcal{X} - \{x_j, x_i\}}(x+s)}\right)^{\frac{x'_j - x_i}{x_j - x_i}} \\
&= g^{\prod_{x \in \mathcal{X} - \{x_j\}}(x+s)} g^{(x'_j - x_i)\prod_{x \in \mathcal{X} - \{x_j, x_i\}}(x+s)} \\
&= g^{\prod_{x \in \mathcal{X} - \{x_j\}}(x+s)} g^{-x_i \prod_{x \in \mathcal{X} - \{x_j, x_i\}}(x+s)} g^{x'_j \prod_{x \in \mathcal{X} - \{x_j, x_i\}}(x+s)} \\
&= g^{s \prod_{x \in \mathcal{X} - \{x_j, x_i\}}(x+s)} g^{x'_j \prod_{x \in \mathcal{X} - \{x_j, x_i\}}(x+s)} \\
&= g^{\prod_{x \in \mathcal{X}' - \{x_i\}}(x+s)} \\
&= \mathsf{W}'_i \,.
\end{aligned}$$

For $i = j$, the witness W_j does not change since, by definition, W_j is not a function of the value of j (x_j or x'_j). This completes the proof. □

Corollary 3.3 (Updating precomputed witnesses) *Given the collection of elements* $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$, *and the witnesses* W_i *for all* $i = 1, \ldots, n$, *computing the updated witnesses* W'_i *of either* $\mathcal{X} \cup \{x_{n+1}\}$ *or* $\mathcal{X} - \{x_j\}$ *or* $\mathcal{X} - \{x_j\} \cup \{x'_j\}$, *without the knowledge of the trapdoor* s, *has* $O(n)$ *complexity.*

Algorithm refresh() computes the updated witnesses as follows: Since the previous witnesses W_i are stored at each node v_i, for $i = 1, \ldots, l$, the algorithm uses Relations 3.24 and 3.25 to update the witnesses within the bucket of the update (depending on whether there is an addition or a deletion of an element) and Relation 3.26 to update the witnesses that correspond to every internal node of the tree. Specifically, for an internal node v, that has children $v_1, v_2, \ldots, v_t$, suppose the accumulation value $\chi(v_j)$ of v_j is modified. Then the element collections $\mathcal{X}$ and $\mathcal{X}'$ used in Formula 3.26 are the following:

$$\mathcal{X} = \{h(\chi(v_1)), h(\chi(v_2)), \ldots, h(\chi(v_t))\}\,,$$

and

$$\mathcal{X}' = \{h(\chi(v_1)), h(\chi(v_2)), \ldots, h(\chi(v_{j-1})), h(\chi'(v_j)), h(\chi(v_{j+1})), \ldots, h(\chi(v_t))\}\,.$$

Case 2. $m = \frac{m}{4}$ **or** $n = m$**:** In this case the hash table is rebuilt according to Definition 3.2: If $n = \frac{m}{4}$, then the algorithm builds a data structure D_{h+1} with $m/2$ buckets. Otherwise, i.e., when $n = m$, the algorithm builds a data structure D_{h+1} with $2m$ buckets. Subsequently, it outputs $\mathsf{auth}(D_{h+1})$ and d_{h+1} from information upd output by $\mathsf{update}()$. We recall that upd includes the new witnesses.

By using the same amortized analysis as in Lemma 3.7 (note now that the work that $\mathsf{refresh}()$ does when rebuilding the hash table is $O(m)$—copying information from upd) and not $O(m \log m)$) but Corollary 3.3 instead of Lemma 3.6 in the proof, we can derive the following result:

Lemma 3.17 *By using the rebuilding policy of Definition 3.2, algorithm* $\mathsf{refresh}()$ *of the authenticated data structure scheme* $\mathcal{BHT}$ *has* $O(1)$ *expected amortized access complexity, without precomputed witnesses. With precomputed witnesses, algorithm* $\mathsf{refresh}()$ *has* $O(n^{\epsilon})$ *expected amortized access complexity.*

3.3.1 Queries and verification

We show now how a proof for an element $e \in \mathcal{X}$ (or an element $e \notin \mathcal{X}$) can be constructed. As in the RSA accumulator case, let $H(e) = j$ (bucket assignment for e) and let $v_0, v_1, \ldots, v_l$ be the path from the node that corresponds to bucket j to the root of $T(\epsilon)$. We recall v_{-1} is a fictitious node that stores element e within bucket j such that $v_{-1}, v_0, v_1, \ldots, v_l$ is the path in $T(\epsilon)$ from the node that corresponds to element e to the root of $T(\epsilon)$. We consider two cases, i.e., *membership* and *non-membership* proof:

- *Element e is contained in the hash table.* The proof is the ordered sequence $\pi_0, \pi_1, \ldots, \pi_l$, where π_i is a tuple of *an accumulation value* $\chi()$ and a witness that authenticates every

node of the path $v_{-1}, v_0, \ldots, v_l$ from the element in question e to the root of the tree v_l. Thus, item π_i of proof $\Pi(e)$ $(i = 0, \ldots, l)$ is defined as:

$$\pi_i = \left(\chi(v_{i-1})), \mathsf{W}_{v_{i-1}(v_i)}\right) , \tag{3.27}$$

where $\mathsf{W}_{v_{i-1}(v_i)}$ is defined in Relation 3.21. For simplicity, we set $\alpha_i = \chi(v_{i-1})$ (note that $\chi(v_{-1}) = e$) and

$$\beta_i = \mathsf{W}_{v_{i-1}(v_i)} . \tag{3.28}$$

For example in Figure 3.1, the proof for an element that belongs to bucket of node a (e.g., element 2) consists of the following tuples:

$$\begin{aligned}
\pi_0 &= \left(2, g^{(s+3)(s+7)(s+9)}\right) , \\
\pi_1 &= \left(\chi(a), g^{(h(\chi(b))+s)(h(\chi(c))+s)(h(\chi(d))+s)}\right) , \\
\pi_2 &= \left(\chi(f), g^{(h(\chi(e))+s)(h(\chi(g))+s)(h(\chi(p))+s)}\right) .
\end{aligned}$$

- *Element e is not contained in the hash table.* Let $y_1, y_2, \ldots, y_u$ be the elements contained in bucket j (all different than e). First, output a *membership proof* (as above) for an element y_i in bucket j (note that $H(y_i) = H(e)$). Then, and by running the extended Euclidean algorithm for polynomials, output a non-membership witness

$$\pi_\nu = (\mathsf{A}_e, \mathsf{B}_e, e) , \tag{3.29}$$

where $\mathsf{A}_e, \mathsf{B}_e$ are elements in $\mathbb{G}$ defined in Relation 3.5. Note that $\mathsf{A}_e, \mathsf{B}_e$ have group complexity $O(1)$ (and not *expected* $O(1)$ as in the RSA accumulator case—see Relation 3.3—since they consist of just one group element each) and they are used to prove non-membership of e in the set $\{y_1, y_2, \ldots, y_u\}$.

We now describe the algorithm formally:

Algorithm $\{\Pi(q), \alpha(q)\} \leftarrow \mathsf{query}(q, D_h, \mathsf{auth}(D_h), \mathsf{pk})$**:** Let $e = q$ be the queried element. If e is *contained* in D_h, set $\Pi(q) = (\pi_0, \pi_1, \ldots, \pi_l)$, as in Relation 3.27 and output $\alpha(q) = \mathsf{true}$.

If e is *not contained* in D_h, output a membership proof for some other element y_i in bucket j, such that $H(e) = H(y_i)$. Then output a non-membership proof π_ν for e in bucket j, as defined in Relation 3.29. Set $\Pi(q) = (\Pi(y_i), \pi_\nu)$ and $\alpha(q) = \mathsf{false}$.

Lemma 3.18 *Without precomputed witnesses, algorithm* $\mathsf{query}()$ *of the authenticated data structure scheme* $\mathcal{BHT}$ *has* $O(n^\epsilon \log n)$ *expected access complexity. With precomputed witnesses, algorithm* $\mathsf{query}()$ *has* $O(1)$ *expected access complexity. Moreover, it outputs a proof* $\Pi(q)$ *of* $O(1)$ *group complexity.*

Proof: The proof is the same with Lemma 3.8, but with the following differences:

1. Without precomputed witnesses, a witness cannot be constructed with direct exponentiation, since the trapdoor s is not known. It is constructed with polynomial interpolation as follows: Suppose the witness is $g^{(y_1+s)(y_2+s)\ldots(y_t+s)}$ (where $t = O(n^\epsilon)$). Compute $a_0, a_1, \ldots, a_t$ by using Lemma 3.15 ($O(n^\epsilon \log n)$ complexity). Then output the witness as
$$g^{a_0} \times (g^s)^{a_1} \times \left(g^{s^2}\right)^{a_2} \times \ldots \times \left(g^{s^t}\right)^{a_t},$$
where $g, g^s, \ldots, g^{s^t}$ are contained in the public key. The final task has $O(n^\epsilon)$ access complexity.

2. The proof has group complexity $O(1)$ and not *expected* $O(1)$, due the compactness of the non-membership proof in the bilinear-map accumulator construction (see Relation 3.5).

This completes the proof. □

Algorithm $\{\mathsf{accept}, \mathsf{reject}\} \leftarrow \mathsf{verify}(q, \alpha, \Pi, d_h, \mathsf{pk})$**:** Let the query q refer to element e, i.e., $q = e$. We distinguish two cases:

1. *Membership proof*: In this case it is $\alpha = \mathsf{true}$. The proof Π should contain $\Pi(e) =$

$\pi_0, \pi_1, \ldots, \pi_l$, i.e., the membership proof for element e, where $\pi_i = (\alpha_i, \beta_i)$. The algorithm outputs reject if one of the following is true (note that the verification algorithm is using the bilinear map function $e(.,.)$):

(a) $\alpha_0 \neq e$ (element α_0 is not correct);

(b) $e(\alpha_i, g) \neq e\left(\beta_{i-1}, g^s g^{h(\alpha_{i-1})}\right)$ for some $1 \leq i \leq l$ (false witness);

(c) $e(d_h, g) \neq e\left(\beta_l, g^s g^{h(\alpha_l)}\right)$ (final digest mismatch).

2. *Non-membership proof*: In this case it is $\alpha = \mathsf{false}$. The proof Π in this case contains $\Pi(y) = \pi_0, \pi_1, \ldots, \pi_l$, i.e., the membership proof for an element $y \neq e$, where $\pi_i = (\alpha_i, \beta_i)$ for $i = 0, \ldots, l$. It also contains $\pi_\nu = (\mathsf{A}, \mathsf{B}, r)$, the non-membership proof for e. The algorithm outputs reject if one of the following is true:

(a) $H(e) \neq H(y)$; (e and y do not belong in the same bucket);

(b) The membership proof for y does not verify, i.e., it is

$$\mathsf{reject} \leftarrow \mathsf{verify}(y, \mathsf{true}, \Pi(y), d_h, \mathsf{pk})\,;$$

(c) $r \neq e$ (the data element contained in π_ν for element e is not correct);

(d) $e(\alpha_1, \mathsf{A})\, e(g^s g^r, \mathsf{B}) \neq e(g, g)$ (verification test for non-membership proof of e does not succeed, see Lemma 3.3).

If all the above tests are successful, the algorithm outputs accept.

Lemma 3.19 *Algorithm* verify() *of the authenticated data structure scheme* $\mathcal{BHT}$ *has* $O(1)$ *access complexity.*

Proof: Same as in Lemma 3.9, with the difference that the complexity is not *expected* any more, due to the compactness of the non-membership proof. □

We finally give the results for correctness and security of the scheme $\mathcal{BHT}$:

Lemma 3.20 *The authenticated data structure scheme* $\mathcal{BHT}$ = {genkey, setup, update, refresh, query, verify} *is correct according to Definition 2.4.*

Proof: The proof follows the same logic with the proof of Lemma 3.10. As such, we only show the correctness for the non-membership proof case. Let $y_1, y_2, \ldots, y_u$ be the elements contained in the bucket where e *should belong*. The non-membership proof, as computed by query(), that is needed for verification, is $(\mathsf{A}_e, \mathsf{B}_e, e)$. Therefore verify() does not reject at Item 2c, since $r = e$. Also it does not reject at Item 2d since

$$e\left(\alpha_1, \mathsf{A}_e\right) e\left(g^s g^e, \mathsf{B}_e\right) = e\left(g^{\prod_{j=1}^{u}(y_j+s)}, \mathsf{A}_e\right) e\left(g^s g^e, \mathsf{B}_e\right) = e(g, g)\,,$$

since, by Relation 3.5, $\mathsf{A}_e = g^{\alpha(s)}$ and $\mathsf{B}_e = g^{\beta(s)}$ such that $\left[\prod_{j=1}^{u}(y_j + s)\right] \alpha(s) + (e+s)\beta(s) = 1$. □

Lemma 3.21 *The authenticated data structure scheme* $\mathcal{BHT}$ = {genkey, setup, update, refresh, query, verify} *is secure according to Definition 2.5 and under the q-strong Diffie-Hellman assumption.*

Proof: The proof follows exactly the same logic with the proof of Lemma 3.11: Let k be the security parameter. Output $\mathsf{pk} = \{h(.), (p, \mathbb{G}, \mathcal{G}, e, g), \{g^s, g^{s^2}, \ldots, g^{s^q}\}, \epsilon\}$ and $\mathsf{sk} = s \in \mathbb{Z}_p^*$ by calling algorithm genkey(). Let Adv be a polynomially-bounded adversary. Adv picks an initial collection of n elements $\mathcal{X}$, stored in hash table D_0. Adv outputs an authenticated data structure $\mathsf{auth}(D_0)$, by calling algorithm setup() through oracle access. Then Adv picks a polynomial number of updates—namely, he picks a polynomial number of elements for insertion or deletion. Let D_h be the final hash table, let the updated final element collection be $\mathcal{X}$, and let d_h be the final digest as produced by the adversary through oracle access to algorithm update(). We will compute the probability that **check**() rejects, while verify() accepts, as required by Definition 2.5.

For the case of a *membership proof*, the adversary Adv outputs an incorrect answer $e \notin \mathcal{X}$ and also a proof $\Pi(e) = (\pi_0, \pi_1, \ldots, \pi_l)$ $(l = \lceil \frac{1}{\epsilon} \rceil)$ where $\pi_i = (\alpha_i, \beta_i)$ (see algorithm query()).

Let $v_0, v_1, \ldots, v_l$ be a path of nodes in $T(\epsilon)$ from the bucket referring to e to the root of the tree. We define now the following events, related to the choice of the proof above made by the adversary. Our goal will be to the express the probability that $\mathsf{verify}(e, \mathsf{true}, \Pi(e), d_h, \mathsf{pk})$ accepts and $e \notin \mathcal{X}$ as a function of the following events. Note that d_h is the correct digest of the authenticated data structure:

1. $\mathcal{E}_{0,0}$: The value α_0 picked by Adv are such that $\alpha_0 = e \notin \mathcal{X}$;

2. $\mathcal{E}_j$: For $j = 1, \ldots, l$, the values α_j, α_{j-1} and β_{j-1} picked by Adv are such that

$$e\left(\alpha_j, g\right) = e\left(\beta_{j-1}, g^s g^{h(\alpha_{j-1})}\right) \text{ for all } 1 \leq j \leq l\,.$$

   This event can be partitioned into two mutually exclusive events, i.e., $\mathcal{E}_j = \mathcal{E}_{j,0} \cup \mathcal{E}_{j,1}$ such that

   - $\mathcal{E}_{j,0}$: Value α_j is *not* the correctly formed digest (i.e., an accumulation of the digests of its children) of some node $v_{j-1} \in \mathcal{N}(v_j)$, as defined in Relation 3.20;
   - $\mathcal{E}_{j,1}$: Value α_j is the correctly formed digest of a node $v_{j-1} \in \mathcal{N}(v_j)$, as defined in Relation 3.20.

3. $\mathcal{E}_{l+1,1}$: The values α_l and β_l picked by Adv are such that

$$e\left(\beta_l, g^s g^{h(\alpha_l)}\right) = e(d_h, g)\,.$$

The probability that $\mathsf{verify}()$ accepts, while $e \notin \mathcal{X}$ is the probability

$$\begin{aligned}
& \Pr[\mathcal{E}_{0,0} \cap \mathcal{E}_1 \cap \mathcal{E}_2 \cap \ldots \cap \mathcal{E}_{l+1,1}] \\
= \; & \Pr[\mathcal{E}_{0,0} \cap (\mathcal{E}_{1,0} \cup \mathcal{E}_{1,1}) \cap (\mathcal{E}_{2,0} \cup \mathcal{E}_{2,1}) \cap \ldots \cap \mathcal{E}_{l+1,1}] \\
\leq \; & \Pr[\mathcal{E}_{1,1}|\mathcal{E}_{0,0}] + \Pr[\mathcal{E}_{2,1}|\mathcal{E}_{1,0}] + \Pr[\mathcal{E}_{3,1}|\mathcal{E}_{2,0}] + \ldots + \Pr[\mathcal{E}_{l+1,1}|\mathcal{E}_{l,0}] \\
= \; & \Pr[\mathcal{E}_{1,1}|\mathcal{E}_{0,0}] + \sum_{j=2}^{l+1} \Pr[\mathcal{E}_{j,1}|\mathcal{E}_{j-1,0}]\,. \qquad (3.30)
\end{aligned}$$

First we examine the event $\mathcal{E}_{1,1}|\mathcal{E}_{0,0}$. This event implies that the adversary has found a value $\alpha_0 = e \notin \mathcal{X}$ and a value β_0 such that

$$e\left(\beta_0, g^s g^{h(\alpha_0)}\right) = e\left(g^{\prod_{t=1,\ldots,l'}(s+x_t)}, g\right) ,$$

where $x_1, x_2, \ldots, x_{l'}$ is a subset of the set $\mathcal{X}$. Since $e = h(\alpha_0) \notin \mathcal{X}$, it is $e \notin \{x_1, x_2, \ldots, x_{l'}\}$. By Lemma 3.3 and Assumption 3.2, this probability is $\mathsf{neg}(k)$. Therefore $\Pr[\mathcal{E}_{1,1}|\mathcal{E}_{0,0}] \leq \mathsf{neg}(k)$.

For the remaining events $\mathcal{E}_{j,1}|\mathcal{E}_{j-1,0}$ $(2 \leq j \leq l+1)$, we have:

- $\mathcal{E}_{j-1,0}$ implies that value α_{j-1} is *not* the correctly formed digest of some node $v_{j-2} \in \mathcal{N}(v_{j-1})$, as defined in Relation 3.20, namely that $\alpha_{j-1} \notin \{\chi(v_t) : v_t \in \mathcal{N}(v_{j-1})\}$ which gives $h(\alpha_{j-1}) \notin \{h(\chi(v_t)) : v_t \in \mathcal{N}(v_{j-1})\}$, by the one-to-one property of $h(.)$;

- However, the event $\mathcal{E}_{j,1}$ implies that (1) digest α_j (for $j = l+1$ this is just d_h) is the correctly formed digest of node v_{j-1}; and (2)

$$e\left(\beta_{j-1}, g^s g^{h(\alpha_{j-1})}\right) = e\left(g^{\prod_{v_t \in \mathcal{N}(v_{j-1})}(s+h(\chi(v_t)))}, g\right) .$$

  where $\chi(v_t)$ are the correctly formed digests of the set of neighbors of v_{j-1}.

Since $h(\alpha_{j-1}) \notin \{h(\chi(v_t)) : v_t \in \mathcal{N}(v_{j-1})\}$, by Lemma 3.3 and Assumption 3.2, this probability is $\mathsf{neg}(k)$. Therefore for all $j = 1, \ldots, l+1$, $\Pr[\mathcal{E}_{j,1}|\mathcal{E}_{j-1,0}]$ is $\mathsf{neg}(k)$. Since $l = O(1)$, the total probability is also $\mathsf{neg}(k)$. This concludes the proof for the membership proof.

For the case of a *non-membership proof*, the proof for this case follows exactly the same logic with Lemma 3.11, so it is omitted. □

We continue with the following corollary that is useful in Chapter 5:

Corollary 3.4 *Let $H(e) = j$ and $\Pi(e) = \{(\alpha_i, \beta_i) : i = 0, \ldots, l\}$ be a membership proof for element e. The probability that* $\mathsf{verify}(e, \mathsf{true}, \Pi(e), d_h, \mathsf{pk})$ *accepts and $\beta_0 \neq g^{\prod_{x \in L_j - \{e\}}(s+x)}$ is* $\mathsf{negl}(k)$.

Proof: The event $\beta_0 \neq g^{\prod_{x \in L_j - \{e\}}(s+x)}$ and verify() accepts implies the event $\alpha_1 \neq g^{\prod_{x \in L_j}(s+x)}$ and verify() accepts. Therefore the probability in question is less or equal than the probability $\Pr[\mathcal{E}_{1,0} \cap \mathcal{E}_2 \cap \ldots \cap \mathcal{E}_{l+1,1}]$, since $\mathcal{E}_{1,0}$ exactly the event

$$\alpha_1 \neq g^{\prod_{x \in L_j}(s+x)}.$$

By following the same proof procedure as in Relations 3.30, this can be proved to be $\mathsf{neg}(k)$ as well. □

Theorem 3.4 *Let k be the security parameter and $0 < \epsilon < 1$. Then there exists a publicly-verifiable authenticated data structure scheme $\mathcal{BHT} =$ {genkey, setup, update, refresh, query, verify} for a data structure scheme defined for a dynamic hash table D storing n elements such that:*

1. *It is correct according to Definition 2.4 and secure according to Definition 2.5 and under the bilinear q-strong Diffie-Hellman assumption;*
2. *The access complexity of* setup() *is $O(n)$, outputting an authenticated data structure* auth(D) *of $O(n)$ group complexity;*
3. *The expected amortized access complexity of* update() *is $O(1)$, outputting update information* upd *of $O(1)$ amortized group complexity;*
4. *The expected amortized access complexity of* refresh() *is $O(n^\epsilon)$ (or $O(1)$);*
5. *The expected access complexity of* query() *is $O(1)$ (or $O(n^\epsilon \log n)$), outputting a proof $\Pi(q)$ for a query q of $O(1)$ group complexity;*
6. *The access complexity of* verify() *is $O(1)$.*

Proof: This result follows directly from Lemmata 3.13, 3.14, 3.17, 3.18, 3.19, 3.20 and 3.21. The complexities in the brackets ($O(1)$ for refresh() and $O(n^\epsilon \log n)$ for query()) refer to the case when no precomputed witnesses are used. □

Finally we note here that both constructions (RSA accumulator and bilinear-map accumulator) use the same algorithmic ideas, i.e., the accumulation tree. We could have described a scheme by using an abstract notion of an accumulator and then derive our results by instantiating the abstract solution with the RSA accumulator and the bilinear-map accumulator. However, we chose not to do that because we feel that this would add more complexity in the presentation, for something that can be derived and described a lot easier with no abstraction.

3.3.2 Protocols

Three-party protocol. By using Theorem 2.1 we can easily derive the following corollary that describes the use of the authenticated data structure scheme $\mathcal{BHT}$ of Theorem 3.4 in the three-party model:

Corollary 3.5 *Let k be the security parameter and assume that the bilinear q-strong Diffie-Hellman assumption holds. Then there exists a three-party authenticated data structures protocol (see Protocol 2.1) for verifying (non)-membership queries q on a dynamic hash table storing n elements such that:*

1. *The setup at the source has $O(n)$ access complexity;*
2. *The update at the source has $O(1)$ expected amortized access complexity;*
3. *The space needed at the source has $O(n)$ group complexity;*
4. *The communication between the source and the server has $O(1)$ amortized group complexity;*
5. *The update at the server has $O(n^{\epsilon})$ (or $O(1)$) expected amortized access complexity;*
6. *The query at the server has $O(1)$ (or $O(n^{\epsilon} \log n)$) expected access complexity;*
7. *The space needed at the server has $O(n)$ group complexity;*

8. *The communication between the server and the client has $O(1)$ group complexity;*

9. *The verification at the client has $O(1)$ access complexity;*

10. *For a query q sent by the client to the server at any time (even after updates), let α be an answer and let π be a proof returned by the server. With probability $\Omega(1 - \mathsf{neg}(k))$, the client accepts the answer α if and only if α is correct.*

Two-party protocol. As a corollary of Lemma 3.12 (the proof follows exactly the same techniques), we can state a similar assumption for the authenticated data structure scheme $\mathcal{BHT}$:

Corollary 3.6 *Assumption 2.1 is true for the authenticated data structure scheme $\mathcal{BHT}$. Moreover, for every update u, $|Q_u|$ has $O(1)$ amortized complexity.*

By Theorems 2.2 and 3.4 and Corollary 3.6, we can now state the final result for the two-party model:

Corollary 3.7 *Let k be the security parameter and assume that the bilinear q-strong Diffie-Hellman assumption holds. Then there exists a two-party authenticated data structures protocol (see Protocol 2.2) for verifying (non)-membership queries q on a dynamic hash table storing n elements such that:*

1. *When precomputed witnesses are used, the protocol requires one round of interaction during updates that cause the hash table to be rebuilt (see Definition 3.2); When no precomputed witnesses are used, it requires one round of interaction during updates;*

2. *The setup at the client has $O(n)$ access complexity;*

3. *The update at the client has $O(1)$ expected amortized access complexity;*

4. *The verification at the client has $O(1)$ access complexity;*

5. *The space needed at the client has $O(1)$ group complexity;*

6. *The communication between the client and the server has $O(1)$ amortized group complexity during updates and $O(1)$ group complexity during queries;*

7. *The update at the server has $O(n^\epsilon)$ (or $O(n^\epsilon \log n)$) expected amortized access complexity;*

8. *The query at the server has $O(1)$ (or $O(n^\epsilon \log n)$) expected access complexity;*

9. *The space needed at the server has $O(n)$ group complexity;*

10. *For a query q sent by the client to the server at any time (even after updates), let α be an answer and let π be a proof returned by the server. With probability $\Omega(1 - \mathsf{neg}(k))$, the client accepts the answer α if and only if α is correct.*

3.4 Complexity limitations

In this chapter, we proposed a new, provably secure, cryptographic construction for verifying hash table queries over a dynamic set. We use nested cryptographic accumulators on a tree of constant depth to achieve *constant* query and verification costs and *sublinear* update costs. Our results are applicable to both the two-party and three-party data authentication models. We use our method to authenticate general set-membership queries and overall improve over previous techniques that use cryptographic accumulators, reducing the main complexity measures to constant, yet keeping sublinear update complexity.

An important open problem is whether one can achieve logarithmic update cost and still keep the communication complexity constant. There has been no such solution to-date. In particular, no method is known that can construct constant-size accumulator proofs (witnesses) in logarithmic time. In Chapter 6 however, which is the full version of the work of Papamanthou et al. [91], we describe a solution for this problem that uses a cryptographic primitive that, unfortunately, is not known to exist yet. On the other hand, we believe that doing even better, i.e., achieving constant complexity for *all* the complexity measures seems

to be unfeasible due to the $\Omega(\log n / \log \log n)$ memory checking lower bound [35] on query complexity (the sum of read and write complexity). This result, however, motivates seeking more general lower bounds for authenticated data structures (similar directions have been followed in the lower bound works of Dwork et al. [35] and Tamassia and Triandopoulos [106]): given any cryptographic primitive, what is the best we can do in terms of complexity?

Finally, it would be interesting to modify our schemes to obtain non-amortized bounds for updates using for example Overmar's global rebuilding technique [87].

Chapter 4

Authenticated structures based on lattices

Lattices, an infinite-sized set of specially constructed vectors, is a mathematical tool that made its first appearance in cryptography with Ajtai's seminal result [3], showing the construction of one-way functions based on hard lattices problems. Since then, lattices have been proven to enjoy appealing properties that have made their application to cryptography very promising. Such properties include the seemingly resistance of lattice-based assumptions to quantum algorithms [98]—as opposed to other assumptions such as factoring—as well as their worst-case to average-case reductions [99], namely the existence of polynomial-time algorithms that can transform a solution to a *random* instance of a certain problem into a solution to *any* (worst-case) instance of another lattice problem. As such, many cryptographic primitives, such as public-key encryption schemes (e.g., see the work of Peikert [95]) and collision-resistant hash functions (e.g., see the work of Lyubashevsky and Micciancio [71]), based on lattice assumptions have been derived during the last decade. Even more significantly, the long-standing open problem of *fully-homomorphic encryption* was settled with a lattice-based construction in 2009 by Gentry [43]. Finally, lattice-based constructions appear to be efficient in practice due to the extensive use of linear algebra (therefore also easily *parallelizable*), and have also led to the deployment of lattice-based cryptographic systems (e.g., see the NTRU system by Hoffstein et al. [57]).

In this chapter we present the first authenticated data structure based on lattices, and

specifically a lattice-based authenticated table of highly desirable complexity features, such as *update optimality* and *parallelism* (i.e., the constructed authenticated table admits parallel algorithms). Specifically, we design the first authenticated data structure based on lattices, the update complexity of which is $O(1)$, improving in this way the $O(\log n)$ update bounds of previous constructions, such as the Merkle tree, and while retaining efficient $O(\log n)$ proof complexity. Moreover, the used lattice-based cryptographic primitive lends itself to a natural notion of parallelism: As such, we describe *parallel* versions of our authenticated data structure algorithms, yielding the first parallel *online memory checker* [15] with $O(1)$ query complexity using $O(\log n)$ checkers in the CREW model (a parallel model of computation where processors can read *concurrently* but can write only *exclusively*) and without using a secret key setting, i.e., there is only need for small *reliable* but not *secret* memory (as opposed to [54]). We base the security of our constructions on the difficulty of approximating the gap version of the shortest vector problem in lattices (GAPSVP) within polynomial factors.

The key idea used here is to combine the simplicity of a Merkle tree [77] with a special property of lattice-based hash functions, which we establish and call *repeated linearity*. Roughly speaking, this property allows using the output of one invocation of the hash function, as an input to another invocation of the function, without losing "structure". This observation, in the authenticated data structures setting, turns out to be crucial in achieving *constant* update complexity (as well as parallel algorithms), while keeping all the remaining complexity bounds *logarithmic*. This is a trade-off that, to the best of our knowledge, has not been achieved so far in the literature—and is feasible due to the use of lattices: For example, for a table data structure of n entries, the constructions of Bellare and Micciancio [11], the authenticated data structure of Papamanthou et al. [90] and the memory checker of Dwork et al. [35] have $O(1)$ update but $\Omega(n^{\epsilon})$ proof (or query) complexity, whereas hierarchical hashing constructions such as the one of Blum et al. [15] and the one of Goodrich and Tamassia [48] impose $O(\log n)$ bounds on *all* the complexity measures, which is to be expected, given the lower bound for hash-based authenticated data structures by Tamassia

and Triandopoulos [106].

The data structure we are considering in this chapter is a *dynamic table* of size n, read and written through indices $0, \ldots, n-1$. We base the security of our construction on the hardness of the GAPSVP problem in lattices [78], which has its own significance given recent attacks on collision-resistant functions such as MD-5 [103]. We note that our construction requires an one-time $O(n \log n)$ preprocessing, which is is however amortized—in comparison with other works (see Table 4.1)— after $\Omega(n \log n)$ updates.

Overview of the solution. Our authenticated data structure scheme, denoted with $\mathcal{LBT}$ in Table 4.1, can be seen as a generalization of the Merkle tree and related hierarchical hashing constructions [15, 48, 81]. By exploiting a property of lattice-based hash functions (which we call repeated linearity) over a typical Merkle tree, we depart from black-box use of *generic collision-resistant* hash functions (e.g., MD-5 or SHA-256) in the authenticated data structures setting. As a consequence, and in the Merkle tree paradigm, the digest of a tree node v can be expressed as the "sum" of well-defined functions (called *partial digests*) applied to data stored at the leaves of v's subtree (Theorem 4.3). Exploiting this property enables constant update complexity as well as deriving parallel algorithms. It may also be of general interest and have other applications. A comparison of our solution with existing work is given in Table 4.1.

We now give the formal definition of the underlying data structure scheme, for which the authenticated data structure scheme $\mathcal{LBT}$ is designed.

Table 4.1: Asymptotic access and group complexities of various authenticated data structure schemes (see Definition 2.3) for a dynamic table of n entries. Parameter $0 < \epsilon < 1$ is a constant and GAPSVP is the gap shortest vector problem in lattices (Definition 4.1). In all schemes, the authenticated structure has group complexity $O(n)$ and genkey() has $O(1)$ complexity. Note that [90] is the published conference version of Chapter 3. The acronyms of the other assumptions can be found in Table 3.1. All presented schemes in the table are publicly verifiable.

	[15, 48, 75, 81]	[11]	[83]	[23, 101]	[51]	[90]	$\mathcal{LBT}$
setup()	n	n	n	n	n	n	$n \log n$
update()	$\log n$	1	1	1	n^ϵ	1	1
refresh()	$\log n$	1	n	$n \log n$	n^ϵ	1	$\log n$
query()	$\log n$	n	1	1	n^ϵ	n^ϵ	$\log n$
verify()	$\log n$	n	1	1	1	1	$\log n$
proof $\Pi(q)$	$\log n$	n	1	1	1	1	$\log n$
info. upd	1	1	1	1	n^ϵ	1	1
assumption	Generic CR	D. Log	B. q-DH	Strong RSA			GAPSVP

The data structure scheme. Let $\mathbf{T}$ be a dynamic table of n indices, storing values $\mathbf{T}[1], \mathbf{T}[2], \ldots, \mathbf{T}[n]$. The data structure scheme $\{\textbf{query}(), \textbf{update}(), \textbf{check}()\}$ (Definition 2.2) for a dynamic table $\mathbf{T}$ is as follows:

1. $\mathbf{T}[i] \leftarrow \textbf{query}(i, \mathbf{T})$: Given an index $1 \leq i \leq n$, return $\mathbf{T}[i]$. Answering this query has $O(1)$ complexity;

2. $\mathbf{T}' \leftarrow \textbf{update}(i, y, \mathbf{T})$: Given an index $1 \leq i \leq n$, set $\mathbf{T}[i] := y$. The complexity for this task is $O(1)$;

3. $\{\mathsf{accept}, \mathsf{reject}\} \leftarrow \textbf{check}(i, y, \mathbf{T})$: If $\mathbf{T}[i] \neq y$ return reject. Else return accept.

4.1 Lattice definitions

We start with some basic definitions related to lattices.We use upper case bold letters to denote matrices, e.g., $\mathbf{B}$, lower case bold letters to denote vectors, e.g., $\mathbf{b}$, and lower case italic letters to denote scalars. Finally, for a vector $\mathbf{x} = [\mathbf{x}_1 \ \mathbf{x}_2 \ \ldots \ \mathbf{x}_k]^\mathsf{T}$ (note that T as an exponent in the vector notation denotes the *transpose* vector), $\|\mathbf{x}\|$ denotes the Euclidean norm of $\mathbf{x}$, i.e., $\|\mathbf{x}\| = (\mathbf{x}_1^2 + \mathbf{x}_2^2 + \ldots + \mathbf{x}_k^2)^{1/2}$.

4.1.1 What is a lattice?

Given the security parameter k, a full-rank k-dimensional lattice is defined as the infinite-sized set of all vectors produced as the integer combinations

$$\left\{ \sum_{i=1}^{k} x_i \mathbf{b}_i : x_i \in \mathbb{Z},\ 1 \leq i \leq k \right\},$$

where $\mathbf{B} = \{\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_k\}$ is the *basis* of the lattice and $\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_k$ are linearly independent, all belonging to $\mathbb{R}^k$. We denote the lattice produced by $\mathbf{B}$ (i.e., the set of vectors) with $L(\mathbf{B})$. A well-known difficult problem in lattices is the approximation within a polynomial factor of the *shortest* vector in a lattice (SVP problem). Namely, given a lattice $L(\mathbf{B})$

produced by a basis $\mathbf{B}$, approximate up to a polynomial factor in k the shortest (in an Euclidean sense) vector in $L(\mathbf{B})$, the length of which we denote with $\lambda(\mathbf{B})$. A similar problem in lattices is the "gap" version of the shortest vector problem (GAPSVP_γ), the difficulty of which is useful in our context:

Definition 4.1 (Problem GAPSVP_γ) *An input to GAPSVP_γ is a k-dimensional lattice basis $\mathbf{B}$ and a number d, where k is the security parameter. In YES inputs $\lambda(\mathbf{B}) \leq d$ and in NO inputs $\lambda(\mathbf{B}) > \gamma \times d$, where $\gamma \geq 1$.*

We note that, for exponential values of γ, i.e., $\gamma = 2^{O(k)}$, one can use the LLL algorithm [65] and decide the above problem in polynomial time. The difficult version of the problem arises for polynomial γ, for which no efficient algorithm is known to date, even for factors slightly smaller than exponential [99], i.e., very big polynomials. Moreover, for polynomial factors, there is no proof that this problem is NP-hard[1], which makes the polynomial approximation cryptographically interesting as well. Therefore, a well-accepted assumption on which the security of our scheme is based is as follows:

Assumption 4.1 (Hardness of GAPSVP_γ) *Let GAPSVP_γ be an instance of the gap version of the shortest vector problem in lattices, as defined in Definition 4.1 and k be the security parameter. There is no polynomial-time algorithm for solving GAPSVP_γ for $\gamma = \mathsf{poly}(k)$, except with negligible probability $\mathsf{neg}(k)$.*

4.1.2 Reductions

After Ajtai's seminal work [3] where an one-way function based on hard lattices problem is presented, Goldreich et al. [44] presented a variation of the function, providing at the same time collision resistance. Based on this collision resistant hash function, Micciancio and Regev [78] described a generalized version of it, a modification of which we are using in

[1] In specific, as outlined in [99], the current state of knowledge indicates that for $\gamma > \sqrt{k/\log k}$, it is unlikely that this problem is NP-hard and no efficient algorithm is known to date.

our construction. The security of the hash function is based on the difficulty of the *small integer solution* problem (SIS):

Definition 4.2 (Problem $\mathsf{SIS}_{q,m,\beta}$) *Given an integer q, a matrix $\mathbf{M} \in \mathbb{Z}_q^{k \times m}$ and a real β, find a non-zero integer vector $\mathbf{z} \in \mathbb{Z}^m \backslash \{\mathbf{0}\}$ such that $\mathbf{Mz} = \mathbf{0} \mod q$ and $\|\mathbf{z}\| \leq \beta$.*

Note that at least one solution to the above problem exists when $\beta \geq \sqrt{m} q^{k/m}$ and $m > k$ [78]. Moreover, if $q \geq 4\sqrt{m} k^{1.5} \beta$, we will see that such a solution is difficult to find. We continue with the definition of SIS', where the solution vector is required to have at least one odd coordinate:

Definition 4.3 (Problem $\mathsf{SIS}'_{q,m,\beta}$) *Given an integer q, a matrix $\mathbf{M} \in \mathbb{Z}_q^{k \times m}$ and a real β, find an integer vector $\mathbf{z} \in \mathbb{Z}^m \backslash 2\mathbb{Z}^m$ such that $\mathbf{Mz} = \mathbf{0} \mod q$ and $\|\mathbf{z}\| \leq \beta$.*

For odd q, there is a polynomial-time reduction from $\mathsf{SIS}'_{q,m,\beta}$ to $\mathsf{SIS}_{q,m,\beta}$ [78]:

Lemma 4.1 (Reduction from $\mathsf{SIS}'_{q,m,\beta}$ to $\mathsf{SIS}_{q,m,\beta}$ [78]) *For any odd integer $q \in 2\mathbb{Z} + 1$ and SIS' instance $I = (q, \mathbf{M}, \beta)$, if I has a solution as an instance of SIS, then it has a solution as an instance of SIS'. Moreover, there is a polynomial-time algorithm that on input a solution to a SIS instance I, outputs a solution to the same SIS' instance I.*

As proved by Micciancio and Regev [78], by choosing certain parameters, GAPSVP_γ can be reduced to SIS' (derived by combining Lemma 5.22 and Theorem 5.23 from the work of Micciancio and Regev [78]):

Lemma 4.2 (Reduction from GAPSVP_γ to $\mathsf{SIS}'_{q,m,\beta}$ [78]) *Let $\beta, m, q = k^{O(1)}$ be values that are polynomially-bounded, with $q \geq 4\sqrt{m} k^{1.5} \beta$ and $\gamma = 14\pi\sqrt{k}\beta$. Then there is a probabilistic polynomial-time reduction from solving GAPSVP_γ in the worst case to solving $\mathsf{SIS}'_{q,m,\beta}$ on the average with non-negligible probability.*

A direct application of Lemma 4.1 and Lemma 4.2 gives the following result.

Theorem 4.1 *Let $q = k^{O(1)}$ be an odd positive integer. For any polynomially-bounded values $\beta, m = k^{O(1)}$, with $q \geq 4\sqrt{m}k^{1.5}\beta$ and $\gamma = 14\pi\sqrt{k}\beta$, there is a probabilistic polynomial-time reduction from solving GAPSVP_γ in the worst case to solving $\mathsf{SIS}_{q,m,\beta}$ on the average with non-negligible probability.*

Theorem 4.1 states that if there is an algorithm that solves an average (i.e., $\mathbf{M} \in \mathbb{Z}_q^{k\times m}$ is chosen uniformly at random) instance of $\mathsf{SIS}_{q,m,\beta}$, for an odd q, $q \geq 4\sqrt{m}k^{1.5}\beta$ and $\gamma = 14\pi\sqrt{k}\beta$, then, this algorithm can be used to solve any instance of GAPSVP_γ.

4.1.3 Lattice-based hash function

Let $m = 2k\log q$ and $\beta = \delta\sqrt{m}$, where δ is $\mathsf{poly}(k)$. Note that $\log\delta = O(\log k)$. We also require $q \geq 4\sqrt{m}k^{1.5}\beta = 8k^{2.5}\delta\log q$. It is easy to see that given k and δ there is always a $q = O(k^{2.5}\delta\log k)$ to satisfy the above constraints—since δ is $\mathsf{poly}(k)$, the bit-size of q is $O(\log k)$. The collision resistant hash function that we are using is a generalization of the function presented by Micciancio and Regev [78], where $\delta = O(1)$ (in the security parameter) is used instead. In our construction we use bigger values for δ. Namely the value that we use to bound the norm of the solution vector can be up to $\mathsf{poly}(k)$. This was observed in the original definition of Ajtai's one-way function [3], i.e., that the input vector can contain larger values (but not so large), and was also noted in its extension that achieves collision resistance [44]. This remark is very useful in our context and implies that, the larger value one picks for β, the larger the modulus q should be so that security is guaranteed (still q's bit size is $O(\log k)$).

Let now $\mathbf{M} \in \mathbb{Z}_q^{k\times m}$ be a $k \times m$ matrix that is chosen uniformly at random. We can define the function $h_\mathbf{M} : \mathbb{Z}^m \to \mathbb{Z}_q^k$ as $h_\mathbf{M}(\mathbf{x}) = \mathbf{M}\mathbf{x} \mod q$, where $\|\mathbf{x}\| \leq \beta$ and the modulo operation is taken component-wise. The above function is collision resistant based on the difficulty of $\mathsf{GAPSVP}_{14\pi\sqrt{k}\beta}$:

Theorem 4.2 (Strong collision resistance) *Let $m = 2k\log q$, $\beta = \delta\sqrt{m}$ and q be an*

odd positive integer such that $q \geq 4\sqrt{m}k^{1.5}\beta$. *Let also* $\mathbf{M} \in \mathbb{Z}_q^{k \times m}$ *be a* $k \times m$ *matrix that is chosen uniformly at random. If there is a polynomial-time algorithm that finds two vectors* $\mathbf{x}, \mathbf{y} \in \{0, 1, \ldots, \delta\}^m$ *and* $\mathbf{x} \neq \mathbf{y}$ *such that* $\mathbf{Mx} = \mathbf{My} \mod q$, *then there is a polynomial-time algorithm to solve any instance of* $\mathsf{GAPSVP}_{14\pi\delta\sqrt{km}}$.

Proof: Suppose there is an algorithm that finds $\mathbf{x}, \mathbf{y} \in \{0, 1, \ldots, \delta\}^m$ with $\mathbf{x} \neq \mathbf{y}$ such that $\mathbf{Mx} = \mathbf{My} \mod q$. Therefore the non-zero vector $\mathbf{z} = \mathbf{x} - \mathbf{y}$, which also has norm $\|\mathbf{z}\| \leq \beta$, since its coordinates are between $-\delta$ and $+\delta$, comprises a solution to the problem $\mathsf{SIS}_{q,m,\beta}$ (note that matrix $\mathbf{M}$ by construction is chosen uniformly at random). By Theorem 4.1, this can be used to solve GAPSVP_γ for $\gamma = 14\pi\sqrt{k}\beta$. Setting $\beta = \delta\sqrt{m}$ we get the desired result. □

Since $\delta = \mathsf{poly}(k)$, γ is also $\mathsf{poly}(k)$ and therefore the presented hash function is secure, by Assumption 4.1. We can now extend the function h to accept two inputs as follows: Denote with $\mathbb{T}^{\delta,+}$ the set of all $m \times 1$ ($m = 2k \log q$) vectors such that their last $k \log q$ entries are zero and the remaining entries are in $\{0, 1, \ldots, \delta\}$ and analogously with $\mathbb{T}^{\delta,-}$ the set of all $m \times 1$ vectors such that their first $k \log q$ entries are zero and the remaining entries are in $\{0, 1, \ldots, \delta\}$:

Definition 4.4 (Lattice-based hash function with two inputs) *We define the function* $h_{\mathbf{M},\delta} : \mathbb{T}^{\delta,+} \times \mathbb{T}^{\delta,-} \rightarrow \mathbb{Z}_q^k$ *as* $h_{\mathbf{M},\delta}(\mathbf{x}, \mathbf{y}) = \mathbf{M}(\mathbf{x} + \mathbf{y}) \mod q$, *where* $\mathbf{x}, \mathbf{y} \in \{0, 1, \ldots, \delta\}^m$.

Note that we use both $\mathbf{M}$ and δ as subscripts for the function. Similarly as in Theorem 4.2, this function is strong collision resistant, i.e., if there is a polynomial-time algorithm that finds $(\mathbf{x}_1, \mathbf{y}_1) \in (\mathbb{T}^{\delta,+} \times \mathbb{T}^{\delta,-})$ and $(\mathbf{x}_2, \mathbf{y}_2) \in (\mathbb{T}^{\delta,+} \times \mathbb{T}^{\delta,-})$ with $(\mathbf{x}_1, \mathbf{y}_1) \neq (\mathbf{x}_2, \mathbf{y}_2)$ such that $\mathbf{M}(\mathbf{x}_1 + \mathbf{y}_1) = \mathbf{M}(\mathbf{x}_2 + \mathbf{y}_2) \mod q$ then there is a polynomial-time algorithm that solves GAPSVP_γ for polynomial γ. To see that, note that the vector $\mathbf{x}_1 - \mathbf{x}_2 + \mathbf{y}_1 - \mathbf{y}_2$ has coordinates in $\{0, 1, \ldots, \delta\}$, since, by the definition of $\mathbb{T}^{\delta,+}$ and $\mathbb{T}^{\delta,-}$, the entries of $\mathbf{x}_1 - \mathbf{x}_2$ and $\mathbf{y}_1 - \mathbf{y}_2$ do not overlap.

Time and space complexity of hash function. In this paragraph we analyze the time and space complexity of the used hash function. Since the modulus q has $O(\log k)$ bits, our hash function is described with a $k \times 2k \log q$ matrix of $O(\log k)$-bit entries. Therefore the space complexity is $O(k^2 \log^2 k)$ bits. Given now an input $x \in \{0, 1, \ldots, \delta\}^{2k \log q}$, we can compute $h_{\mathbf{M},\delta}(x)$ in $O(k^2 \log^2 k \log^2 \log k)$ time. To see that, an application of the hash function requires the computation of k internal products between vectors of $2k \log q$ entries, and each multiplication in the internal product is a multiplication in $\mathbb{Z}_q$, which can be computed in $O(\log k \log^2 \log k)$ time using FFT [29]. This makes the total time equal to $O(k^2 \log^2 k \log^2 \log k)$.

4.1.4 Parallel models of computation

As we mentioned in the beginning of this chapter, we also give *parallel versions* of our lattice-based authenticated data structures algorithms. We use the PRAM model (parallel random access machine) and specifically EREW PRAM, CREW PRAM and CRCW PRAM. We recall the definition of these models below:

1. EREW: This model allows all processors to read and write *exclusively* at the same time. Therefore no conflicts need to be resolved;

2. CREW: This model allows all processors to read *concurrently* and write *exclusively* at the same time. Read conflicts are resolved with $O(1)$ complexity;

3. CRCW: This model allows all processors to read *concurrently* and write *concurrently* at the same time. Read and write conflicts are resolved with $O(1)$ complexity.

Note that EREW requires minimal assumptions, CREW requires a stronger assumption (as there is a need to resolve read conflicts) and CRCW requires the strongest assumptions since both read and write conflicts need to be resolved. Ways to resolve conflicts in the PRAM model have been extensively studied by the literature. A great introduction to

most fundamental results related to the PRAM model of computation as well as to parallel algorithms is given in the book of JaJa [59].

4.2 Main construction

In this section we present our update-optimal authenticated data structure scheme for a dynamic table, i.e., the scheme $\mathcal{LBT} = \{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$. We recall that the data structure for which we describe an authenticated data structure scheme for is a table $\mathbf{T}$ that consists of n indices $1, 2, \ldots, n$, supporting index queries and index updates.

A direct solution for this problem would be to use a Merkle tree with some collision-resistant hash function (e.g., SHA-2, see first column of Table 4.1), which would bear logarithmic complexities in all the complexity measures—also inherently enforcing *sequential computations*. Here we build an authenticated structure for this data structure that uses the lattice-based hash function introduced in Section 4.1 and also supports constant complexity updates, allowing at the same time a great deal of *parallelism*.

4.2.1 Algebraic tools

We now discuss some algebraic tools to be used in our construction. Without loss of generality, assume that q, the modulus is a power of two:

Definition 4.5 (Binary representation) *Define* $f(x) = [\mathbf{f}_0\ \mathbf{f}_1\ \ldots\ \mathbf{f}_{\log q-1}]^\mathsf{T} \in \{0,1\}^{\log q}$ *to be the binary representation of* $x \in \mathbb{Z}_q$. *Namely,* $x = \sum_{i=0}^{\log q-1} \mathbf{f}_i 2^i \mod q$.

Definition 4.6 (Radix-2 representation) *Define* $g(x) = [\mathbf{f}_0\ \mathbf{f}_1\ \ldots\ \mathbf{f}_{\log q-1}]^\mathsf{T} \in \mathbb{Z}_q^{\log q}$ *to be* some *radix-2 representation of* $x \in \mathbb{Z}_q$. *Namely,* $x = \sum_{i=0}^{\log q-1} \mathbf{f}_i 2^i \mod q$.

By "some" radix-2 representation we mean that the function $g : \mathbb{Z}_q \to \mathbb{Z}_q^{\log q}$ is "one-to-many". For example, for $q = 16$, $x = 7$, possible values for $g(x)$ can be $[0\ 1\ 1\ 1]^\mathsf{T}$ (the usual

binary representation), $[0\ -2\ 0\ -1]^{\mathsf{T}}$ or $[-2\ 2\ 0\ -1]^{\mathsf{T}}$ (and many more). We now give an important result for our construction:

Lemma 4.3 *For any $x_1, x_2, \ldots, x_t \in \mathbb{Z}_q$ there exist a radix-2 representation $g(.)$ such that $g(x_1 + x_2 + \ldots + x_t \mod q) = f(x_1) + f(x_2) + \ldots + f(x_t) \mod q$. Moreover it is $g(x_1 + x_2 + \ldots + x_t \mod q) \in \{0, \ldots, t\}^{\log q}$.*

Proof: Let $\mathbf{x}_i = f(x_i)$ be the binary representation of x_i for $i = 1, \ldots, t$. Then

$$\sum_{i=1}^{t} \mathbf{x}_i = \left[\sum_{i=1}^{t} \mathbf{x}_{i0} \ \sum_{i=1}^{t} \mathbf{x}_{i1} \ldots \sum_{i=1}^{t} \mathbf{x}_{i(k-1)}\right]^{\mathsf{T}} \mod q\,.$$

The resulting vector is a radix-2 representation of

$$\left(\sum_{i=1}^{t} \mathbf{x}_{i0}\right) \times 2^0 + \left(\sum_{i=1}^{t} \mathbf{x}_{i1}\right) \times 2^1 + \ldots + \left(\sum_{i=1}^{t} \mathbf{x}_{i(k-1)}\right) \times 2^{k-1} \mod q\,,$$

which can be written as

$$\sum_{j=0}^{k-1} \mathbf{x}_{1j} \times 2^j + \sum_{j=0}^{k-1} \mathbf{x}_{2j} \times 2^j + \ldots + \sum_{j=0}^{k-1} \mathbf{x}_{tj} \times 2^j = x_1 + x_2 + \ldots + x_t \mod q.$$

Therefore there exists a radix-2 representation g such that $g(x_1 + x_2 + \ldots + x_t \mod q) = f(x_1) + f(x_2) + \ldots + f(x_t) \mod q$. Finally note that since $g(.)$ is the sum of t binary representations, it cannot contain a entry that is greater than t. □

Lemma 4.3 is useful in the following sense: Given two binary representations of x_1 and x_2, namely f_1 and f_2, a radix-2 representation of $x_1 + x_2$ is $f_1 + f_2$. Definitions 4.5 and 4.6 and also Lemma 4.3 (see Corollary 4.1) can be naturally extended for vectors $\mathbf{x} \in \mathbb{Z}_q^k$: For $i = 1, \ldots, k$, $\mathbf{x}_i$ is mapped to the respective $\log q$ entries $f(\mathbf{x}_i)$ (or $g(\mathbf{x}_i)$) in the resulting vector $f(\mathbf{x})$ (or $g(\mathbf{x})$). Therefore we have the following:

Corollary 4.1 *For any $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_t \in \mathbb{Z}_q^k$ there exist a radix-2 representation $g(.)$ such that $g(\mathbf{x}_1 + \mathbf{x}_2 + \ldots + \mathbf{x}_t \mod q) = f(\mathbf{x}_1) + f(\mathbf{x}_2) + \ldots + f(\mathbf{x}_t) \mod q$. Moreover it is $g(\mathbf{x}_1 + \mathbf{x}_2 + \ldots + \mathbf{x}_t \mod q) \in \{0, \ldots, t\}^{k \log q}$.*

To constrain the inputs to our hash function, we need the following definition:

Definition 4.7 *Let $x \in \mathbb{Z}_q^k$. We say that the radix-2 representation $g(x) \in \mathbb{Z}_q^{k \log q}$ is δ-*admissible *if and only if $g(x) \in \{0, 1, \ldots, \delta\}^{k \log q}$.*

4.2.2 Algorithms of the scheme

We now describe the algorithms of the scheme $\mathcal{LBT}$ (see Definition 2.3). All expressions below are reduced modulo q, i.e., we work in $\mathbb{Z}_q$.

Algorithm $\{\mathsf{sk}, \mathsf{pk}\} \leftarrow \mathsf{genkey}(1^k)$: On input the security parameter k, this algorithm computes an odd number $q = O(k^{2.5}\delta \log k)$, for some $\delta = n = \mathsf{poly}(k)$. Namely we set δ to be equal to the size of the table, n. Then it samples $\mathbf{M} \in \mathbb{Z}_q^{k \times m}$ uniformly at random, where $m = 2k \log q$. It sets $\mathsf{sk} = \emptyset$ and $\mathsf{pk} = \{\mathbf{M}, q\}$, i.e., there is no secret (trapdoor information) in our scheme. The access complexity of this algorithm is $O(1)$.

Lattice-based digests. Before we describe algorithm $\mathsf{setup}()$, we describe how we define the lattice-based digests on the table $\mathbf{T}$, by using the hash function of Definition 4.4. Let D_0 be the initial state of our table, storing values $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n \in \mathbb{Z}_q^k$. Let T be the binary tree of ℓ levels on top of the values $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$—recall we have assumed that $n = 2^\ell$, and r be the root of tree T. By convention, the root of the tree lies at level 0 and the leaves of the tree lie at level ℓ. For every leaf node v_i of the tree, $i = 1, \ldots, n$, the digest $d(v_i)$ is defined as $d(v_i) = \mathbf{x}_i$. Then, for any internal node u, with left child v and right child w, by using the hash function $h_{\mathbf{M},n}(\mathbf{x}, \mathbf{y})$ given in Definition 4.4 in a recursive way, the digest $d(u)$ of node u can be defined as

$$d(u) = h_{\mathbf{M},n}(\mathbf{U}g(d(v)), \mathbf{D}g(d(w))) = \mathbf{M}\left[\mathbf{U}g(d(v)) + \mathbf{D}g(d(w))\right] , \tag{4.1}$$

where $g(d(v))$ and $g(d(w))$ are *some* n-admissible radix-2 representations of $d(v)$ and $d(w)$, i.e., by Definition 4.4, it must be $g(d(v)), g(d(w)) \in \{0, 1, \ldots, n\}^{k \log q}$.

In the above relations, matrices $\mathbf{U}$ and $\mathbf{D}$ are special matrices such that multiplying matrices $\mathbf{U}$ and $\mathbf{D}$ with a vector in $\{0, 1, \ldots, n\}^{k \log q}$ doubles the dimension of the vector by shifting its entries accordingly and by filling the vacant entries with zeros. This operation is used to prepare the vectors in the appropriate input format for the hash function. More formally, $\mathbf{U} = [\mathbf{I}_{k \log q}\ \mathbf{O}_{k \log q}]^\mathsf{T}$ and $\mathbf{D} = [\mathbf{O}_{k \log q}\ \mathbf{I}_{k \log q}]^\mathsf{T}$, where $\mathbf{I}_l$ denotes the square identity

matrix of dimension l and $\mathbf{O}_l$ denotes the square zero matrix of dimension l. Indeed, it easy to see that for all $\mathbf{x} \in \{0, 1, \ldots, n\}^{k \log q}$ it is $\mathbf{Ux} \in \mathbb{T}^{n,+}$ and $\mathbf{Dx} \in \mathbb{T}^{n,-}$, where $\mathbb{T}^{n,+}$ and $\mathbb{T}^{n,+}$ are defined in Section 4.1.

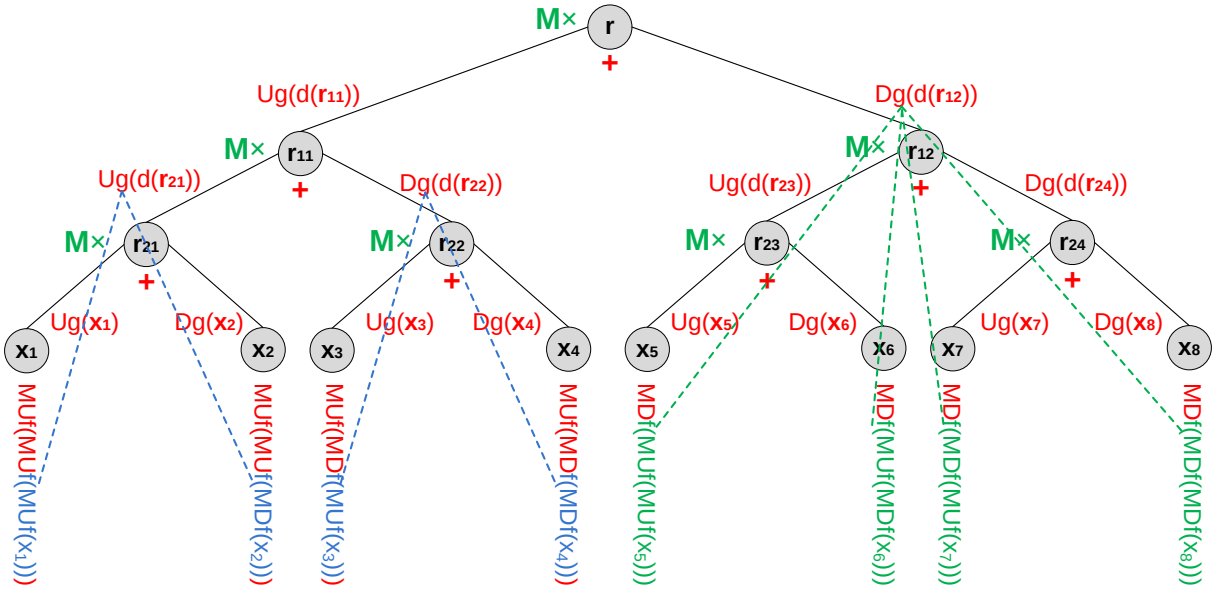

Figure 4.1: Tree T built on top of a table with 8 values $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_8$. After producing an *n-admissible* radix-2 $g(.)$ representation of the children digests, we multiply with either $\mathbf{U}$ or $\mathbf{D}$, then we add the two resulting digests and we compute the hash function on them by multiplying with $\mathbf{M}$. At the leaves of the tree we show the terms that correspond to each index, as computed by Theorem 4.3 (i.e., the *partial digests* of the root r with reference to every value at the table). The $g(.)$ representation of the internal nodes are indicated with dashed lines (see Definition 4.9). Note that the $g(.)$ representations of the internal nodes are the sum of specific $f(.)$ representations of the leaves, for example, $g(d(r_{12})) = f(\mathbf{L}f(\mathbf{L}f(\mathbf{x}_5))) + f(\mathbf{L}f(\mathbf{R}f(\mathbf{x}_6))) + f(\mathbf{R}f(\mathbf{L}f(\mathbf{x}_7))) + f(\mathbf{R}f(\mathbf{R}f(\mathbf{x}_8)))$, where $\mathbf{MU} = \mathbf{L}$ and $\mathbf{MD} = \mathbf{R}$.

The computation in Relation 4.1 is as follows (see Figure 4.1): Suppose a node $u \in T$ has children v and w of digests $d(v), d(w) \in \mathbb{Z}_q^k$. Applying $g(.)$ *transforms* $d(v), d(w)$ into vectors of $k \log q$ small entries (*admissible* radix-2 representations). Multiplying with $\mathbf{U}$ and $\mathbf{D}$ *prepares* $g(d(v)), g(d(w))$ to be input to the hash function.[2]

[2]The procedure so far is the same with a Merkle tree construction that uses a collision-resistant function such as SHA-2, i.e., recursive computation over the nodes of a tree.

4.2.3 Partial digests

Here we show how to express the digest $d(u)$ (computed in Relation 4.1) for every node $u \in T$ somehow differently, which is crucial for deriving our final results. To simplify some notation, we set $\mathbf{MU} = \mathbf{L}$ and $\mathbf{MD} = \mathbf{R}$ (stand for *left*/*right*)—note that $\mathbf{L}, \mathbf{R} \in \mathbb{Z}_q^{k \times k \log q}$. Let also $\mathsf{range}(u)$ be the range of successive indices corresponding to the leaves of the subtree of T rooted on u. E.g., in Figure 4.1, it is $\mathsf{range}(r_{11}) = \{1, 2, 3, 4\}$. For every node $u \in T$ and for every $i \in \mathsf{range}(u)$ we define the *partial digest* of u with reference to $\mathbf{x}_i$:

Definition 4.8 (Partial digest of a node u) *For a leaf node $u \in T$ storing value $\mathbf{x}_i$, the* partial digest *of u with reference to $\mathbf{x}_i$ is defined as $d(u, \mathbf{x}_i) = \mathbf{x}_i$. Else, for every other node u of T, with* left *child v and* right *child w, and for every $i \in \mathsf{range}(u)$, the* partial digest *$d(u, \mathbf{x}_i)$ of u with reference to $\mathbf{x}_i$ is recursively defined as $d(u, \mathbf{x}_i) = \mathbf{L}f(d(v, \mathbf{x}_i))$, if $\mathbf{x}_i$ belongs to the left subtree of u; Else, $d(u, \mathbf{x}_i) = \mathbf{R}f(d(w, \mathbf{x}_i))$.*

E.g., in Figure 4.1, the partial digests of root r with reference to $\mathbf{x}_2$ and $\mathbf{x}_3$ are $d(r, \mathbf{x}_2) = \mathbf{R}f(\mathbf{R}f(\mathbf{L}f(\mathbf{x}_2)))$ and $d(r, \mathbf{x}_3) = \mathbf{R}f(\mathbf{L}f(\mathbf{R}f(\mathbf{x}_3)))$ respectively ($f(z)$ is z's binary representation). We now give the main result of this section.

Theorem 4.3 *The digest $d(u)$ of node $u \in T$ in Relation 4.1 can be expressed as*

$$d(u) = \sum_{i \in \mathsf{range}(u)} d(u, \mathbf{x}_i)\,,$$

where $d(u, \mathbf{x}_i)$ is the partial digest of node u with reference to $\mathbf{x}_i$.

Proof: We prove the claim by induction on the levels of the tree T. For any internal node u that lies at level $\ell - 1$, there are only two nodes (that store for example values $\mathbf{x}_i$ (left child) and $\mathbf{x}_j$ (right child) and belong to $\mathsf{range}(u)$) in the subtree rooted on u. Therefore

$$\begin{aligned} d(u, \mathbf{x}_i) + d(u, \mathbf{x}_j) &= \mathbf{L}f(\mathbf{x}_i) + \mathbf{R}f(\mathbf{x}_j) = \mathbf{MU}f(\mathbf{x}_i) + \mathbf{MD}f(\mathbf{x}_j) \\ &= \mathbf{M}\left[\mathbf{U}g(\mathbf{x}_i) + \mathbf{D}g(\mathbf{x}_j)\right] = d(u)\,. \end{aligned}$$

This is due to Relation 4.1 and also due to the fact that $g(.)$ can be picked to be $f(.)$, which is an n-admissible radix-2 representation, therefore satisfying the constraint of the inputs of Definition 4.4. Hence the base case holds. Assume the theorem holds for any internal node z that lies at level $0 < t+1 \leq \ell$. Therefore

$$d(z) = \sum_{i \in \mathsf{range}(z)} d(z, \mathbf{x}_i)\,.$$

Let u be an internal node that lies at level t and let $i_1, i_2, \ldots, i_u$ be the indices in $\mathsf{range}(u)$ in *sorted* order. Let v be the left child of u and w be the right child of u. Then, by the definition of the partial digest of the node u (Definition 4.8) we

$$\begin{aligned} d(u) &= \sum_{i \in \mathsf{range}(u)} d(u, \mathbf{x}_i) = \sum_{j=1}^{u/2} \mathbf{L} f(d(v, \mathbf{x}_j)) + \sum_{j=u/2+1}^{u} \mathbf{R} f(d(w, \mathbf{x}_j)) \\ &= \mathbf{MU} \sum_{j=1}^{u/2} f(d(v, \mathbf{x}_j)) + \mathbf{MD} \sum_{j=u/2+1}^{u} f(d(w, \mathbf{x}_j))\,. \end{aligned}$$

By Corollary 4.1 there exist $g(.)$ representations whose entries are at most $u/2 \leq n$ such that

$$d(u) = \mathbf{MU} g\left(\sum_{j=1}^{u/2} d(v, \mathbf{x}_j)\right) + \mathbf{MD} g\left(\sum_{j=u/2+1}^{u} d(w, \mathbf{x}_j)\right)\,.$$

By the inductive step this can be written as

$$d(u) = \mathbf{M}[\mathbf{U} g(d(v)) + \mathbf{D} g(d(w))]\,,$$

where $g(.)$ are radix-2 representations that are n-admissible, since they are the sum of at most $u/2 = n/2$ binary representations. Therefore this satisfies Definition 4.1 and $d(u)$ is indeed the correct digest of any internal node u, as computed by Relation 4.1. This completes the proof. □

Algorithm $\{\mathsf{auth}(D_0), d_0\} \leftarrow \mathsf{setup}(D_0, \mathsf{sk}, \mathsf{pk})$**:** Let D_0 be the initial table, storing values $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n \in \mathbb{Z}_q^k$. The algorithm computes the digests of the nodes: It sets $d(u) = \mathbf{x}_i$ for all leaf nodes u storing value $\mathbf{x}_i$ and $d(u) = \mathbf{M}\left[\mathbf{U} g(d(v)) + \mathbf{D} g(d(w))\right]$ (application of

the hash function in Definition 4.4) for all internal nodes u with left child v and right child w, where $g(d(v))$ and $g(d(w))$, i.e., the radix-2 representations of the children digests, are computed according to the following definition[3]:

Definition 4.9 *The radix-2 representation of $d(u)$ of node $u \in T$ is computed as the sum of $|\mathsf{range}(u)|$ binary representations, i.e.,*

$$g(d(u)) = \sum_{i \in \mathsf{range}(u)} f(d(u, \mathbf{x}_i)) \,,$$

where $d(u, \mathbf{x}_i)$ is the partial digest of node u with reference to $\mathbf{x}_i$.

By combining Theorem 4.3 and Definition 4.9, by Corollary 4.1, we have:

Corollary 4.2 *Let u be an internal node of tree T. The $g(.)$ representation of $d(u)$ defined in Definition 4.9 is an n-admissible radix-2 representation of $d(u)$.*

This concludes the description of $\mathsf{setup}()$. The algorithm outputs $d_0 = d(r)$, where r is the root of T (i.e., the digest of the data structure is the digest of the root of the tree) and also it outputs $\mathsf{auth}(D_0)$ to be a structure that contains: (a) Tree T; (b) $g(d(u))$ for all nodes u of T as computed in Definition 4.9. The complexity of the algorithm is $O(n \log n)$, since the computation of $g(d(u))$ involves a linear number of operations per tree level, and there are $O(\log n)$ levels in total:

Lemma 4.4 *Algorithm $\mathsf{setup}()$ of the authenticated data structure scheme $\mathcal{LBT}$ has $O(n \log n)$ access complexity, outputting an authenticated data structure $\mathsf{auth}(D_0)$ of $O(n)$ group complexity. Moreover it is parallelizable with $O(n)$ access complexity using $O(\log n)$ processors in the CREW model.*

Proof: The algorithm needs to compute the n-admissible radix-2 representations $g(d(u))$ of digests $d(u)$ for every internal node u of the tree T. Note that by Definition 4.9, there

[3]Note here that the *binary* representations $f(d(v)), f(d(w))$ could be used instead; However, in lieu of achieving our efficiency goals, the algorithm uses Definition 4.9.

are $n/2, n/4, n/8, \ldots, 2$ such representations that need to be computed for levels $\ell - 1, \ell - 2, \ell - 3, \ldots, 1$ respectively, each one being the sum of $2, 4, 8 \ldots, n/2$ binary representations respectively, i.e.,

$$g(d(u)) = \sum_{i \in \mathsf{range}(u)} f(d(u, \mathbf{x}_i)) \,.$$

Since computing $f(d(u, \mathbf{x}_i))$ has access complexity $O(1)$ (they are just functions of specific values), it follows that the computation of the $g(.)$ representations for all the internal nodes of the tree requires access complexity

$$\frac{n}{2} \times 2 + \frac{n}{4} \times 4 + \frac{n}{8} \times 8 + \ldots + 2 \times \frac{n}{2} = O(n \log n) \,.$$

Note now in the CREW model, we can use $O(\log n)$ processors, i.e., one processor for each level of the tree. By reading the values $\mathbf{x}_i$ concurrently and writing the values $g(d(u))$ at different memory locations, it follows that each processor will have to do $O(n)$ work in the CREW model. Finally, we note that the output authenticated data structure stores with each internal node u of the tree T the respective n-admissible radix-2 representations $g(d(u))$. Therefore the group complexity of $\mathsf{auth}(D)$ is $O(n)$. This completes the proof. □

We continue by noting that Theorem 4.3 allows us to express $d(r)$ as a sum of well-defined functions of the leaves, namely the *partial digests* of the root r with reference to values in the table. This allows us to achieve our desired complexity bounds:

Corollary 4.3 *Let $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ be the values stored in our table. Then the digest $d(r)$ of the root r of the tree T can be expressed as*

$$d(r) = \sum_{i=1}^{n} d(r, \mathbf{x}_i) \,,$$

where $d(r, \mathbf{x}_i)$ is the partial digest of the root r with reference to $\mathbf{x}_i$.

We observe that computing the partial digest $d(r, \mathbf{x}_i)$ requires *one* query to the authenticated data structure, i.e., a query for value $\mathbf{x}_i$, therefore yielding $O(1)$ access complexity. Matrices $\mathbf{L}$ and $\mathbf{R}$, both used for its computation (Definition 4.8) are *not* part of the authenticated

data structure (they are fixed by $\mathsf{setup}()$ as public information) and accessing them any number of times does not add to the access complexity. We continue with describing the remaining algorithms of our authenticated data structure scheme:

Algorithm $\{d_{h+1}, D_{h+1}, \mathsf{upd}\} \leftarrow \mathsf{update}(u, D_h, d_h, \mathsf{sk}, \mathsf{pk})$**:** Let the update u be *set* $\mathbf{T}[i] = \mathbf{x}'_i$ and let the value of $\mathbf{T}[i]$ before the update be $\mathbf{x}_i$. Then the algorithm sets

$$d_{h+1} = d_h - d(r, \mathbf{x}_i) + d(r, \mathbf{x}'_i)\,,$$

where $d(r, \mathbf{x}_i)$ and $d(r, \mathbf{x}'_i)$ are the *partial digests* of r with reference to $\mathbf{x}_i$ and $\mathbf{x}'_i$, defined in Definition 4.8. Due to Corollary 4.3, d_{h+1} is the correct updated digest. Since the computation of partial digests has constant access complexity, algorithm $\mathsf{update}()$ has $O(1)$ access complexity, since it involves two operations in $\mathbb{Z}_q^k$. The algorithm outputs d_{h+1} as well as the updated table D_{h+1} (note that the algorithm does not need to access the authenticated data structure at all—see Definition 2.3—and does not output anything as upd):

Lemma 4.5 *Algorithm* $\mathsf{update}()$ *of the authenticated data structure scheme* $\mathcal{LBT}$ *has* $O(1)$ *access complexity. Moreover, the update information* upd *output by* $\mathsf{update}()$ *is empty.*

Algorithm $\{D_{h+1}, \mathsf{auth}(D_{h+1}), d_{h+1}\} \leftarrow \mathsf{refresh}(u, D_h, \mathsf{auth}(D_h), d_h, \mathsf{upd}, \mathsf{pk})$**:** This algorithm should update the authenticated data structure $\mathsf{auth}(D_h)$. Let the update u be *set* $\mathbf{T}[i] = \mathbf{x}'_i$ and let the value of $\mathbf{T}[i]$ before the update be $\mathbf{x}_i$. Suppose $v_\ell, v_{\ell-1}, \ldots, v_1$ is the path from the node of index i to the child v_1 of the root of the tree. The algorithm should update the values $g(d(v_j))$ for $j = \ell, \ell-1, \ldots, 1$. This is achieved via Definition 4.9, by setting

$$g(d'(v_j)) = g(d(v_j)) - f(d(v_j, \mathbf{x}_i)) + f(d(v_j, \mathbf{x}'_i)) \text{ for } j = \ell, \ell-1, \ldots, 1\,,$$

namely the invariant of Definition 4.9 must be maintained, and where $d(v_j, \mathbf{x}_i)$, $d(v_j, \mathbf{x}'_i)$ are the partial digests of node v_j with reference to $\mathbf{x}_i$ and $\mathbf{x}'_i$. The algorithm outputs D_{h+1}, the updated $g(d'(.))$ representations as $\mathsf{auth}(D_{h+1})$ and d_{h+1} as in $\mathsf{update}()$, i.e., $d_{h+1} = d_h - d(r, \mathbf{x}_i) + d(r, \mathbf{x}'_i)$.

Lemma 4.6 *Algorithm* refresh() *of the authenticated data structure scheme* $\mathcal{LBT}$ *has* $O(\log n)$ *access complexity. Moreover, it is parallelizable with* $O(1)$ *access complexity using* $O(\log n)$ *processors in the CREW model.*

Proof: For each update from $\mathbf{x}_i$ to $\mathbf{x}'_i$, the algorithm should update the values $g(d(v_j))$ for $j = \ell, \ell-1, \ldots, 1$. This is achieved via Definition 4.9, by setting

$$g(d'(v_j)) = g(d(v_j)) - f(d(v_j, \mathbf{x}_i)) + f(d(v_j, \mathbf{x}'_i))\,, \tag{4.2}$$

(the invariant of Definition 4.9 must be maintained) for $j = \ell, \ell-1, \ldots, 1$ and where $d(v_j, \mathbf{x}_i)$, $d(v_j, \mathbf{x}'_i)$ are the partial digests of node v_j with reference to $\mathbf{x}_i$ and $\mathbf{x}'_i$ respectively. Since $\ell = O(\log n)$ the result follows. Note also that the update Relations 4.2 are independent from one another. Therefore we can take advantage of that, and, in the CREW model, we can use $O(\log n)$ processors, i.e., one processor for each level of the tree. By reading the values $\mathbf{x}_i$ concurrently and writing the values $g(d(u))$ at different memory locations (as required), it follows that each processor will have to do $O(1)$ work in the CREW model. □

Algorithm $\{\Pi(q), \alpha(q)\} \leftarrow \mathsf{query}(q, D_h, \mathsf{auth}(D_h), \mathsf{pk})$**:** Let the query q be *return the value stored at index* i *of table* $\mathbf{T}$. Suppose $v_\ell, v_{\ell-1}, \ldots, v_1$ is the path from the node of index i to the child v_1 of the root of the tree T. The algorithm sets $\alpha(q) = \mathbf{T}[i]$ and sets the proof $\Pi(q)$ to be the array π of $g(.)$ representations such that

$$\pi_i = (g(d(v_i)), g(d(\mathsf{sib}(v_i))))\,, \tag{4.3}$$

for $i = \ell, \ell-1, \ldots, 1$, where $\mathsf{sib}(u)$ denotes the sibling of a node u in tree T.

Lemma 4.7 *Algorithm* query() *of the authenticated data structure scheme* $\mathcal{LBT}$ *has* $O(\log n)$ *access complexity. Moreover, it is parallelizable with* $O(1)$ *access complexity using* $O(\log n)$ *processors in the EREW model. Finally, it outputs a proof* $\Pi(q)$ *of* $O(\log n)$ *group complexity.*

Proof: Since $\ell = O(\log n)$ values have to be collected to construct the proof, the result follows. Moreover, with $O(\log n)$ processors—one processor per node, this algorithm is parallelizable in the EREW model, with $O(1)$ complexity: For $p = 1, \ldots, \ell$, processor p outputs

$\pi_p = (g(d(v_p)), g(d(\mathsf{sib}(v_p))))$, as defined in Relation 4.3. □

Algorithm $\{\mathsf{accept}, \mathsf{reject}\} \leftarrow \mathsf{verify}(q, \alpha, \Pi, d_h, \mathsf{pk})$**:** Let the query q be *return the value at index* i, $y = \alpha$, and $\Pi = \pi$ such that $\pi_j = (\alpha_j, \beta_j)$ $(j = \ell, \ell-1, \ldots, 1)$. For $j = \ell, \ell-1, \ldots, 1$ the algorithm performs the following:

1. If α_j is *not* a $g(.)$ representation of y or α_j, β_j are not n-admissible $g(.)$ representations, output reject;

2. Set $y = \mathbf{M}(\mathbf{U}\alpha_j + \mathbf{D}\beta_j)$ if v_j is v_{j-1}'s left child, or $y = \mathbf{M}(\mathbf{D}\alpha_j + \mathbf{U}\beta_j)$ otherwise.

After the loop terminates, if $y \neq d_h$, reject is output, else, accept is output.

Lemma 4.8 *Algorithm* verify() *of the authenticated data structure scheme* $\mathcal{LBT}$ *has* $O(\log n)$ *access complexity. Moreover, it is parallelizable with* $O(1)$ *access complexity using* $O(\log n)$ *processors in the CRCW model.*

Proof: Since $\ell = O(\log n)$ values have to be processed to perform the verification of the proof, the result follows. The parallel algorithm that a processor $p = \ell, \ell-1, \ldots, 1$ executes is the following (assume that α_0 is defined as a $g(.)$ representation of the digest d_h):

> **If** $p < \ell$ **then** $y = \mathbf{M}(\mathbf{U}\alpha_p + \mathbf{D}\beta_p)$ (or $y = \mathbf{M}(\mathbf{D}\alpha_p + \mathbf{U}\beta_p)$) **else** $y = \alpha$;
>
> **If** α_{p-1} is *not* a $g(.)$ representation of y **or** $\alpha_{p-1}, \alpha_p, \beta_p$ are not n-admissible
>
> **then** output reject **else** output accept;

Note that the algorithm requires *concurrent* write, since all the processors need to write to a location storing the "reject" bit concurrently. Therefore the algorithm is parallelizable in the CRCW model with $O(1)$ access complexity using $O(\log n)$ processors. □

4.2.4 Correctness and security

Lemma 4.9 *The authenticated data structure scheme* $\mathcal{LBT} = \{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$ *is correct according to Definition 2.4.*

Proof: Let $\mathbf{T} = D_0$ be any table of n entries. Fix the security parameter k and output sk and $\mathsf{pk} = (\mathbf{M}, q)$ by calling algorithm $\mathsf{genkey}()$. Then output an authenticated data structure $\mathsf{auth}(D_0)$ and the respective digest d_0, by calling algorithm $\mathsf{setup}()$. Pick a polynomial number of updates—namely, pick a polynomial number of pairs of indices and values to be written on the respective indices—and update $\mathsf{auth}(D_0)$ and d_0 by calling algorithm $\mathsf{refresh}()$. Let D_h be the final table $\mathbf{T}$, $\mathsf{auth}(D_h)$ be the produced authenticated data structure and d_h be the final digest. Let i be an index and let $y = \mathbf{T}[i]$. Output a proof $\Pi(q)$ for index i and answer y by calling $\mathsf{query}()$. $\Pi(q)$ contains pairs $(g(d(v_j)), g(d(\mathsf{sib}(v_j))))$ $(j = \ell, \ell-1, \ldots, 1)$ of n-admissible representations, where $v_\ell, v_{\ell-1}, \ldots, v_1$ are the nodes on the path from index i (i.e., node v_ℓ) to the first child v_1 of the root of the root of the tree T. For the elements of the proof, the following are true:

1. $g(d(v_\ell)) = f(y)$ (definition of a leaf digest);
2. $d(v_{j-1}) = \mathbf{M}(\mathbf{U}g(d(v_j))+\mathbf{D}g(d(\mathsf{sib}(v_j))))$ or $d(v_{j-1}) = \mathbf{M}(\mathbf{D}g(d(v_j))+\mathbf{U}g(d(\mathsf{sib}(v_j))))$—according to left child or right child relation—, for $j = \ell, \ell-1, \ldots, 1$ and where v_0 is the root of the tree (by Relation 4.1);
3. The $g(.)$ representations in $\Pi(q)$ are always n-admissible, i.e., they are maintained to be n-admissible during updates, since $\mathsf{refresh}()$ always updates the $g(.)$ representations so that Definition 4.9 is satisfied, which by Corollary 4.2 gives n-admissible representations.

Based on the above, and the code of $\mathsf{verify}()$, we conclude that $\mathsf{verify}()$ always accepts a proof for index i (of answer $y = \mathbf{T}[i]$) computed by $\mathsf{query}()$. □

Lemma 4.10 *The authenticated data structure scheme* $\mathcal{LBT} = \{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$ *is secure according to Definition 2.5 and assuming the hardness of* GAPSVP_γ *for* $\gamma = O(nk\sqrt{\log n + \log k})$.

Proof: Fix the security parameter k and output sk and $\mathsf{pk} = (\mathbf{M}, q)$ by calling algorithm $\mathsf{genkey}()$. Let Adv be a polynomially-bounded adversary. Adv picks an initial table $\mathbf{T} = D_0$

of n entries and outputs authenticated data structure $\mathsf{auth}(D_0)$, the respective digest d_0, tree T of ℓ levels, by calling algorithm $\mathsf{setup}()$ through oracle access. Then Adv picks a polynomial number of updates—namely, he picks a polynomial number of pairs of indices and values to be written on the respective indices: Let D_h be the final table $\mathbf{T}$, and d_h be the final digest as produced by the adversary through oracle access to algorithm $\mathsf{update}()$. Let $q = i$ be the query index picked by Adv, $y = \mathbf{T}[i]$ be the value stored in this index and $v_l, v_{l-1}, \ldots, v_0$ be the path of T from the node referring to index i to the root of T. The adversary Adv outputs an incorrect answer $\alpha \neq y$ and also a proof $\Pi = (\pi_l, \pi_{l-1}, \ldots, \pi_1)$ $(l = O(\log n))$ where $\pi_j = (\alpha_j, \beta_j)$ (see algorithm $\mathsf{query}()$). We define now the following events, related to the choice of the proof above made by the adversary. Our goal will be to the express the probability that $\mathsf{verify}(i, \alpha, \Pi, d_h, \mathsf{pk})$ accepts while $\alpha \neq y$ as a function of the following events. Note that d_h is the correct digest of the authenticated data structure:

1. $\mathcal{E}_{\ell,0}$: The value α_l picked by Adv is such that α_ℓ is *not* an n-admissible $g(.)$ representation of y;

2. $\mathcal{E}_j$: For $j = \ell - 1, \ldots, 1$, the values α_j and $\alpha_{j+1}, \beta_{j+1} \in \{0, 1, \ldots, n\}^{k \log q}$ picked by Adv are such that α_j *is* an n-admissible $g(.)$ representation of

$$\mathbf{M}(\mathbf{U}\alpha_{j+1} + \mathbf{D}\beta_{j+1}) .$$

   Assume, without loss of generality that a convenient index $i = 0$ is used so that the order of $\mathbf{U}$ and $\mathbf{D}$ is always the same. This event can be partitioned into two mutually exclusive events, i.e., $\mathcal{E}_j = \mathcal{E}_{j,0} \cup \mathcal{E}_{j,1}$ such that

   - $\mathcal{E}_{j,0}$: Value α_j is *not* an n-admissible $g(.)$ representation of the digest of node v_j, as defined in Relation 4.1, i.e., $\alpha_j \neq g(d(v_j))$;
   - $\mathcal{E}_{j,1}$: Value α_j is an n-admissible $g(.)$ representation of the digest of node v_j, as defined in Relation 4.1, i.e., $\alpha_j = g(d(v_j))$.

3. $\mathcal{E}_{0,1}$: The values $\alpha_1 \in \{0,1,\ldots,n\}^{k\log q}$ and $\beta_1 \in \{0,1,\ldots,n\}^{k\log q}$ picked by Adv are such that

$$d_h = \mathbf{M}(\mathbf{U}\alpha_1 + \mathbf{D}\beta_1).$$

The probability that $\mathsf{verify}()$ accepts, while α_ℓ is *not* an n-admissible $g(.)$ representation of y is the probability

$$\begin{aligned}
& \Pr[\mathcal{E}_{\ell,0} \cap \mathcal{E}_{\ell-1} \cap \mathcal{E}_{\ell-2} \cap \ldots \cap \mathcal{E}_{0,1}] \\
= \; & \Pr[\mathcal{E}_{\ell,0} \cap (\mathcal{E}_{\ell-1,0} \cup \mathcal{E}_{\ell-1,1}) \cap (\mathcal{E}_{2,0} \cup \mathcal{E}_{2,1}) \cap \ldots \cap \mathcal{E}_{0,1}] \\
\leq \; & \Pr[\mathcal{E}_{\ell,0}|\mathcal{E}_{\ell-1,1}] + \Pr[\mathcal{E}_{\ell-1,0}|\mathcal{E}_{\ell-2,1}] + \Pr[\mathcal{E}_{\ell-2,0}|\mathcal{E}_{\ell-3,1}] + \ldots + \Pr[\mathcal{E}_{1,0}|\mathcal{E}_{0,1}] \\
= \; & \sum_{j=1}^{\ell} \Pr[\mathcal{E}_{j,0}|\mathcal{E}_{j-1,1}]\,.
\end{aligned}$$

Note that the event $\mathcal{E}_{j,0}|\mathcal{E}_{j-1,1}$ implies the following:

1. $\alpha_j \neq g(d(v_j))$;

2. $\alpha_{j-1} = g(d(v_{j-1}))$, where $d(v_{j-1}) = \mathbf{M}(\mathbf{U}\alpha_j + \mathbf{D}\beta_j)$.

However, from Relation 4.1, it should be that

$$d(v_{j-1}) = \mathbf{M}(\mathbf{U}g(d(v_j)) + \mathbf{D}g(d(\mathsf{sib}(v_j))))\,,$$

where $g(d(v_j))$ and $g(d(\mathsf{sib}(v_j)))$ are the digests of nodes v_j and $\mathsf{sib}(v_j)$ respectively. Therefore (α_j, β_j) is a collision with $(g(d(v_j)), g(d(\mathsf{sib}(v_j))))$, since $\alpha_j \neq g(d(v_j))$. Note now that by Theorem 4.2, which gives $\gamma = O(nk\sqrt{\log n + \log k}) = \mathsf{poly}(k)$ since $q = O(k^{2.5}\delta \log k)$ and $\delta = n$, and Assumption 4.1, $\Pr[\mathcal{E}_{j,0}|\mathcal{E}_{j-1,1}]$ is $\mathsf{neg}(k)$, for all $j = \ell, \ell-1, \ldots, 1$. Therefore the sum

$$\sum_{j=1}^{\ell} \Pr[\mathcal{E}_{j,0}|\mathcal{E}_{j-1,1}]$$

is also $\mathsf{neg}(k)$, since $\ell = O(\log n) = O(\log k)$. This concludes the proof. □

We can now present the main result of this section.

Theorem 4.4 *Let k be the security parameter. Then there exists a publicly-verifiable authenticated data structure scheme $\mathcal{LBT} = \{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$ for a data structure scheme defined for a dynamic table D of n entries such that:*

1. *It is correct according to Definition 2.4 and secure according to Definition 2.5 and assuming the hardness of GAPSVP_γ for $\gamma = O(nk\sqrt{\log n + \log k})$;*

2. *The access complexity of $\mathsf{setup}()$ is $O(n \log n)$ or $O(n)$ using $O(\log n)$ processors in the CREW model, outputting an authenticated data structure $\mathsf{auth}(D)$ of $O(n)$ group complexity;*

3. *The access complexity of $\mathsf{update}()$ is $O(1)$, outputting update information upd of $O(1)$ group complexity;*

4. *The access complexity of $\mathsf{refresh}()$ is $O(\log n)$ or $O(1)$ using $O(\log n)$ processors in the CREW model;*

5. *The access complexity of $\mathsf{query}()$ is $O(\log n)$ or $O(1)$ using $O(\log n)$ processors in the EREW model, outputting a proof $\Pi(q)$ for a query q of $O(\log n)$ group complexity;*

6. *The access complexity of $\mathsf{verify}()$ is $O(\log n)$ or $O(1)$ using $O(\log n)$ processors in the CRCW model.*

Proof: This result follows directly from Lemmata 4.4, 4.5, 4.6, 4.7, 4.8, 4.9 and 4.10. Note that the presented scheme is publicly verifiable since $\mathsf{verify}()$ does not take the secret key as an input. □

4.2.5 A note on repeated linearity

We note here that the fact that the used collision-resistant hash function is additive (homomorphic), i.e., it is

$$\mathbf{Mx} + \mathbf{My} = \mathbf{M}(\mathbf{x} + \mathbf{y}),$$

is not enough for deriving our results. This is the reason that other *homomorphic* collision-resistant hash functions (e.g., exponentiation with secret factorization) could not be employed instead. The crucial property we can exhibit here, which is what we call *repeated linearity*, is a means of "feeding" the output of the function again as an input, so that certain homomorphic properties are still satisfied—and in specific the properties of Corollary 4.1. Therefore, it might be the case that other functions could be also used instead, would they satisfy such a property.

4.3 Authenticated bloom filters

In this paragraph we show how we can use the lattice-based hash function to verify the Bloom filter functionality, a space-efficient dictionary, originally introduced by Bloom [14]. The Bloom filter consists of an array (table) $A[0 \dots n-1]$ storing n bits. All the bits are initially set to 0. Suppose one needs to store a set S of r elements. Then K hash functions $h_i(.)$ with range $\{0, \dots, n-1\}$ are used (these are not lattice-based hash functions) and for each element $s \in S$ we set the bits $A[h_i(s)]$ to 1, for $i = 1, \dots, K$. In this way, false positives can occur, i.e., an element that is not present might be represented in A. The probability of a false positive can be proved to be $(1-p)^K$, where $p = e^{-Kr/n}$, which is minimized for $K = \ln 2(n/r)$ [14].

The Bloom filter above supports only insertions though. A deletion (i.e., setting some bits to 0) can cause the undesired deletion of many elements. To deal with this problem, *counting Bloom filters* were introduced by Fan et al. [38]. In this solution, by keeping a counter for each index of A (instead of just 0 or 1), we can tolerate deletions by incrementing the counter during insertions and decrementing the counter during deletions. However, the problem of *overflow* exists. As observed by Broder and Mitzenmacher [21], the overflow (at least one counter goes over some value C) occurs with probability $n(e \ln 2/C)^C$, for a certain set of r elements. Setting $C = O(1)$ (e.g., $C = 16$) is suitable for most of the applications [21].

By the above description, it is clear that we can use our lattice-based construction to authenticate the Bloom filter functionality: Each update in the Bloom filter corresponds to K updates in table T and querying one element in the Bloom filter corresponds to K queries to table T. Note that constant update complexity in this application is very important given that a Bloom filter is an *update-intensive* data structure (i.e., an insertion or deletion of an element involves K operations):

Theorem 4.5 *Let k be the security parameter. Then there exists a publicly-verifiable authenticated data structure scheme $\mathcal{ABF} = \{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$ for a data structure scheme defined for a Bloom filter D of n entries, storing r elements and using K hash functions such that:*

1. *It is correct according to Definition 2.4 and secure according to Definition 2.5 and assuming the hardness of GAPSVP_γ for $\gamma = O(nk\sqrt{\log n + \log k})$;*

2. *The access complexity of $\mathsf{setup}()$ is $O(n \log n)$ or $O(n)$ using $O(\log n)$ processors in the CREW model, outputting an authenticated data structure $\mathsf{auth}(D)$ of $O(n)$ group complexity;*

3. *The access complexity of $\mathsf{update}()$ is $O(K)$, outputting update information upd of $O(1)$ group complexity;*

4. *The access complexity of $\mathsf{refresh}()$ is $O(K \log n)$ or $O(K)$ using $O(\log n)$ processors in the CREW model;*

5. *The access complexity of $\mathsf{query}()$ is $O(K \log n)$ or $O(K)$ using $O(\log n)$ processors in the EREW model, outputting a proof $\Pi(q)$ for a query q of $O(K \log n)$ group complexity;*

6. *The access complexity of $\mathsf{verify}()$ is $O(K \log n)$ or $O(K)$ using $O(\log n)$ processors in the CRCW model.*

Proof: The construction for an authenticated Bloom filter is the same with Theorem 4.4. The extra K multiplicative factor in the complexities is due to the fact that one operation in the authenticated Bloom filter (insertion, deletion and query of an element) requires $O(K)$ operations on an authenticated table. This follows by the construction and the definition of the Bloom filter data structure. □

4.4 Parallel online memory checking

In this section, we establish our results concerning *parallel* online memory checking. The online memory checking model [15] can be (informally) described as follows: Suppose $\mathcal{M}$ is an *unreliable* (malicious) memory of n cells. A user $\mathcal{U}$ wants to read (through operation $\mathsf{read}(i)$) or write (through operation $\mathsf{write}(i, x)$, where x is the new content) a cell $i \in \{1, 2, \ldots, n\}$. However, his requests go through a checker $\mathcal{C}$. The checker is supposed to read cells from the unreliable memory $\mathcal{C}$ and also some reliable (and possibly secret) information s of *sublinear* size and output either the correct answer (i.e., the latest content of cell i) or **BUGGY**, if the content of cell i is corrupted. The probability of returning the corrupted content of a cell as correct should be negligible. The checker is called *non-adaptive*, if, given an index i, the set and the order of the cells accessed in order to output the answer is deterministic. In this paper we are considering such checkers. In the following, we give the formal definition:

Definition 1 (Online memory checking [35]) *Let $\mathcal{M}$ be an n-cell unreliable memory. An online non-adaptive memory checker $\mathcal{C} = (\Sigma, n, q, s)$ over an alphabet Σ with reliable (and possibly secret) memory s is a probabilistic Turing machine with five tapes:*

- *A read-only input tape for receiving read/write requests from the user $\mathcal{U}$ to the unreliable memory $\mathcal{M}$ of n cells, indexed by $1, 2, \ldots, n$;*
- *A write-only output tape for sending responses back to the user;*
- *A read-write work tape, i.e., the (secret) reliable memory s;*

- *A write-only tape for sending read/write requests to the memory $\mathcal{M}$;*
- *A read only input tape for receiving $\mathcal{M}$'s responses.*

A checker is presented with $\mathsf{write}(i, x)$ *and* $\mathsf{read}(i)$ *requests made by* $\mathcal{U}$ *to* $\mathcal{M}$*, where* $i \in \{1, 2, \ldots, n\}$*. After each* read *request* $\mathcal{C}$ *returns an answer or outputs that* $\mathcal{M}$*'s operation is* **BUGGY**. $\mathcal{C}$*'s operation should be both correct and secure:*

1. Correctness*: For any polynomially-large sequence of user requests, as long as* $\mathcal{M}$ *answers all of* $\mathcal{C}$*'s* read *requests correctly,* $\mathcal{C}$ *also answers all of the user's* read *requests correctly;*
2. Security*: For any any polynomially-large sequence of user requests, for any (even incorrect or malicious) answers returned by* $\mathcal{M}$*, the probability that* $\mathcal{C}$ *answers a user request incorrectly is* $\mathsf{neg}(k)$*, where* k *is the security parameter.* $\mathcal{C}$ *may either recover the correct answer independently or answer that* $\mathcal{M}$ *is* **BUGGY***, but it may not answer a request incorrectly (beyond negligible probability).*

In online memory checking settings, the complexity measure we are interested in minimizing is the *query* complexity, which is defined as the sum of the *number of requests* that the checker makes to the *unreliable* memory $\mathcal{M}$ during a $\mathsf{read}(i)$ operation plus the *number of requests* that the checker makes to the *unreliable* memory $\mathcal{M}$ during a $\mathsf{write}(i, x)$ operation [82]. So far in the literature, and in the computational model, checkers with $O(\log n)$ [15] or $O(\log_d n)$ [35] query complexity have appeared. Specifically for these checkers, we can distinguish two cases:

1. In the secret key setting, i.e., when there is requirement for *both* reliable and secret small memory s, these checkers have been shown to be parallelizable, e.g., see the work of Hall and Julta [54], as well as the construction based on PRFs [15]—although this has not been reported in the literature[4];

[4]The construction based on PRFs appearing [15] is easily parallelizable since the PRF tag computed on each node of the tree is not a function of the PRF tags of its children.

2. In the non-secret key setting, i.e., when there is requirement for *only* reliable memory (e.g., the construction using UOWHFs from [15] and Merkle tree constructions), these checkers have appeared to be *inherently* sequential.

However, in this section we establish the first parallel online memory checker in the non-secret key setting:

Theorem 4.6 *In the non-secret key setting and in the CREW model of parallel computation, there is a non-adaptive online memory checker for an unreliable memory of n cells with $O(1)$ query complexity, using $O(\log n)$ checkers and $O(1)$ reliable memory.*

Proof: Let $\mathcal{LBT} = \{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$ be the authenticated data structure scheme derived in Theorem 4.4. We show how to construct a parallel online memory checker by using this scheme, in the non-secret key setting. Let $\mathcal{M}$ be the unreliable memory accessed through indices $1, 2, \ldots, n$. Assume we can use u checkers $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_u$, where $u = O(\log n)$. The user $\mathcal{U}$ sends his requests to all the checkers simultaneously and all the checkers have access to the *unreliable* memory $\mathcal{M}$ and to some *reliable* memory s. We work in the CREW model—i.e., all the checkers can read simultaneously the same value but writing at the same location simultaneously is not feasible. Let $\{\mathsf{sk}, \mathsf{pk}\} \leftarrow \mathsf{genkey}()$, where $\mathsf{sk} = \emptyset$. The checkers run the algorithm $\{\mathsf{auth}(\mathcal{M}), d_0\} \leftarrow \mathsf{setup}(\mathcal{M}, \mathsf{pk})$ (since $\mathsf{sk} = \emptyset$ we do not use the secret key as input from now on) in parallel, requiring $O(n)$ access complexity in the CREW model (Theorem 4.4). The authenticated structure $\mathsf{auth}(\mathcal{M})$ is stored in the unreliable memory (all its parts can be uniquely referenced) and d_0 is stored in the small reliable memory, i.e., $s = d_0$. We have two cases:

1. User $\mathcal{U}$ sends the request $\mathsf{read}(i)$ to all checkers $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_u$. The checkers run the algorithm $\mathsf{query}(i, \mathcal{M}, \mathsf{auth}(\mathcal{M}), \mathsf{pk})$ in parallel and output the answer $\mathcal{M}[i]$ and the proof $\Pi(i)$. This requires $O(1)$ requests to the unreliable memory per checker in the EREW model (Theorem 4.4). Then the algorithm $\mathsf{verify}(i, \mathcal{M}[i], \Pi(i), s, \mathsf{pk})$ is run by the checkers (note that running $\mathsf{query}()$ and $\mathsf{verify}()$ can be combined in one algorithm).

The algorithm writes either $\mathcal{M}[i]$ (in this case $\mathsf{verify}()$ accepts) or **BUGGY** (in this case $\mathsf{verify}()$ rejects) in a location of the reliable memory. User $\mathcal{U}$ reads that location and gets the result. We note here that the fact that $\mathsf{verify}()$ is parallelizable in the CRCW model does not affect our complexity results since the *write* part of the algorithm is done on the *reliable* memory—however, requests to the reliable memory are not taken into account in query complexity (only requests to the unreliable memory). Therefore the query complexity of the parallel checker due to *read* operations is $O(1)$ in EREW model;

2. User $\mathcal{U}$ sends the request $\mathsf{write}(i, x)$ to all checkers $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_u$. First the current content of cell i is verified through a $\mathsf{read}(i)$ operation. If this verification succeeds the checkers run algorithm

$$\{\mathcal{M}', \mathsf{auth}(\mathcal{M}'), s'\} \leftarrow \mathsf{refresh}(\mathsf{write}(i, x), \mathcal{M}, \mathsf{auth}(\mathcal{M}), s, \mathsf{pk})$$

in parallel. Note that this algorithm has $O(1)$ access complexity using $O(\log n)$ processors in the CREW model, by Theorem 4.4. We need *concurrent read* because all the checkers should be able to read the same value of the old (verified) content of cell i.

Finally, we note that the correctness and the security of the checker comes as a direct result of the correctness and the security of the authenticated data structure scheme $\mathcal{LBT}$. Also, since our lattice-based construction does not use any secret key, it follows that the construction we have described is in the non-secret key setting. This completes the proof. □

4.5 Protocols

Three-party protocol. By using Theorem 2.1 we can easily derive the following corollary that describes the use of the authenticated data structure scheme $\mathcal{LBT}$ of Theorem 4.4 in the three-party model:

Corollary 4.4 *Let k be the security parameter and assume the hardness of* GAPSVP_γ *for* $\gamma = \mathsf{poly}(k)$. *Then there exists a three-party authenticated data structures protocol (see Protocol 2.1) for verifying queries q on a dynamic table of n entries such that:*

1. *The setup at the source has $O(n \log n)$ access complexity or $O(n)$ access complexity using $O(\log n)$ processors in the CREW model;*
2. *The update at the source has $O(1)$ access complexity;*
3. *The space needed at the source has $O(n)$ group complexity;*
4. *The communication between the source and the server has $O(1)$ group complexity;*
5. *The update at the server has $O(\log n)$ access complexity or $O(1)$ access complexity using $O(\log n)$ processors in the CREW model;*
6. *The query at the server has $O(\log n)$ access complexity or $O(1)$ access complexity using $O(\log n)$ processors in the EREW model;*
7. *The space needed at the server has $O(n)$ group complexity;*
8. *The communication between the server and the client has $O(\log n)$ group complexity;*
9. *The verification at the client has $O(\log n)$ access complexity or $O(1)$ access complexity using $O(\log n)$ processors in the CRCW model;*
10. *For a query q sent by the client to the server at any time (even after updates), let α be an answer and let π be a proof returned by the server. With probability $\Omega(1 - \mathsf{neg}(k))$, the client accepts the answer α if and only if α is correct.*

Two-party protocol. As a corollary of Example 2.1 (the Merkle tree techniques apply in the lattice-based authenticated table as are), we can state the following for the authenticated data structure scheme $\mathcal{LBT}$:

Corollary 4.5 *Assumption 2.1 is true for the authenticated data structure scheme $\mathcal{LBT}$. Moreover, for every update u, $|Q_u|$ has $O(1)$ complexity.*

By Theorems 2.2 and 4.4 and Corollary 4.5, we can now state the final result for the two-party model:

Corollary 4.6 *Let k be the security parameter and assume the hardness of GAPSVP_γ for $\gamma = \mathsf{poly}(k)$. Then there exists a two-party authenticated data structures protocol (see Protocol 2.2) for verifying queries q on a dynamic table of n entries such that:*

1. *The protocol is non-interactive;*
2. *The setup at the client has $O(n \log n)$ access complexity or $O(n)$ access complexity using $O(\log n)$ processors in the CREW model;*
3. *The update at the client has $O(\log n)$ access complexity or $O(1)$ access complexity using $O(\log n)$ processors in the CRCW model;*
4. *The verification at the client has $O(\log n)$ access complexity or $O(1)$ access complexity using $O(\log n)$ processors in the CRCW model;*
5. *The space needed at the client has $O(1)$ group complexity;*
6. *The communication between the client and the server has $O(\log n)$ group complexity;*
7. *The update at the server has $O(\log n)$ access complexity or $O(1)$ access complexity using $O(\log n)$ processors in the CREW model;*
8. *The query at the server has $O(\log n)$ access complexity or $O(1)$ access complexity using $O(\log n)$ processors in the EREW model;*
9. *The space needed at the server has $O(n)$ group complexity;*

10. *For a query q sent by the client to the server at any time (even after updates), let α be an answer and let π be a proof returned by the server. With probability $\Omega(1 - \mathsf{neg}(k))$, the client accepts the answer α if and only if α is correct.*

Finally, we note that similar protocols can be derived for the authenticated data structure scheme $\mathcal{ABF}$ (Theorem 4.5), referring to Bloom filters.

Chapter 5

Authenticated sets operations with bilinear maps

In the previous chapters of this thesis, we mainly studied the verification of fundamental data structure queries, such as hash table queries (Chapter 3) and index queries on tables (Chapter 4). The verification of these queries in the authenticated data structures setting allows us to secure *outsourced storage* efficiently, i.e., to ensure that data has not been tampered with by the untrusted party that stores it. In this chapter, we follow a different direction where we are interested in verifying *outsourced computation*. Namely, how can one verify the outcome of a computation that has been performed by an untrusted entity? Of course, the main challenge in this paradigm is that the verification procedure should not involve executing the computation from scratch: This would defeat the purpose of employing a powerful (but untrusted) machine in the cloud to perform the computation for us.

Motivated mainly by computations performed by search engines (e.g., keyword searches using an inverted index) as well as by database applications, in this chapter, we examine a very fundamental class of computations: We study the verification of outsourced operations on general *sets*, where a dynamic collection of m sets $S_1, S_2, \ldots, S_m$ is remotely stored at an untrusted server and we wish to *publicly* verify primitive queries on these sets, such as *intersection*, *union* and *set difference*. For example, for the query requesting the intersection

of t sets specified by indices $i_1, i_2, \ldots, i_t$ between 1 and m, we wish to design techniques that allow any client to cryptographically check the correctness of the returned intersection $S_{i_1} \cap S_{i_2} \cap \ldots \cap S_{i_t}$. In addition, we wish the verification of any set operation be *operation-sensitive*, meaning that the required complexity depends only on the (description and outcome of the) operation, and not on the sizes of the involved sets. For example, if $|S_{i_1} \cap S_{i_2} \cap \ldots \cap S_{i_t}| = \delta$, then we would like the verification cost to be proportional to $t + \delta$. This achieves optimality, as the query and the answer require $O(t + \delta)$ complexity.

Relation to outsourced verifiable computation. Recent works on *outsourced verifiable computation* by Gennaro et al. [41], Chung et al. [28] and Applebaum et al. [5] achieve operation-sensitive verification of general functionalities. Although such approaches completely cover set operations as a special case, clearly meeting our goal with respect to optimal verifiability, they are inherently inadequate to meet our other goals with respect to *public verifiability* and *dynamic updates*, both important properties in the context of data querying. Indeed, the works on outsourced verifiable computation [5, 28, 41] are primarily designed to provide secrecy of the outsourced computations, and as such, the client makes use of some secret information to outsource the computation as a circuit and in an encrypted form. This secret information is also used in the verifying computation, therefore effectively supporting only one verifier; instead, we seek for schemes that allow any client to query the sets collection and verify the returned results. Finally, in the outsourced verifiable computation framework [5, 28, 41], the description of the circuit is fixed at the initialization of the scheme, therefore effectively supporting no updates (or, very expensive updates as shown in Table 5.1 for [41]) in the outsourced data; instead, we seek for schemes supporting efficient updates. We accordingly study our problem in the model of *authenticated data structures*, which provides mechanisms for supporting public verifiability and queries on dynamic data.

Achieving operation-sensitive verification. In this chapter, we design a new authenticated data structure scheme (denoted with $\mathcal{ASC}$ in Table 5.1) for the verification of set

operations in an operation-sensitive manner, that is, with proof and verification complexity depending only on the description and outcome of the operation and not on the size of the sets involved. Conceptually, this property is similar to the property of *super-efficient verification* that has been studied in certifying algorithms [63] and certification data structures [52, 107] (as well as in the context of *outsourced verifiable computation* [5, 28, 41]), where an answer can be verified in complexity asymptotically less than the complexity required to produce it. Whether the above optimality property is achievable for set operations (with linear storage) was posed as an open problem by Devanbu et al. [33]. We close this problem in the affirmative.

All existing schemes for verifying outsourced set operations fall into the following two rather straightforward and highly inefficient solutions (for a detailed comparison see Table 5.1): Either short proofs for the answer of every possible set operation query are precomputed allowing for highly imbalanced schemes (exponential storage is required in order to achieve optimal verification, e.g., see the work by Pang and Tan [89]) or integrity proofs for all the elements of the sets participating in the query are given to the client who locally verifies the set operation (in this case verification complexity can be linear in the problem size, e.g., see the work by Devanbu et al. [33]).

Table 5.1: Asymptotic *access* and *group* complexities of various authenticated data structure schemes defined by algorithms {genkey, setup, update, refresh, query, verify}, for a sets collection data structure of m sets: The sum of sizes of all the sets is M and $0 < \epsilon < 1$ is a constant. FHE stands for *fully-homomorphic encryption*, the security of which is based on lattice assumptions, such as the *bounded distance decoding* and the SplitKey *distinguishing* problems—see [43]. We note that the scheme based on FHE is not publicly-verifiable. It however provides privacy on top of integrity of computations. We show complexities for an intersection query on $t = O(1)$ sets, outputting an intersection δ elements. All sizes of the intersected and updated sets are $\Theta(n)$.

	[33, 112]	[79]	[89]	[41]	$\mathcal{ASC}$
setup()	$m + M$	$m + M$	$m^t + M$	$m + M$	$m + M$
update()	$\log n + \log m$	$m + M$	m^t	$m + M$	1
refresh()	$\log n + \log m$	$m + M$	m^t	$m + M$	1
query()	$n + \log m$	n	1	$m + M$	$n \log^2 n \log \log n + m^\epsilon \log m$
verify()	$n + \log m$	n	δ	δ	δ
proof $\Pi(q)$	$n + \log m$	n	δ	δ	δ
info. upd	1	n	m^t	$m + M$	1
publicly verifiable	yes	yes	yes	no	yes
assumption	Generic CR	Strong RSA	D. Log	FHE	B. q-DH

Intuition of our construction. We achieve optimal verification complexity by departing from the above approaches as follows. We first reduce the problem of verifying set operations to the problem of *verifying the validity of some more primitive relations on sets*, namely *subset containment* and *set disjointness*. Then for each such primitive relation we employ a corresponding cryptographic primitive to optimally verify its validity. In particular, we extend the bilinear-map accumulator to optimally verify *subset containment*, inspired by [90]. We then employ the extended Euclidean algorithm over polynomials in combination with subset containment proofs to provide a novel optimal verification test for *set disjointness*. The intuition behind our technique is that disjoint sets can be represented by polynomials mutually indivisible, therefore there exist other polynomials so that the sum of their pairwise products equals to one—this is the test to be used in the proof. However, transmitting (and processing) these polynomials is bandwidth (and time)-prohibitive and does not lead to operation-sensitivity. Taking advantage of bilinearity properties, we can compress their coefficients in the exponent and still use them in a meaningful way, i.e., compute an internal product. This is why although using a conceptually simpler RSA accumulator [11] would lead to a mathematically sound solution, a bilinear-map accumulator [83] is essential for achieving the desired complexity goal.

Related work for securing sets operations. Despite the fact that privacy-related problems for set operations have been extensively studied in the cryptographic literature (e.g., see the work by Boneh and Waters [20] and the work by Freedman et al. [39]), existing work on the integrity dimension of set operations appears mostly in the database literature. Devanbu et al. [33] identify the importance of coming up with an operation-sensitive scheme. In the work by Morselli et al. [79], possibly the closest in context work to ours, set intersection, union and difference are authenticated with linear verification and proof costs. Same linear asymptotic bounds are achieved by Yang et al. [112]. Pang and Tan [89] take a different

approach: In order to achieve operation-sensitivity, expensive pre-processing and exponential space are required (i.e., answers to all possible queries are signed). Finally, related to our work are non-membership proofs, both for the RSA [68] and the bilinear-map [8, 32] accumulators. We note here that the first part of the solution presented in this chapter uses a modification of the authenticated data structure scheme $\mathcal{BHT}$ presented in Chapter 3.

5.1 Preliminaries

The data structure for which we design an authenticated data structure scheme for is called *sets collection* and is a generalization of the *inverted index* [9]. We describe it in detail in the following paragraph.

5.1.1 Sets collection data structure scheme

The sets collection data structure consists of a collection of m sets, denoted with $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$, each containing elements from a universe $\mathcal{U}$. Without loss of generality we assume that our universe $\mathcal{U}$ is the set of nonnegative integers in the interval $[m+1, p-1]$, where p is k-bit prime, m is the number of the sets in our collection that has bit size $O(\log k)$, and where k is the security parameter[1]. Every set S_i is maintained to be sorted and does not contain duplicate elements; however an element x can appear in more than one set. The space usage of the sets collection is $O(m+M)$, where M is the sum of the sizes of the sets. Let now $\mathcal{I}_t$ be a collection of t indices, all between 1 and m. The data structure scheme $\{\mathbf{query}(), \mathbf{update}(), \mathbf{check}()\}$ (Definition 2.2) for a sets collection data structure $\mathbf{T}(\mathcal{S})$ supports various set operations over a collection $\mathcal{S}$ of dynamic sets and is defined as follows:

1. $\mathsf{answer} \leftarrow \mathbf{query}(\mathcal{I}_t, \mathbf{T}(\mathcal{S}), \mathsf{op})$: Depending on the input parameter op, a query on the sets collection data structure is one of the following standard set operations:

[1] As we are going to see later in this chapter, we could have easily set our universe to be $\mathbb{Z}_p$ by using CRHFs, but we choose not to do so in sake of a cleaner presentation. However, even with this constraint, our universe contains $O(2^k - \mathsf{poly}(k)) = O(2^k)$ elements, since m is polynomially large.

- *Intersection*: Given indices $\mathcal{I}_t = \{i_1, i_2, \ldots, i_t\}$, return set $\mathsf{I} = S_{i_1} \cap S_{i_2} \cap \ldots \cap S_{i_t}$ as answer;
- *Union*: Given indices $\mathcal{I}_t = \{i_1, i_2, \ldots, i_t\}$, return set $\mathsf{U} = S_{i_1} \cup S_{i_2} \cup \ldots \cup S_{i_t}$ as answer;
- *Subset*: Given indices $\mathcal{I}_t = \{i, j\}$, return true as answer if $S_i \subseteq S_j$ and false otherwise;
- *Set difference*: Given indices $\mathcal{I}_t = \{i, j\}$, return the set $\mathsf{D} = S_i - S_j$ as answer.

2. $\mathbf{T}(\mathcal{S}') \leftarrow$ **update**$(x, i, \mathbf{T}(\mathcal{S}))$: Given an element $x \in \mathcal{U}$ and $1 \leq i \leq m$ such that $x \notin S_i$, *insert* element x into S_i and output $\mathbf{T}(\mathcal{S}')$; Given an element $x \in \mathcal{U}$ such that $x \in S_i$, *delete* element x from S_i and output $\mathbf{T}(\mathcal{S}')$.

3. $\{\mathsf{accept}, \mathsf{reject}\} \leftarrow$ **check**$(\mathsf{answer}, \mathsf{op}, \mathbf{T}(\mathcal{S}))$: Output true if answer is the correct answer to the query on $\mathbf{T}(\mathcal{S})$ defined by op.

Complexity. Let N be the sum of the sizes of the sets participating in the queries defined by algorithm **query**(). By using a generalized merge, all these queries can be answered with $O(N)$ complexity. Moreover, due to the requirement of keeping the sets sorted, all the updates require $O(\log N)$ complexity. Also, for the remainder of the chapter, we denote with δ the size of the answer to a query operation, i.e., δ is equal to the size of I, U, or D. For a subset query, δ is $O(1)$ (true/false).

Sets collection as a hash table. We observe here that the sets collection data structure $\mathbf{T}(\mathcal{S})$ for the sets collection $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ can be viewed as a special *hash table*: Every set S_i refers to a bucket L_i of the hash table data structure scheme in Chapter 3. This construction does not have expected $O(1)$ size for the buckets, since the sets can have arbitrary size. However, viewing the sets collection as a hash table—that uses a different function for distributing elements in the buckets—will allow as to employ scheme $\mathcal{BHT}$ from Chapter 3 as a black box for verifying set operations queries.

5.1.2 Subset witnesses

Our construction uses bilinear maps and the bilinear-map accumulator, which were introduced in Section 3.1.3. We urge the reader to review the bilinear-map accumulator section (Section 3.1.3) before continuing. We begin with introducing an extra property that is going to be used in this context, the property of *subset witnesses*, which also appeared simultaneously (without a proof though) in the recent work of Canard and Gouget [25]. Assume that the bilinear-map parameters are in place, as described in Section 3.1.3. The proof for *subset containment* of a set $\mathcal{S} \subseteq \mathcal{X}$—for $|\mathcal{S}| = 1$, this is a proof of membership—is the witness $(\mathsf{W}_{\mathcal{S},\mathcal{X}}, \mathcal{S})$ where

$$\mathsf{W}_{\mathcal{S},\mathcal{X}} = g^{\prod_{x \in \mathcal{X} - \mathcal{S}}(x+s)} . \tag{5.1}$$

A verifier can test subset containment for $\mathcal{S}$ by checking the relation $e(\mathsf{W}_{\mathcal{S},\mathcal{X}}, g^{\prod_{x\in\mathcal{S}}(x+s)}) \stackrel{?}{=} e\left(\mathsf{acc}(\mathcal{X}), g\right)$. We continue with the proof of security, which is a generalization of the membership proof presented by Nguyen [83]:

Lemma 5.1 (Proving subsets) *Let k be the security parameter and let $(p, \mathbb{G}, \mathcal{G}, e, g)$ be a uniformly randomly generated tuple of bilinear pairings parameters. Given the elements $g, g^s, \ldots, g^{s^q} \in \mathbb{G}$ for some s chosen at random from $\mathbb{Z}_p^*$ and a set of elements $\mathcal{X}$ in $\mathbb{Z}_p$ $(q \geq |\mathcal{X}|)$, suppose there is a polynomial-time algorithm that finds $\mathcal{S}$ and W such that $\mathcal{S} \nsubseteq \mathcal{X}$ and $e(\mathsf{W}, g^{\prod_{x\in\mathcal{S}}(x+s)}) = e(\mathsf{acc}(\mathcal{X}), g)$. Then there is a polynomial-time algorithm for breaking the bilinear q-strong Diffie-Hellman assumption.*

Proof: Suppose there is a polynomial-time algorithm that computes such a set $\mathcal{S} = \{y_1, y_2, \ldots, y_\ell\}$. Let $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ and $y_j \notin \mathcal{X}$ for some $1 \leq j \leq \ell$. That means that

$$e(\mathsf{W}, g)^{\prod_{y\in\mathcal{S}}(y+s)} = e(g, g)^{(x_1+s)(x_2+s)\ldots(x_n+s)} .$$

Note that (y_j+s) does not divide $(x_1+s)(x_2+s)\ldots(x_n+s)$. Therefore there exist polynomial $Q(s)$ of degree $n-1$ and constant λ, such that $(x_1+s)(x_2+s)\ldots(x_n+s) = Q(s)(y_j+s)+\lambda$.

Thus we have

$$
\begin{aligned}
&e(\mathsf{W},g)^{(y_j+s)\prod_{1\leq i\neq j\leq \ell}(y_i+s)} = e(g,g)^{Q(s)(y_j+s)+\lambda} \Rightarrow \\
&e(\mathsf{W},g)^{\prod_{1\leq i\neq j\leq \ell}(y_i+s)} = e(g,g)^{Q(s)+\frac{\lambda}{(y_j+s)}} \Rightarrow \\
&e(\mathsf{W},g)^{\prod_{1\leq i\neq j\leq \ell}(y_i+s)} = e(g,g)^{Q(s)} e(g,g)^{\frac{\lambda}{(y_j+s)}} \Rightarrow \\
&e(g,g)^{\frac{1}{y_j+s}} = \left[e\left(\mathsf{W}, g^{\prod_{1\leq i\neq j\leq \ell}(y_i+s)}\right) e\left(g,g\right)^{-Q(s)}\right]^{\lambda^{-1}} .
\end{aligned}
$$

This means that the algorithm can be used to break the bilinear q-strong Diffie-Hellman assumption. □

5.2 Construction and algorithms

In the following, we recall that m denotes the number of the sets of our sets collection data structure and M denotes the sum of the sizes of the sets in our collections, i.e.,

$$M = \sum_{i=1}^{m} |S_i| .$$

We now describe $\mathcal{ASC} = \{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$, our authenticated data structure scheme for a sets collection data structure $S_1, S_2, \ldots, S_m$. To do that, we are going to employ an *extended* version of the authenticated data structure scheme $\mathcal{BHT} = \{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$ described in Chapter 3: Instead of employing $\mathcal{BHT}$ on top of some m buckets created by using a two-universal hash function H on an elements collection $\mathcal{X}$, we are going to employ it on top of the sets $S_1 \cup \{1\}, S_2 \cup \{2\}, \ldots, S_m \cup \{m\}$. By the constraint of our universe $\mathcal{U}$, note that $i \notin S_i$.

Algorithm $\{\mathsf{sk}, \mathsf{pk}\} \leftarrow \mathsf{genkey}(1^k)$**:** The algorithm calls $\{\mathsf{sk}, \mathsf{pk}\} \leftarrow \mathcal{BHT}.\mathsf{genkey}()$. It outputs $\mathcal{BHT}.\mathsf{sk}$ as sk and $\mathcal{BHT}.\mathsf{pk}$ as pk. The access complexity is of this algorithm is $O(1)$.

Algorithm $\{\mathsf{auth}(D_0), d_0\} \leftarrow \mathsf{setup}(D_0, \mathsf{sk}, \mathsf{pk})$**:** The authenticated data structure $\mathsf{auth}(D_0)$ is built as follows: First of all, the accumulation values of sets S_i

$$\mathsf{acc}(S_i) = g^{\prod_{x \in S_i}(s+x)} \text{ for all } i = 1, \ldots, m\,, \tag{5.2}$$

are computed (see Section 3.1). Then the algorithm calls

$$\{\mathsf{auth}(D_0), d_0\} \leftarrow \mathcal{BHT}.\mathsf{setup}(D_0, \mathsf{sk}, \mathsf{pk})\,,$$

without precomputed witnesses, and where D_0 is the collection of m sets

$$S_1 \cup \{1\}, S_2 \cup \{2\}, \ldots, S_m \cup \{m\}\,.$$

Namely, the "bucket" L_i in the scheme $\mathcal{BHT}$ is defined as the set $S_i \cup \{i\}$ in this construction. The algorithm outputs both the authenticated data structure $\mathcal{BHT}.\mathsf{auth}(D_0)$ and the accumulation values $\mathsf{acc}(S_i)$ for all $i = 1, \ldots, m$ as $\mathsf{auth}(D_0)$. Also it sets d_0 to be $\mathcal{BHT}.d_0$.

Lemma 5.2 *Algorithm* $\mathsf{setup}()$ *of the authenticated data structure scheme* $\mathcal{ASC}$ *has* $O(m + M)$ *access complexity. Moreover, the authenticated data structure* $\mathsf{auth}(D_0)$ *output by* $\mathsf{setup}()$ *has* $O(m + M)$ *group complexity.*

Proof: When the scheme $\mathcal{BHT}$ is used with buckets of size $O(1)$, the complexity of algorithm $\mathsf{setup}()$ as well as the group complexity of the output authenticated data structure, by Lemma 3.13, are both $O(m)$. However in our case, since the size of the "buckets" sums to $M \geq m$ (and not to $O(m)$ as it happens with the authenticated hash table), both these complexities are $O(m + M)$. □

Algorithm $\{D_{h+1}, \mathsf{auth}(D_{h+1}), d_{h+1}, \mathsf{upd}\} \leftarrow \mathsf{update}(u, D_h, \mathsf{auth}(D_h), d_h, \mathsf{sk}, \mathsf{pk})$**:** Suppose the update u is *insert element* $x \in \mathcal{U}$ *into set* S_i. The algorithm initially sets

$$\mathsf{acc}(S_i) = \mathsf{acc}(S_i)^{x+s}\,,$$

if the update is an insertion. If the update is a deletion, the algorithm sets

$$\mathsf{acc}(S_i) = \mathsf{acc}(S_i)^{(x+s)^{-1}}\,,$$

i.e., it updates the accumulation value that corresponds to the updated set. Then it calls $\{D_{h+1}, \mathsf{auth}(D_{h+1}), d_{h+1}, \mathsf{upd}\} \leftarrow \mathcal{BHT}.\mathsf{update}(u, D_h, \mathsf{auth}(D_h), d_h, \mathsf{sk}, \mathsf{pk})$. However, no rebuilding policy is applied here, as is done in $\mathcal{BHT}$ (therefore the complexity is not amortized). Information upd and $\mathsf{auth}(D_{h+1})$ are set equal to $\mathcal{BHT}.\mathsf{upd}$ and $\mathcal{BHT}.\mathsf{auth}(D_{h+1})$ respectively, both enhanced with the updated accumulated value $\mathsf{acc}(S_i)$.

Lemma 5.3 *Algorithm* $\mathsf{update}()$ *of the authenticated data structure scheme* $\mathcal{ASC}$ *has* $O(1)$ *access complexity. Moreover, the update information* upd *output by* $\mathsf{update}()$ *has* $O(1)$ *group complexity.*

Proof: The complexity bounds follow from Lemma 3.14 and by the fact that no rebuilding of the sets collection data structure is employed in this case. □

Algorithm $\{D_{h+1}, \mathsf{auth}(D_{h+1}), d_{h+1}\} \leftarrow \mathsf{refresh}(u, D_h, \mathsf{auth}(D_h), d_h, \mathsf{upd}, \mathsf{pk})$**:** Suppose the update u is *insert element* $x \in \mathcal{U}$ *into set* S_i. The algorithm calls the respective procedure from the authenticated data structure scheme $\mathcal{BHT}$, i.e., it calls $\{D_{h+1}, \mathsf{auth}(D_{h+1}), d_{h+1}\} \leftarrow \mathcal{BHT}.\mathsf{refresh}(u, D_h, \mathsf{auth}(D_h), d_h, \mathsf{upd}, \mathsf{pk})$ (again, no rebuilding policy is applied) and stores the new accumulation value $\mathsf{acc}(S_i)$ contained in upd. The updated authenticated data structure $\mathsf{auth}(D_{h+1})$ is set equal to $\mathcal{BHT}.\mathsf{auth}(D_{h+1})$, enhanced with the updated accumulation value $\mathsf{acc}(S_i)$, contained in upd.

Lemma 5.4 *Algorithm* $\mathsf{refresh}()$ *of the authenticated data structure scheme* $\mathcal{ASC}$ *has* $O(1)$ *access complexity.*

Proof: It follows directly from Lemma 3.17, and since we are not using precomputed witnesses and any rebuilding of the table. □

5.3 Queries and verification

In this section, we show how compact proofs for the answers to set queries (e.g., intersection, union) can be constructed using the authenticated sets collection data structure presented

earlier. The proofs have optimal size $O(t+\delta)$, where t is the size of the query parameters (e.g., $t=2$ for an intersection of two sets) and δ is the answer size (e.g., $\delta=1$ if the intersection consists of one element). Our solutions use polynomial arithmetic, since the basis of our construction involves the bilinear-map accumulator.

We begin with a result, to be used extensively by our methods, related to *certifying algorithms* [63]. Lemma 5.5 states that if the vector of coefficients $\mathbf{a}=[a_n,a_{n-1},\ldots,a_0]$ of a polynomial having roots $\mathbf{x}=[-x_1,-x_2,\ldots,-x_n]$ is claimed to be correct, it can be certified, with high probability, with complexity less than $O(n\log n)$, i.e., without a fast Fourier transform computation (FFT) from scratch (see Lemma 3.15 from Chapter 3). This can be achieved with the following algorithm (not part of the $\mathcal{ASC}$ scheme):

Algorithm $\{\mathsf{accept},\mathsf{reject}\}\leftarrow\mathsf{certify}(\mathbf{a},\mathbf{x},\mathsf{pk})$**:** Pick a random $\kappa\in\mathbb{Z}_p^*$. If $\sum_{i=0}^{n}a_i\kappa^i=\prod_{i=1}^{m}(\kappa+x_i)$, then the algorithm outputs accept, else it outputs reject.

Lemma 5.5 (Verification of polynomial coefficients) *Let* $\mathbf{a}=[a_n,a_{n-1},\ldots,a_0]$ *and* $\mathbf{x}=[x_1,x_2,\ldots,x_n]$. *If* $\mathsf{accept}\leftarrow\mathsf{certify}(\mathbf{a},\mathbf{x},\mathsf{pk})$, *then* $a_n,a_{n-1},\ldots,a_0$ *are the coefficients of the polynomial* $\prod_{i=1}^{n}(s+x_i)$ *with probability* $\Omega(1-\mathsf{neg}(k))$. *Moreover, algorithm* $\mathsf{certify}(\mathbf{a},\mathbf{x},\mathsf{pk})$ *has* $O(n)$ *complexity.*

Proof: Algorithm $\mathsf{certify}()$ has complexity $O(n)$ since it involves $O(n)$ multiplications, additions and exponentiations. The probability that $\mathsf{certify}()$ accepts while $a_0,a_1,\ldots,a_n$ are not the coefficients of the polynomial that has roots $-x_1,-x_2,\ldots,-x_n$ is equal to the probability of κ being the root of the polynomial $R(\kappa)=\sum_{i=0}^{n}a_i\kappa^i-\prod_{i=1}^{m}(\kappa+x_i)$. This follows from polynomial equality that should hold for all κ. Now, polynomial $R(\kappa)$ has degree $n=\mathsf{poly}(k)$ and has $O(n)$ roots. Since κ is picked at random from $\mathbb{Z}_p^*$, it follows that this probability is bounded by $O(\mathsf{poly}(k)/2^k)$, which is $\mathsf{neg}(k)$, and therefore the validity of the coefficients can be verified with probability $\Omega(1-\mathsf{neg}(k))$ with $\Theta(n)$ complexity. □

In the following we describe the algorithms for the queries *intersection*, *union*, *subset* and *set difference* in detail. The parameters of our queries are $t\geq 2$ indices (for subset and set

difference queries it is $t = 2$), namely the indices $i_1, i_2, \ldots, i_t$, with $1 \leq t \leq m$. To simplify the notation, we assume without loss of generality that these indices are $1, 2, \ldots, t$. We denote with n_i the size of set S_i $(i = 1, 2, \ldots, t)$ and we define $N = \sum_{i=1}^{t} n_i$. I.e., N is the total size of the sets involved in the execution of our queries. We repeat that δ denotes the size of our answer (e.g., size of the output intersection). Note, that in all cases $\delta = O(N)$ and that performing the actual operations has $O(N)$ complexity, by using a generalized merge.

5.3.1 Intersection query

We begin with the intersection query. Let $\mathsf{I} = S_1 \cap S_2 \cap \ldots \cap S_t = \{y_1, y_2, \ldots, y_\delta\}$ be the intersection of sets $S_1, S_2, \ldots, S_t$. We express the correctness of the answer I to the intersection query by means of the following two conditions:

$$\textbf{Subset condition: } \mathsf{I} \subseteq S_1 \wedge \mathsf{I} \subseteq S_2 \wedge \ldots \wedge \mathsf{I} \subseteq S_t\,; \tag{5.3}$$

$$\textbf{Completeness condition: } (S_1 - \mathsf{I}) \cap (S_2 - \mathsf{I}) \cap \ldots \cap (S_t - \mathsf{I}) = \emptyset\,. \tag{5.4}$$

Note the completeness condition in Equation 5.4 is necessary since I should contain *all* the common elements. Given an intersection I, and for every set S_j, we define polynomial $P_j(s) = \prod_{x \in S_j - \mathsf{I}}(x + s)$, of degree n_j. We can now state the following lemma:

Lemma 5.6 *Set I that is a subset of sets $S_1, S_2, \ldots, S_t$ is the intersection of sets $S_1, S_2, \ldots, S_t$ if and only if there exist polynomials $q_1(s), q_2(s), \ldots, q_t(s)$ such that $q_1(s)P_1(s) + q_2(s)P_2(s) + \ldots + q_t(s)P_t(s) = 1$. Moreover, computing polynomials $q_1(s), q_2(s), \ldots, q_t(s)$ can be achieved with complexity $O(N \log^2 N \log\log N)$.*

Proof: ($\Rightarrow$) This direction follows by the fact that we can use the extended Euclidean algorithm and find polynomials $q_1(s), \ldots, q_t(s)$ such that

$$q_1(s)P_1(s) + \ldots + q_t(s)P_t(s) = \mathsf{GCD}(P_1(s), P_2(s), \ldots, P_t(s)).$$

Since $P_1(s), P_2(s), \ldots, P_t(s)$ share no common factors, it follows that

$$\mathsf{GCD}(P_1(s), P_2(s), \ldots, P_t(s)) = 1\,.$$

($\Leftarrow$) Suppose there exist polynomials $q_1(s), q_2(s), \ldots, q_t(s)$ that satisfy relation $q_1(s)P_1(s) + q_2(s)P_2(s) + \ldots + q_t(s)P_t(s) = 1$ but I is not the intersection. This means that polynomials $P_1(s), P_2(s), \ldots, P_t(s)$ share at least one common factor, e.g., $(s+r)$. Therefore there exists some polynomial $A(s)$ such that $(s+r)A(s) = 1$, i.e., the polynomials $(s+r)A(s)$ and 1 are *equal*, which is a contradiction (note that we want the polynomials to be equal for every $s \in \mathbb{Z}_p$).

In order to compute these coefficients, we use the extended Euclidean algorithm recursively, based on the fact that the greatest common divisor $\mathsf{GCD}(P_1(s), \ldots, P_t(s))$ equals $\mathsf{GCD}(P_1(s), \mathsf{GCD}(P_2(s), \ldots, P_t(s)))$. To compute the greatest common divisor of two $O(n)$-degree polynomials, we can use the algorithm described in the book by von zur Gathen and Gerhard [40] that has $O(n \log^2 n \log\log n)$ complexity. Since we are using this algorithm t times, the time complexity is bounded by $O(tn \log^2 n \log\log n)$. Moreover, by the property that $x\log x + y\log y \leq (x+y)\log(x+y)$ and since the size of the sets participating in the intersection is N, this equals $O(N \log^2 N \log\log N)$. This algorithm also outputs the required coefficients. If we arrange our data (i.e., t polynomials) on a binary tree, after all the coefficients of the internal nodes have been computed, the final coefficients for all elements at the leaves can be computed in $O(t)$ multiplications (we can avoid the $O(t \log t)$ cost) of $O(n_i)$ degree polynomials, where n_i are the degrees of the polynomials of the leaves. Therefore the result holds. $\square$

We use Lemmata 3.15 and 5.6 to construct efficient proofs for both conditions in Relations 5.3 and 5.4:

Proof of subset condition. For each set S_j, $1 \leq j \leq t$, the *subset witnesses*

$$\mathsf{W}_{\mathsf{I},j} = g^{P_j(s)} = g^{\prod_{x \in S_j - \mathsf{I}}(x+s)} \tag{5.5}$$

are computed, as defined in Relation 5.1.

Proof of completeness condition. Suppose $q_1(s), q_2(s), \ldots, q_t(s)$ are polynomials computed in Lemma 5.6 that satisfy $q_1(s)P_1(s)+q_2(s)P_2(s)+\ldots+q_t(s)P_t(s) = 1$. For $j = 1, \ldots, t$, the *completeness witnesses*

$$\mathsf{F}_{\mathsf{I},j} = g^{q_j(s)} \tag{5.6}$$

are computed. We can now formally define algorithms query() and verify() of the authenticated data structure scheme $\mathcal{ASC}$ and for the *intersection* query:

Algorithm $\{\Pi(q), \alpha(q)\} \leftarrow \mathsf{query}(q, D_h, \mathsf{auth}(D_h), \mathsf{pk})$**:** (*intersection*)
The query q is the set of indices $\{1, 2, \ldots, t\}$, requiring the intersection of $S_1, S_2, \ldots, S_t$. Let $\alpha(q) = \{y_1, y_2, \ldots, y_\delta\}$ be the intersection I. The proof $\Pi(q)$ consisting of the following pieces:

1. *Coefficients* $b_\delta, b_{\delta-1}, \ldots, b_0$ of the polynomial $(s + y_1)(s + y_2)\ldots(s + y_\delta)$;
2. *Accumulation values proofs* $\Pi_j = \{(\alpha_{ji}, \beta_{ji}) : i = 0, \ldots, l\}$, as defined in Relation 3.27, output by calling algorithm $\mathcal{BHT}$.query$(j, D_h, \mathsf{auth}(D_h), \mathsf{pk})$, for all $j = 1, \ldots, t$;
3. *Subset witnesses* $\mathsf{W}_{\mathsf{I},j}$, as defined in Relation 5.5, for all $j = 1, \ldots, t$;
4. *Completeness witnesses* $\mathsf{F}_{\mathsf{I},j} = g^{q_j(s)}$, as defined in Relation 5.6, for all $j = 1, \ldots, t$.

Algorithm $\{\mathsf{accept}, \mathsf{reject}\} \leftarrow \mathsf{verify}(q, \alpha, \Pi, d_h, \mathsf{pk})$**:** (*intersection*)
Given a proof Π and an answer $\alpha = \{y_1, y_2, \ldots, y_\delta\}$, the verification algorithm for the intersection query $S_1 \cap S_2 \cap \ldots \cap S_t$ outputs accept if *all* of the following tests are successful, else it outputs reject:

1. *Coefficients test*: It is $\mathsf{accept} \leftarrow \mathsf{certify}([b_\delta, b_{\delta-1}, \ldots, b_0], [-y_1, -y_2, \ldots, -y_\delta], \mathsf{pk})$;[2]
2. *Accumulation values tests*: For all $j = 1, \ldots, t$, it is

$$\mathsf{accept} \leftarrow \mathcal{BHT}.\mathsf{verify}(j, \mathsf{true}, \Pi_j, d_h, \mathsf{pk})\,;$$

[2] Algorithm certify() is used to achieve optimal verification complexity.

3. *Subset tests*: For all $j = 1, \ldots, t$, it is

$$e\left(\prod_{i=0}^{\delta}\left(g^{s^i}\right)^{b_i}, \mathsf{W}_{\mathsf{I},j}\right) = e\left(\beta_{j0}, g\right) , \tag{5.7}$$

where β_{j0} is taken from Π_j;

4. *Completeness test*: It is

$$\prod_{j=1}^{t} e\left(\mathsf{W}_{\mathsf{I},j}, \mathsf{F}_{\mathsf{I},j}\right) = e(g, g) . \tag{5.8}$$

5.3.2 Union query

The answer to a union query is the set $\mathsf{U} = S_1 \cup S_2 \cup \ldots \cup S_t = \{y_1, y_2, \ldots, y_\delta\}$. We express the correctness of the answer U to the union query by means of the following two conditions:

$$\textbf{Membership condition: } \forall y_i \in \mathsf{U}\ \exists j \in \{1, 2, \ldots, t\} : y_i \in S_j ; \tag{5.9}$$

$$\textbf{Superset condition: } (\mathsf{U} \supseteq S_1) \wedge (\mathsf{U} \supseteq S_2) \wedge \ldots \wedge (\mathsf{U} \supseteq S_t) . \tag{5.10}$$

Note that the *superset condition* is needed to make sure that no element has been excluded from the returned answer U. We now formally describe algorithms $\mathsf{query}()$ and $\mathsf{verify}()$ for the union query.

Algorithm $\{\Pi(q), \alpha(q)\} \leftarrow \mathsf{query}(q, D_h, \mathsf{auth}(D_h), \mathsf{pk})$**:** *(union)*

The query q is the set of indices $\{1, 2, \ldots, t\}$, requiring the union of $S_1, S_2, \ldots, S_t$. Let $\alpha(q) = \{y_1, y_2, \ldots, y_\delta\}$ be the union U. The proof $\Pi(q)$ consisting of the following pieces:

1. *Coefficients* $b_\delta, b_{\delta-1}, \ldots, b_0$ of the polynomial $(s + y_1)(s + y_2) \ldots (s + y_\delta)$;

2. *Accumulation values proofs* $\Pi_j = \{(\alpha_{ji}, \beta_{ji}) : i = 0, \ldots, l\}$, as defined in Relation 3.27, output by calling algorithm $\mathcal{BHT}.\mathsf{query}(j, D_h, \mathsf{auth}(D_h), \mathsf{pk})$, for all $j = 1, \ldots, t$;

3. *Membership witnesses* W_{y_i, S_k} (for some $1 \leq k \leq t$), as defined in Relation 3.4, for all $i = 1, \ldots, \delta$;

4. *Subset witnesses* $\mathsf{W}_{S_j, \mathsf{U}}$, as defined in Relation 5.1, for all $j = 1, \ldots, t$.

Algorithm $\{\mathsf{accept}, \mathsf{reject}\} \leftarrow \mathsf{verify}(q, \alpha, \Pi, d_h, \mathsf{pk})$**:** $(union)$

Given a proof Π and an answer $\alpha = \{y_1, y_2, \ldots, y_\delta\}$, the verification algorithm for the union query $S_1 \cup S_2 \cup \ldots \cup S_t$ outputs accept if *all* of the following tests are successful, else it outputs reject:

1. *Coefficients test*: It is $\mathsf{accept} \leftarrow \mathsf{certify}([b_\delta, b_{\delta-1}, \ldots, b_0], [-y_1, -y_2, \ldots, -y_\delta], \mathsf{pk})$;

2. *Accumulation values tests*: For all $j = 1, \ldots, t$, it is

$$\mathsf{accept} \leftarrow \mathcal{BHT}.\mathsf{verify}(j, \mathsf{true}, \Pi_j, d_h, \mathsf{pk})\,;$$

3. *Membership tests*: For all $i = 1, \ldots, \delta$, it is

$$e\left(\mathsf{W}_{y_i, S_k}, g^{y_i} g^{s}\right) = e\left(\beta_{k0}, g\right)\,,$$

   where β_{k0} is taken from Π_k;

4. *Subset tests*: For all $j = 1, \ldots, t$, it is

$$e\left(\mathsf{W}_{S_j, \mathsf{U}}, \beta_{j0}\right) = e\left(\prod_{i=0}^{\delta} \left(g^{s^i}\right)^{b_i}, g\right)\,,$$

   where β_{j0} is taken from Π_j.

5.3.3 Subset query

The correctness properties we need for the subset query are expressed with the relations

$$S_1 \subseteq S_2 \Leftrightarrow \forall y \in S_1 : y \in S_2\,.$$

Algorithm $\{\Pi(q), \alpha(q)\} \leftarrow \mathsf{query}(q, D_h, \mathsf{auth}(D_h), \mathsf{pk})$**:** $(subset)$

The query q is the set of indices $\{1, 2\}$ (wlog). Let $\alpha(q) = \mathsf{true}$ if $S_1 \subseteq S_2$ or $\alpha(q) = \mathsf{false}$ otherwise. The proof $\Pi(q)$ consisting of the following pieces:

1. *Accumulation values proofs* $\Pi_j = \{(\alpha_{ji}, \beta_{ji}) : i = 0, \ldots, l\}$, as defined in Relation 3.27, output by calling algorithm $\mathcal{BHT}.\mathsf{query}(j, D_h, \mathsf{auth}(D_h), \mathsf{pk})$, for $j = 1, 2$;

2. We distinguish two cases:

   (a) $\alpha(q) = \mathsf{true}$: The proof contains the *subset witness* W_{S_1,S_2} as defined in Relation 5.1;

   (b) $\alpha(q) = \mathsf{false}$: The proof contains a *membership witness* W_{y,S_1} (for some y) as defined in Relation 3.4 and a *non-membership witness* $(\mathsf{A}_y, \mathsf{B}_y)$—that proves that $y \notin S_2$— as defined in Relation 3.5.

Algorithm $\{\mathsf{accept}, \mathsf{reject}\} \leftarrow \mathsf{verify}(q, \alpha, \Pi, d_h, \mathsf{pk})$**:** (*subset*)
Given a proof Π and an answer $\alpha \in \{\mathsf{true}, \mathsf{false}\}$, the verification algorithm for the subset query $S_1 \subseteq S_2$ outputs accept if *all* of the following tests are successful, else it outputs reject:

1. *Accumulation values tests*: For $j = 1, 2$, it is $\mathsf{accept} \leftarrow \mathcal{BHT}.\mathsf{verify}(j, \mathsf{true}, \Pi_j, d_h, \mathsf{pk})$;

2. *(Non)-membership tests*: When $\alpha = \mathsf{true}$, it is $e(\mathsf{W}_{S_1,S_2}, \beta_{10}) = e(\beta_{20}, g)$, otherwise $(\alpha = \mathsf{false})$ it is $e(\mathsf{W}_{y,S_1}, g^y g^s) = e(\beta_{10}, g)$ (verification of membership of y in S_1) and

$$e(g^y g^s, \mathsf{A}_y) e(\beta_{20}, \mathsf{B}_y) = e(g, g)$$

   (verification of non-membership of y in S_2), where β_{10} and β_{20} are taken from Π_1 and Π_2.

5.3.4 Set difference query

The correctness properties for a set difference query are expressed with the following relations. It is

$$\mathsf{D} = S_1 - S_2 \Leftrightarrow \mathsf{D} \subseteq S_1 \wedge S_1 - \mathsf{D} = S_1 \cap S_2\,.$$

Algorithm $\{\Pi(q), \alpha(q)\} \leftarrow \mathsf{query}(q, D_h, \mathsf{auth}(D_h), \mathsf{pk})$**:** (*set difference*)
The query q is the set of indices $\{1, 2\}$ (wlog), requiring the difference $S_1 - S_2$. Let $\alpha(q) = \{y_1, y_2, \ldots, y_\delta\}$ be the difference D. The proof $\Pi(q)$ consisting of the following pieces:

1. *Coefficients* $b_\delta, b_{\delta-1}, \ldots, b_0$ of the polynomial $(s+y_1)(s+y_2)\ldots(s+y_\delta)$;
2. *Accumulation values proofs* $\Pi_j = \{(\alpha_{ji}, \beta_{ji}) : i = 0, \ldots, l\}$, as defined in Relation 3.27, output by calling algorithm $\mathcal{BHT}.\mathsf{query}(j, D_h, \mathsf{auth}(D_h), \mathsf{pk})$, for $j = 1, 2$;
3. *Subset witness* $\mathsf{W}_{\mathsf{D},S_1}$, as defined in Relation 5.1;
4. *Subset witnesses* $\mathsf{W}_{S_1-\mathsf{D},1}$ and $\mathsf{W}_{S_1-\mathsf{D},2}$ as defined in Relation 5.5;
5. *Completeness witnesses* $\mathsf{F}_{S_1-\mathsf{D},1}$ and $\mathsf{F}_{S_1-\mathsf{D},2}$ as defined in Relation 5.6.

Algorithm $\{\mathsf{accept}, \mathsf{reject}\} \leftarrow \mathsf{verify}(q, \alpha, \Pi, d_h, \mathsf{pk})$**:** (*set difference*)
Given a proof Π and an answer $\alpha = \{y_1, y_2, \ldots, y_\delta\}$, the verification algorithm for the difference query $S_1 - S_2$ outputs accept if *all* of the following tests are successful, else it outputs reject:

1. *Coefficients test*: It is $\mathsf{accept} \leftarrow \mathsf{certify}([b_\delta, b_{\delta-1}, \ldots, b_0], [-y_1, -y_2, \ldots, -y_\delta], \mathsf{pk})$;
2. *Accumulation values tests*: For all $j = 1, 2$, it is $\mathsf{accept} \leftarrow \mathcal{BHT}.\mathsf{verify}(j, \mathsf{true}, \Pi_j, d_h, \mathsf{pk})$;
3. *Subset tests*: It is $e\left(\mathsf{W}_{\mathsf{D},S_1}, \prod_{i=0}^{\delta}\left(g^{s^i}\right)^{b_i}\right) = e\left(\beta_{10}, g\right)$ and
   (a) $e\left(\mathsf{W}_{S_1-\mathsf{D},1}, \mathsf{W}_{\mathsf{D},S_1}\right) = e(\beta_{10}, g)$,
   (b) $e\left(\mathsf{W}_{S_1-\mathsf{D},2}, \mathsf{W}_{\mathsf{D},S_1}\right) = e(\beta_{20}, g)$,
   where β_{10} and β_{20} are taken from Π_1 and Π_2;
4. *Completeness test*: It is $e\left(\mathsf{W}_{S_1-\mathsf{D},1}, \mathsf{F}_{S_1-\mathsf{D},1}\right) e\left(\mathsf{W}_{S_1-\mathsf{D},2}, \mathsf{F}_{S_1-\mathsf{D},2}\right) = e\left(g, g\right)$.

This concludes the description of the verification and the query algorithms for all four set operations supported by the authenticated data structure scheme $\mathcal{ASC}$.

5.4 Complexity

Let now $n_1, n_2, \ldots, n_t$ be the sizes of the involved sets in our queries and $N = \sum_{i=1}^{t} n_i$. We have the following result:

Lemma 5.7 *For all queries q, algorithm* query() *of the authenticated data structure scheme $\mathcal{ASC}$ has $O(N \log^2 N \log\log N + tm^{\epsilon} \log m)$ access complexity. Moreover, it outputs a proof $\Pi(q)$ of $O(t + \delta)$ group complexity.*

Proof: For all queries involving t sets $S_1, S_2, \ldots, S_t$, accumulation proofs $\Pi_1, \Pi_2, \ldots, \Pi_t$ have to be constructed, by using the authenticated data structure scheme algorithm $\mathcal{BHT}$.query(). By Lemma 3.18 (no precomputed witnesses), this requires $O(tm^{\epsilon} \log m)$ complexity and the output proofs $\Pi_1, \Pi_2, \ldots, \Pi_t$ have $O(t)$ group complexity. Moreover:

- Queries intersection, union and set difference require the computation of the coefficients $b_{\delta}, b_{\delta-1}, \ldots, b_0$ of the polynomial that has roots $-y_1, -y_2, \ldots, -y_{\delta}$. This task, by Lemma 3.15 has $O(\delta \log \delta) = O(N \log N)$ complexity, since $\delta \leq N$. Since $b_{\delta}, b_{\delta-1}, \ldots, b_0 \in \mathbb{Z}_p$, their total group complexity is $O(\delta)$;

- All queries require computing t subset witnesses (note that for the subset and set difference queries it is $t = O(1)$). By Lemma 3.15 and by the definition of subset witnesses in Relation 5.1, computing the subset witnesses has
$$O\left(\sum_{i=1}^{t}(n_i - \delta)\log(n_i - \delta)\right) = O(N \log N)$$
complexity. Since all subset witnesses are elements in $\mathbb{G}$, their total group complexity is $O(t)$. Moreover, for the union query, δ membership witnesses have to be computed. This, by Lemma 3.15 has complexity that is bounded above by $O(N \log N)$. Also, these membership witnesses are elements in $\mathbb{G}$, therefore their total group complexity is $O(\delta)$;

- Queries intersection, subset and set difference require computing t completeness (or non-membership) witnesses (note that for the subset and set difference queries it is $t =$

$O(1)$), which involves running the extended Euclidean algorithm. By Lemma 5.6, this task has $O(N \log^2 N \log\log N)$ complexity. The group complexity of these witnesses is $O(t)$, since they are elements in $\mathbb{G}$.

Summing up, we conclude that the proof has always group complexity $O(t+\delta)$ (hence, operation-sensitive) and the complexity to compute it is $O(N \log^2 N \log\log N + tm^{\epsilon} \log m)$ for all queries, except for the union proof, which requires slightly less complexity, i.e., $O(N \log N + tm^{\epsilon} \log m)$. □

Lemma 5.8 *Algorithm* verify() *of the authenticated data structure scheme* $\mathcal{ASC}$ *has* $O(t+\delta)$ *access complexity.*

Proof: Algorithm certify() has $O(\delta)$ complexity, by Lemma 5.5. Also, the verification algorithm for all queries performs a number of constant-complexity operations—such as verification of proofs Π_i with $\mathcal{BHT}$.verify() (see Lemma 3.19) and bilinear-map computations—, that is proportional to $t+\delta$. Therefore the access complexity of verify() is $O(t+\delta)$. □

5.5 Proof of correctness

Lemma 5.9 *The authenticated data structure scheme* $\mathcal{ASC}$ = {genkey, setup, update, refresh, query, verify} *that uses the correct authenticated data structure scheme* $\mathcal{BHT}$ *from Chapter 3 is correct according to Definition 2.4.*

Proof: Let D_0 be any sets collection data structure containing m sets. Fix the security parameter k and output $\mathsf{pk} = \{h(.), (p, \mathbb{G}, \mathcal{G}, e, g), g^s, g^{s^2}, \ldots, g^{s^q}\}$ and $\mathsf{sk} = s$ by calling algorithm genkey(). Then output an authenticated data structure auth(D_0) and the respective digest d_0, by calling algorithm setup(). Pick a polynomial number of updates—namely, pick a polynomial number of elements for insertion (or deletion) into (or from) a set S_r—and update auth(D_0) and d_0 by calling algorithm refresh(). Let D_h be the final sets collection data structure, auth(D_h) be the produced authenticated data structure and d_h be the final

digest. By the way $\mathsf{refresh}()$ operates, at every time, the digest $d(v_j)$ of a leaf node v_j (that corresponds to set S_j) of the tree T is

$$d(v_j) = \mathsf{acc}(S_j)^{(s+j)} \,. \tag{5.11}$$

We prove correctness for all four query operations, i.e., for intersection, union, subset and difference.

Intersection. Let our query q be $\{1, 2, \ldots, t\}$ (wlog), i.e., a set of indices that refers to the intersection of sets $S_1, S_2, \ldots, S_t$. Algorithm $\mathsf{query}()$ outputs the proof $\Pi(q)$ and the *correct* answer $\mathsf{I} = \{y_1, y_2, \ldots, y_\delta\} = S_1 \cap S_2 \cap \ldots \cap S_t$. The proof $\Pi(q)$ for the intersection contains the following parts:

1. The coefficients $b_\delta, b_{\delta-1}, \ldots, b_0$ of polynomial $(s + y_1)(s + y_2)\ldots(s + y_\delta)$ associated with the intersection $\mathsf{I} = \{y_1, y_2, \ldots, y_\delta\}$. Since for every $\kappa \in \mathbb{Z}_p$ it is $\sum_{i=0}^{\delta} b_i \kappa^i = \prod_{i=1}^{\delta}(\kappa + y_i)$, Algorithm $\mathsf{certify}()$ accepts;

2. The proofs Π_j, output by $\mathcal{BHT}.\mathsf{query}(j, D_h, \mathsf{auth}(D_h), \mathsf{pk})$, for $j = 1, \ldots, t$. We recall that each proof Π_j is the ordered sequence $(\alpha_{ji}, \beta_{ji})$ for $i = 0, \ldots, l$, as defined in Relations 3.27. Specifically, by Relations 3.27 it should be $\beta_{j0} = d(v_j)^{(s+j)^{-1}}$, which by Relation 5.11 gives

$$\beta_{j0} = \mathsf{acc}(S_j) \,. \tag{5.12}$$

   Now, by the correctness of the scheme $\mathcal{BHT}$, $\mathcal{BHT}.\mathsf{verify}(j, \mathsf{true}, \Pi_j, d_h, \mathsf{pk})$, on inputs Π_j output by $\mathcal{BHT}.\mathsf{query}(j, D_h, \mathsf{auth}(D_h))$, always accepts (see Lemma 3.20);

3. The subset witnesses $\mathsf{W}_{\mathsf{I},j} = g^{P_j(s)} = g^{\prod_{x \in S_j - \mathsf{I}}(x+s)}$ for $j = 1, \ldots, t$. The equality

$$e\left(\prod_{i=0}^{\delta}\left(g^{s^i}\right)^{b_i}, \mathsf{W}_{\mathsf{I},j}\right) = e(\beta_{j0}, g)\,,$$

   is always true, by the properties of the bilinear map, by Relation 5.12 and by the fact that $\sum_{i=0}^{\delta} b_i s^i = \prod_{i=1}^{\delta}(s + y_i)$ (Item 1);

4. The completeness witnesses $\mathsf{F}_{\mathsf{I},j} = g^{q_j(s)}$ for $j = 1, \ldots, t$. The following equality

$$\prod_{j=1}^{t} e\left(\mathsf{W}_{\mathsf{I},j}, \mathsf{F}_{\mathsf{I},j}\right) = e(g,g)^{\sum_{j=1}^{t} q_j(s) P_j(s)} = e(g,g)\,,$$

is always true: By construction of the completeness witnesses it should be

$$\sum_{j=1}^{t} q_j(s) P_j(s) = 1\,.$$

This completes the proof of correctness for the case of intersection, since we proved that for every intersection query q and for every correct answer and proof output by $\mathsf{query}()$, $\mathsf{verify}()$ always accepts.

Union. Let our query q be $\{1, 2, \ldots, t\}$ (wlog), i.e., a set of indices that refers to the union of sets $S_1, S_2, \ldots, S_t$. Algorithm $\mathsf{query}()$ outputs the proof $\Pi(q)$ and the *correct* answer $\mathsf{U} = \{y_1, y_2, \ldots, y_\delta\} = S_1 \cup S_2 \cup \ldots \cup S_t$. The proof $\Pi(q)$ for a union contains the following parts:

1. The coefficients $b_\delta, b_{\delta-1}, \ldots, b_0$ and the proofs Π_j. These are always verified as in the case of intersection. See Items 1 and 2 above;

2. The membership witnesses W_{y_i,S_k} for some $k = 1, \ldots, t$, for each element y_i $(i = 1, \ldots, \delta)$. For $i = 1, \ldots, \delta$, it is $e(\mathsf{W}_{y_i,S_k}, g^{y_i} g^{s}) = e(\beta_{k0}, g)$, since W_{y_i,S_k} is the subset witness as defined in Relation 5.1 and $\beta_{k0} = \mathsf{acc}(S_k)$, by Relation 5.12;

3. The subset witnesses $\mathsf{W}_{S_j,\mathsf{U}}$, for all $j = 1, \ldots, t$. For all $j = 1, \ldots, t$ it is

$$e(\mathsf{W}_{S_j,\mathsf{U}}, \beta_{j0}) = e\left(\prod_{i=0}^{\delta} \left(g^{s^i}\right)^{b_i}, g\right)\,,$$

where $\mathsf{W}_{S_j,\mathsf{U}}$ is the subset witness of S_j with respect to U (the coefficients of which are $b_0, b_1, \ldots, b_\delta$), as defined in Relation 5.1 and $\mathsf{U} \supseteq S_j$ for all $j = 1, \ldots, t$ Therefore this relation also verifies, since $\beta_{j0} = \mathsf{acc}(S_j)$ by Relation 5.12.

This completes the proof of correctness for the case of union, since we proved that for every union query q and for every correct answer and proof output by $\mathsf{query}()$, $\mathsf{verify}()$ always accepts.

Subset. Let the query be *is* $S_1 \subseteq S_2$*?* (wlog). Algorithm $\mathsf{query}()$ outputs the proof $\Pi(q)$ and the *correct* answer, i.e., either true or false. The proof $\Pi(q)$ for a subset query contains the following parts:

1. The accumulation values Π_1 and Π_2. These are always verified as in the case of intersection. See Item 2 in the proof of correctness of the intersection operation;

2. Depending on whether we have a positive or a negative answer, we distinguish the following cases:

   - Positive answer, i.e., S_1 is a subset of S_2. The proof contains the subset witness W_{S_1,S_2}. Then it is $e(\mathsf{W}_{S_1,S_2}, \beta_{10}) = e(\beta_{20}, g)$, by the definition of W_{S_1,S_2} (see Relation 5.1), since $S_1 \subseteq S_2$ and since $\beta_{10} = \mathsf{acc}(S_1)$ and $\beta_{20} = \mathsf{acc}(S_2)$, by Relation 5.12;

   - Negative answer, i.e., S_1 is not a subset of S_2. The proof contains an element y such that $y \in S_1$ but $y \notin S_2$, the respective membership witness W_{y,S_1} and a non-membership proof $(\mathsf{A}_y, \mathsf{B}_y)$. It is $e(\mathsf{W}_{y,S_1}, g^y g^s) = e(\beta_{10}, g)$, by definition of W_{y,S_1} in Relation 5.1, since $y \in S_1$ and since $\beta_{10} = \mathsf{acc}(S_1)$, by Relation 5.12. Also it holds $e(g^y g^s, \mathsf{A}_y) e(\beta_{20}, \mathsf{B}_y) = e(g, g)$, since $\mathsf{A}_y = g^{q(s)}$ and $\mathsf{B}_y = g^{p(s)}$ are such that $(y+s)q(s) + p(s)\prod_{x \in S_2}(x+s) = 1$, $y \notin S_2$ and $\beta_{20} = \mathsf{acc}(S_2)$, by Relation 5.12.

This completes the proof of correctness for the case of the subset query, since we proved that for every subset query q and for every correct answer and proof output by $\mathsf{query}()$, $\mathsf{verify}()$ always accepts.

Set difference. Let our query q be $S_1 - S_2$ (wlog). Algorithm $\mathsf{query}()$ outputs the proof $\Pi(q)$ and the *correct* answer $\mathsf{D} = S_1 - S_2 = \{y_1, y_2, \ldots, y_\delta\}$. The proof $\Pi(q)$ for a difference query contains the following parts:

1. The coefficients $b_\delta, b_{\delta-1}, \ldots, b_0$ (that relate to the difference $\{y_1, y_2, \ldots, y_\delta\}$) and the proofs Π_1 and Π_2. These are always verified as in the case of intersection. See Items 1 and 2 in the proof of correctness of the intersection query;

2. The subset witness $\mathsf{W}_{\mathsf{D},S_1}$. Then it is $e\left(\mathsf{W}_{\mathsf{D},S_1}, \prod_{i=0}^{\delta}\left(g^{s^i}\right)^{b_i}\right) = e(\beta_{10}, g)$, by the definition of $\mathsf{W}_{\mathsf{D},S_1}$ (see Relation 5.1), since $\mathsf{D} \subseteq S_1$ and since $\beta_{10} = \mathsf{acc}(S_1)$ by Relation 5.12;

3. Note now that $\mathsf{W}_{\mathsf{D},S_1} = g^{\prod_{x \in S_1 - \mathsf{D}}(x+s)}$. The remaining relations involving the subset witnesses $\mathsf{W}_{S_1-\mathsf{D},1}, \mathsf{W}_{S_1-\mathsf{D},2}$ and the completeness witnesses $\mathsf{F}_{S_1-\mathsf{D},1}, \mathsf{F}_{S_1-\mathsf{D},1}$ always verify since they comprise an intersection proof, i.e., the proof that $S_1 - \mathsf{D} = S_1 \cap S_2$ and we have already shown the correctness of the intersection operation.

This completes the proof of correctness for *all* the queries supported by sets collection, since we proved that for every query q (intersection/union/subset/difference) and for every correct answer and proof output by $\mathsf{query}()$, $\mathsf{verify}()$ always accepts. □

5.6 Proof of security

Lemma 5.10 *The authenticated data structure scheme* $\mathcal{ASC} = \{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$ *that uses the secure authenticated data structure scheme* $\mathcal{BHT}$ *from Chapter 3 is secure according to Definition 2.5 and under the bilinear q-strong Diffie-Hellman assumption.*

Proof: Let Adv be a computationally-bounded adversary, D_0 be a sets collection data structure consisting of m sets $S_1, S_2, \ldots, S_m$, $\mathcal{ASC} = \{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$ be our authenticated data structure scheme, k be the security parameter and $\{\mathsf{sk}, \mathsf{pk}\} \leftarrow \mathsf{genkey}(1^k)$. The adversary Adv is given the public key pk, namely he is given the values $\{h(.), (p, \mathbb{G}, \mathcal{G}, e, g), g^s, g^{s^2}, \ldots, g^{s^q}\}$ and unlimited access to all the algorithms of $\mathcal{ASC}$, except for $\mathsf{setup}()$ and $\mathsf{update}()$ to which he only has oracle access. The adversary initially outputs the authenticated data structure $\mathsf{auth}(D_0)$ and the digest d_0, through an oracle call

to algorithm $\mathsf{setup}()$. Then the adversary picks a polynomial number of updates (e.g., insert an element x into a set S_r) and eventually outputs the data structure D_h, the authenticated data structure $\mathsf{auth}(D_h)$ and the digest d_h through oracle access to $\mathsf{update}()$. Note that since d_h, the digest of the authenticated data structure, is produced through oracle access to $\mathsf{setup}()$ and $\mathsf{update}()$, it follows that it is the correct one. We now prove the security of each operation separately. For each operation we will express the probability of Definition 2.5 as the intersection of several events that we are going to define precisely below. Then, by using well-accepted assumptions already introduced, we are going to prove that this probability is negligible.

Intersection. Let the intersection query be a set of indices $\{1, 2, \ldots, t\}$ (wlog). The adversary Adv outputs an incorrect answer $\mathsf{I} = \{e_1, e_2, \ldots, e_\delta\} \neq S_1 \cap S_2 \cap \ldots \cap S_t$ and also a proof that consists of the following elements:

1. Coefficients $\gamma_\delta, \gamma_{\delta-1}, \ldots, \gamma_0$;
2. Proofs $\Pi_1, \Pi_2, \ldots, \Pi_t$;
3. Subset witnesses $\mathsf{W}_1, \mathsf{W}_2, \ldots, \mathsf{W}_t$;
4. Completeness witnesses $\mathsf{F}_1, \mathsf{F}_2, \ldots, \mathsf{F}_t$.

We define now the following events, related to the choice of the proof above made by the adversary. Our goal will be to the express the probability of the security definition (Definition 2.5) as a function of the following events.

- $\mathcal{E}_1$: The values $\gamma = [\gamma_\delta, \gamma_{\delta-1}, \ldots, \gamma_0]$ and the answer $\mathbf{e} = \{e_1, e_2, \ldots, e_\delta\}$ picked by Adv are such that $\mathsf{accept} \leftarrow \mathsf{certify}(\gamma, \mathbf{e}, \mathsf{pk})$. Event $\mathcal{E}_1$ can be partitioned into two mutually exclusive events $\mathcal{E}_{1,0}$ and $\mathcal{E}_{1,1}$, i.e, $\mathcal{E}_1 = \mathcal{E}_{1,0} \cup \mathcal{E}_{1,1}$:
  - $\mathcal{E}_{1,0}$: The coefficients $\gamma_\delta, \gamma_{\delta-1}, \ldots, \gamma_0$ are not the coefficients of the polynomial $(s + e_1)(s + e_2) \ldots (s + e_\delta)$;

  - $\mathcal{E}_{1,1}$: The coefficients $\gamma_\delta, \gamma_{\delta-1}, \ldots, \gamma_0$ are the coefficients of the polynomial $(s + e_1)(s + e_2) \ldots (s + e_\delta)$.

- $\mathcal{E}_2$: The proofs $\Pi_1, \Pi_2, \ldots, \Pi_t$ picked by Adv are accepted by algorithm $\mathcal{BHT}.\mathsf{verify}()$, i.e., it is $\mathsf{accept} \leftarrow \mathcal{BHT}.\mathsf{verify}(j, \mathsf{true}, \Pi_j, d_h, \mathsf{pk})$, for all $j = 1, \ldots, t$. Let

$$(\beta_{j0}, \alpha_{j0}) \tag{5.13}$$

  be the first element of the proof Π_j. Event $\mathcal{E}_2$ can be partitioned into two mutually exclusive events $\mathcal{E}_{2,0}$ and $\mathcal{E}_{2,1}$, i.e, $\mathcal{E}_2 = \mathcal{E}_{2,0} \cup \mathcal{E}_{2,1}$:

  - $\mathcal{E}_{2,0}$: There exists $j \in \{1, 2, \ldots, t\}$ such that $\beta_{j0} \neq \mathsf{acc}(S_j)$;
  - $\mathcal{E}_{2,1}$: For all $j = 1, \ldots, t$ it is $\beta_{j0} = \mathsf{acc}(S_j)$.

- $\mathcal{E}_3$: The values $\gamma_\delta, \gamma_{\delta-1}, \ldots, \gamma_0$, $\mathsf{W}_1, \mathsf{W}_2, \ldots, \mathsf{W}_t$ and $\beta_{10}, \beta_{20}, \ldots, \beta_{t0}$, which are contained in $\Pi_1, \Pi_2, \ldots, \Pi_t$, picked by Adv satisfy

$$e\left(\prod_{i=0}^{\delta} \left(g^{s^i}\right)^{\gamma_i}, \mathsf{W}_j\right) = e\left(\beta_{j0}, g\right) \text{ for } j = 1, \ldots, t\,.$$

  Event $\mathcal{E}_3$ can be partitioned into two mutually exclusive events $\mathcal{E}_{3,0}$ and $\mathcal{E}_{3,1}$, i.e, $\mathcal{E}_3 = \mathcal{E}_{3,0} \cup \mathcal{E}_{3,1}$:

  - $\mathcal{E}_{3,0}$: There exists $j \in \{1, 2, \ldots, t\}$ such that the opposites of the roots of the polynomial $\sum_{i=0}^{\delta} \gamma_i s^i$ are not a subset of S_j;
  - $\mathcal{E}_{3,1}$: The opposites of the roots of the polynomial $\sum_{i=0}^{\delta} \gamma_i s^i$ are a subset of S_j for all $j = 1, \ldots, t$.

- $\mathcal{E}_4$: The values $\mathsf{W}_1, \mathsf{W}_2, \ldots, \mathsf{W}_t$ and $\mathsf{F}_1, \mathsf{F}_2, \ldots, \mathsf{F}_t$ picked by Adv satisfy $\prod_{j=1}^{t} e\left(\mathsf{W}_j, \mathsf{F}_j\right) = e(g, g)$;

- $\mathcal{F}$: The answer (intersection) I picked by Adv is not correct, i.e., $\mathsf{I} = \{e_1, e_2, \ldots, e_\delta\} \neq S_1 \cap S_2 \cap \ldots \cap S_t$.

Let now $\mathcal{P}$ be the probability of Definition 2.5, i.e., it is

$$\mathcal{P} = \Pr\left[\begin{array}{ll} \{Q, \Pi, \alpha, h\} \leftarrow \mathsf{Adv}(1^k, \mathsf{pk}); & \mathsf{accept} \leftarrow \mathsf{verify}(Q, \alpha, \Pi, d_h, \mathsf{pk}); \\ & \mathsf{reject} \leftarrow \textbf{check}(Q, \alpha, D_{ih}). \end{array}\right].$$

We recall that the authenticated data structure scheme $\mathcal{ASC}$ is secure if $\mathcal{P} \leq \nu(k)$, where $\nu(k)$ is $\mathsf{neg}(k)$. We observe that for the case of the intersection query, $\mathcal{P}$ can be expressed as the probability of the intersection of the events $\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3, \mathcal{E}_4, \mathcal{F}$. By using simple probability calculus, this can be written as

$$\begin{aligned} \mathcal{P} &= \Pr\left[\mathcal{E}_1 \cap \mathcal{E}_2 \cap \mathcal{E}_3 \cap \mathcal{E}_4 \cap \mathcal{F}\right] = \Pr\left[(\mathcal{E}_{1,0} \cup \mathcal{E}_{1,1}) \cap (\mathcal{E}_{2,0} \cup \mathcal{E}_{2,1}) \cap (\mathcal{E}_{3,0} \cup \mathcal{E}_{3,1}) \cap \mathcal{E}_4 \cap \mathcal{F}\right] \\ &\leq \Pr[\mathcal{E}_{1,0}] + \Pr\left[\mathcal{E}_{1,1} \cap (\mathcal{E}_{2,0} \cup \mathcal{E}_{2,1}) \cap (\mathcal{E}_{3,0} \cup \mathcal{E}_{3,1}) \cap \mathcal{E}_4 \cap \mathcal{F}\right] \\ &\leq \Pr[\mathcal{E}_{1,0}] + \Pr[\mathcal{E}_{2,0}] + \Pr\left[\mathcal{E}_{1,1} \cap \mathcal{E}_{2,1} \cap (\mathcal{E}_{3,0} \cup \mathcal{E}_{3,1}) \cap \mathcal{E}_4 \cap \mathcal{F}\right] \\ &\leq \Pr[\mathcal{E}_{1,0}] + \Pr[\mathcal{E}_{2,0}] + \Pr[\mathcal{E}_{3,0} \cap \mathcal{E}_{2,1} \cap \mathcal{E}_{1,1}] + \Pr\left[\mathcal{E}_{1,1} \cap \mathcal{E}_{2,1} \cap \mathcal{E}_{3,1} \cap \mathcal{E}_4 \cap \mathcal{F}\right] \\ &\leq \Pr[\mathcal{E}_{1,0}] + \Pr[\mathcal{E}_{2,0}] + \Pr[\mathcal{E}_{3,0} | \mathcal{E}_{2,1} \cap \mathcal{E}_{1,1}] + \Pr\left[\mathcal{E}_4 | \mathcal{E}_{3,1} | \mathcal{E}_{2,1} \cap \mathcal{E}_{1,1} \cap \mathcal{F}\right]. \end{aligned}$$

We compute each such probability separately:

1. $\Pr[\mathcal{E}_{1,0}]$ is $\mathsf{neg}(k)$ by Lemma 5.5;

2. $\Pr[\mathcal{E}_{2,0}]$ is $\mathsf{neg}(k)$ by Corollary 3.4 (security of scheme $\mathcal{BHT}$);[3]

3. $\Pr[\mathcal{E}_{3,0} | \mathcal{E}_{2,1} \cap \mathcal{E}_{1,1}]$: For this event we note that the event $\mathcal{E}_{3,0}$ is conditioned on the event $\mathcal{E}_{2,1} \cap \mathcal{E}_{1,1}$. This condition allows us to replace β_{j0} with $\mathsf{acc}(S_j)$ (due to $\mathcal{E}_{2,1}$) and $\sum_{i=0}^{\delta} \gamma_i s^i$ with $\prod_{x \in \mathsf{I}}(x + s)$ (due to $\mathcal{E}_{1,1}$) in the event $\mathcal{E}_{3,0}$. Therefore the event $\mathcal{E}_{3,0} | \mathcal{E}_{2,1} \cap \mathcal{E}_{1,1}$ is the event

$$e\left(g^{\prod_{x \in \mathsf{I}}(x+s)}, \mathsf{W}_j\right) = e\left(\mathsf{acc}(S_j), g\right) \wedge \mathsf{I} \nsubseteq S_j \text{ for some } j \in \{1, 2, \ldots, t\}.$$

   This event implies breaking the bilinear q-strong Diffie-Hellman assumption (Assumption 3.2), by Lemma 5.1. Therefore the probability $\Pr[\mathcal{E}_{3,0} | \mathcal{E}_{2,1} \cap \mathcal{E}_{1,1}]$ is $\mathsf{neg}(k)$;

[3] Note that in order to apply this corollary in the sets collection data structure, we have to consider that the respective "bucket" of the hash table representing the sets collection is $L_j = S_j \cup \{j\}$ and therefore $L_j - \{j\} = S_j$.

4. $\Pr[\mathcal{E}_4|\mathcal{E}_{3,1}|\mathcal{E}_{2,1} \cap \mathcal{E}_{1,1} \cap \mathcal{F}]$: For this event we note that the event $\mathcal{E}_4$ is conditioned on the event $\mathcal{E}_{3,1}|\mathcal{E}_{2,1} \cap \mathcal{E}_{1,1} \cap \mathcal{F}$. This condition allows us to replace β_{j0} with $\mathsf{acc}(S_j)$ (due to $\mathcal{E}_{2,1}$) and $\sum_{i=0}^{\delta} \gamma_i s^i$ with $\prod_{x \in \mathsf{I}}(x+s)$ (due to $\mathcal{E}_{1,1}$) in the event $\mathcal{E}_{3,1}$. Therefore, the event $\mathcal{E}_{3,1}|\mathcal{E}_{2,1} \cap \mathcal{E}_{1,1}$ is the event

$$e\left(g^{\prod_{x\in\mathsf{I}}(x+s)}, \mathsf{W}_j\right) = e\left(\mathsf{acc}(S_j), g\right) \wedge \mathsf{I} \subseteq S_j \text{ for all } j = 1, 2, \ldots, t\,.$$

This is equivalent to writing W_j as the subset witness $\mathsf{W}_{\mathsf{I},S_j}$, i.e.,

$$\mathsf{W}_j = g^{\prod_{x\in S_j - \mathsf{I}}(x+s)} = g^{P_j(s)}\,. \tag{5.14}$$

Note now that $\mathcal{E}_4$ is also conditioned on $\mathcal{F}$. Therefore I is has to be incorrect. Specifically, since $\mathsf{I} \subseteq S_j$ for all $j = 1, \ldots, t$ (due to the condition on $\mathcal{E}_{31}$), it follows that I does not contain all the elements of the intersection, i.e., it is *incomplete*. Thus the polynomials $P_1(s), P_2(s), \ldots, P_t(s)$ (Relation 5.14) have at least one common factor, say $(s+r)$ and it holds $P_j(s) = (s+r)Q_j(s)$ for some polynomials $Q_j(s)$—computable in polynomial time—, for all $j = 1, \ldots, t$. Therefore the event $\mathcal{E}_4|\mathcal{E}_{3,1}| \cap \mathcal{E}_{2,1} \cap \mathcal{E}_{1,1} \cap \mathcal{F}$ implies that

$$\begin{aligned} e(g,g) &= \prod_{j=1}^{t} e\left(\mathsf{W}_j, \mathsf{F}_j\right) = \prod_{j=1}^{t} e\left(g^{P_j(s)}, \mathsf{F}_j\right) = \prod_{j=1}^{t} e\left(g^{(s+r)Q_j(s)}, \mathsf{F}_j\right) \\ &= \prod_{j=1}^{t} e\left(g^{Q_j(s)}, \mathsf{F}_j\right)^{(s+r)} = \left(\prod_{j=1}^{t} e\left(g^{Q_j(s)}, \mathsf{F}_j\right)\right)^{(s+r)}. \end{aligned}$$

Therefore we can derive an $(s+r)$-th root of $e(g,g)$ as

$$e(g,g)^{\frac{1}{s+r}} = \prod_{j=1}^{t} e\left(g^{Q_j(s)}, \mathsf{F}_j\right)\,.$$

This implies breaking the bilinear q-strong Diffie-Hellman assumption for $(p, \mathbb{G}, \mathcal{G}, e, g)$ (Assumption 3.2). By Assumption 3.2, this probability is $\mathsf{neg}(k)$, and therefore $\Pr[\mathcal{E}_4|\mathcal{E}_{3,1}|\mathcal{E}_{2,1} \cap \mathcal{E}_{1,1} \cap \mathcal{F}]$ is $\mathsf{neg}(k)$. Thus the total probability $\mathcal{P}$ is $\mathsf{neg}(k)$. This concludes the proof for the security of an intersection query.

Union. Let the union query be a set of indices $\{1, 2, \ldots, t\}$ (wlog). The adversary Adv outputs an incorrect answer $\mathsf{U} = \{e_1, e_2, \ldots, e_\delta\} \neq S_1 \cup S_2 \cup \ldots \cup S_t$ and also a proof that consists of the following elements:

1. Coefficients $\gamma_\delta, \gamma_{\delta-1}, \ldots, \gamma_0$;
2. Proofs $\Pi_1, \Pi_2, \ldots, \Pi_t$;
3. For each element $e_i \in \mathsf{U}$, membership witnesses $\mathsf{W}_{i,j}$ with reference to some set S_j, where $1 \leq j \leq t$;
4. Subset witnesses $\mathsf{W}_1, \mathsf{W}_2, \ldots, \mathsf{W}_t$ that prove that U is a superset of S_j, for all $j = 1, 2 \ldots, t$.

We define now the following events, related to the choice of the proof above made by the adversary. Our goal will be to the express the probability of the security definition as a function of the following events.

- $\mathcal{E}_1, \mathcal{E}_{1,0}, \mathcal{E}_{1,1}$: Same as in intersection;
- $\mathcal{E}_2, \mathcal{E}_{2,0}, \mathcal{E}_{2,1}$: Same as in intersection;
- $\mathcal{E}_3$: The values $\{e_1, e_2, \ldots, e_\delta\}$, $\mathsf{W}_{1,j_1}, \mathsf{W}_{1,j_2}, \ldots, \mathsf{W}_{1,j_\delta}$ picked by Adv satisfy

  $$e\left(\mathsf{W}_{i,j_i}, g^s g^{e_i}\right) = e(\beta_{j_i 0}, g) \text{ for all } i = 1, \ldots, \delta \text{ and } j_i \in \{1, 2, \ldots, t\},$$

  where $\beta_{j_i 0}$ is the first element of proof Π_{j_i}, as defined in Relation 5.13. Event $\mathcal{E}_3$ can be partitioned into two mutually exclusive events $\mathcal{E}_{3,0}$ and $\mathcal{E}_{3,1}$, i.e, $\mathcal{E}_3 = \mathcal{E}_{3,0} \cup \mathcal{E}_{3,1}$:
  - $\mathcal{E}_{3,0}$: There exists $i \in \{1, 2, \ldots, \delta\}$ such that $e_i \notin S_{j_i}$;
  - $\mathcal{E}_{3,1}$: For all $i = 1, 2, \ldots, \delta$ it is $e_i \in S_{j_i}$.
- $\mathcal{E}_4$: The values $\mathsf{W}_1, \mathsf{W}_2, \ldots, \mathsf{W}_t$, $\beta_{10}, \beta_{20}, \ldots, \beta_{t0}$ (contained in $\Pi_1, \Pi_2, \ldots, \Pi_t$) as well as the values $\gamma_\delta, \gamma_{\delta-1}, \ldots, \gamma_0$ picked by Adv satisfy

  $$e\left(\mathsf{W}_j, \beta_{j0}\right) = e\left(\prod_{i=0}^{\delta} \left(g^{s^i}\right)^{\gamma_i}, g\right).$$

- $\mathcal{F}$: The answer (union) U picked by Adv is not correct, i.e., $\mathsf{U} = \{e_1, e_2, \ldots, e_\delta\} \neq S_1 \cup S_2 \cup \ldots \cup S_t$.

Similarly with the intersection security proof, let $\mathcal{P}$ be the probability of Definition 2.5. We observe that for the case of the union query, $\mathcal{P}$ can be expressed as the probability of the intersection of the events $\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3, \mathcal{E}_4, \mathcal{F}$. By using simple probability calculus (and similarly with the intersection security proof), this can be written as

$$\begin{aligned}\mathcal{P} &= \Pr\left[\mathcal{E}_1 \cap \mathcal{E}_2 \cap \mathcal{E}_3 \cap \mathcal{E}_4 \cap \mathcal{F}\right] = \Pr\left[(\mathcal{E}_{1,0} \cup \mathcal{E}_{1,1}) \cap (\mathcal{E}_{2,0} \cup \mathcal{E}_{2,1}) \cap (\mathcal{E}_{3,0} \cup \mathcal{E}_{3,1}) \cap \mathcal{E}_4 \cap \mathcal{F}\right] \\ &\leq \Pr[\mathcal{E}_{1,0}] + \Pr[\mathcal{E}_{2,0}] + \Pr[\mathcal{E}_{3,0}|\mathcal{E}_{2,1}] + \Pr\left[\mathcal{E}_4|\mathcal{E}_{3,1} \cap \mathcal{E}_{2,1} \cap \mathcal{E}_{1,1} \cap \mathcal{F}\right] .\end{aligned}$$

We compute each such probability separately:

1. $\Pr[\mathcal{E}_{1,0}]$ is $\mathsf{neg}(k)$ by Lemma 5.5;

2. $\Pr[\mathcal{E}_{2,0}]$ is $\mathsf{neg}(k)$ by Corollary 3.4;

3. $\Pr[\mathcal{E}_{3,0}|\mathcal{E}_{2,1}]$: For this event we note that the event $\mathcal{E}_{3,0}$ is conditioned on the event $\mathcal{E}_{2,1}$. This condition allows us to replace β_{j0} with $\mathsf{acc}(S_j)$ in the event $\mathcal{E}_{3,0}$. Therefore the event $\mathcal{E}_{3,0}|\mathcal{E}_{2,1}$ is the event

$$e\left(\mathsf{W}_{i,j_i}, g^s g^{e_i}\right) = e(\mathsf{acc}(S_{j_i}), g) \wedge \exists i \in \{1, 2, \ldots, \delta\} \wedge j_i \in \{1, 2, \ldots, t\} : e_i \notin S_{j_i} .$$

   This event implies breaking the bilinear q-strong Diffie-Hellman assumption (Assumption 3.2), by Lemma 5.1. Therefore the probability $\Pr[\mathcal{E}_{3,0}|\mathcal{E}_{2,1}]$ is $\mathsf{neg}(k)$;

4. $\Pr[\mathcal{E}_4|\mathcal{E}_{3,1} \cap \mathcal{E}_{2,1} \cap \mathcal{E}_{1,1} \cap \mathcal{F}]$: For this event we note that the event $\mathcal{E}_4$ is conditioned on the event $\mathcal{E}_{3,1} \cap \mathcal{E}_{2,1} \cap \mathcal{E}_{1,1} \cap \mathcal{F}$. This condition allows us to replace β_{j0} with $\mathsf{acc}(S_j)$ (due to $\mathcal{E}_{2,1}$) and $\sum_{i=0}^{\delta} \gamma_i s^i$ with $\prod_{x \in \mathsf{U}}(x + s)$ (due to $\mathcal{E}_{1,1}$) in the event $\mathcal{E}_4$. Therefore, the event $\mathcal{E}_4|\mathcal{E}_{3,1} \cap \mathcal{E}_{2,1} \cap \mathcal{E}_{1,1}$ is the event

$$e\left(\mathsf{W}_j, \mathsf{acc}(S_j)\right) = e\left(g^{\prod_{x \in \mathsf{U}}(x+s)}, g\right) . \tag{5.15}$$

Note now that $\mathcal{E}_4$ is also conditioned on $\mathcal{E}_{3,1}$. Thus it holds that all elements $e_i \in \mathsf{U}$ belong to some S_{j_i}. Therefore the reported union cannot contain extra elements. Also, $\mathcal{E}_4$ is conditioned on $\mathcal{F}$ (incorrect union). Therefore the reported union must contain less elements and there should be an S_j $(1 \leq j \leq t)$ that contains an r such that $r \notin \mathsf{U}$. Therefore since Relation 5.15 holds, the adversary Adv can find $P(s)$, $Q(s)$ and α such that

$$e\left(\mathsf{W}_j, \mathsf{acc}(S_j)\right) = e(\mathsf{W}_j, g)^{(s+r)P(s)} = e\left(g^{\prod_{x\in\mathsf{U}}(x+s)}, g\right) = e(g,g)^{(s+r)Q(s)+\alpha}\,.$$

Therefore we can derive an $(s+r)$-th of $e(g,g)$ as

$$e(g,g)^{\frac{1}{s+r}} = e(g, \mathsf{W}_{S_j})^{P(s)/\alpha} e(g,g)^{-Q(s)/\alpha}\,.$$

This implies breaking the bilinear q-strong Diffie-Hellman assumption for the setting $(p, \mathbb{G}, \mathcal{G}, e, g)$ (Assumption 3.2). By Assumption 3.2, this probability is $\mathsf{neg}(k)$, and therefore $\Pr[\mathcal{E}_4 | \mathcal{E}_{3,1} \cap \mathcal{E}_{2,1} \cap \mathcal{E}_{1,1} \cap \mathcal{F}]$ is $\mathsf{neg}(k)$. Thus the total probability $\mathcal{P}$ is $\mathsf{neg}(k)$. This concludes the proof for the security of a union query.

Subset. Let the subset query be $S_1 \subseteq S_2$. For a positive answer, the adversary Adv outputs an incorrect answer false and also a proof that consists of the following elements:

1. Proofs Π_1 and Π_2;
2. A membership witness W_{S_1,S_2} with reference to set S_2.

We define now the following events, related to the choice of the proof above made by the adversary. Our goal will be to the express the probability of the security definition as a function of the following events.

- $\mathcal{E}_2, \mathcal{E}_{2,0}, \mathcal{E}_{2,1}$: Same as in intersection, with the difference that we only refer to two sets, i.e., sets S_1 and S_2;
- $\mathcal{E}_3$: The values β_{10} (contained in Π_1), β_{20} (contained in Π_2) and W_{S_1,S_2} picked by Adv satisfy $e\left(\mathsf{W}_{S_1,S_2}, \beta_{10}\right) = e\left(\beta_{20}, g\right)$.

- $\mathcal{F}$: $S_1 \nsubseteq S_2$.

Similarly with the intersection security proof, let $\mathcal{P}$ be the probability of Definition 2.5. We observe that for the case of the positive subset query, $\mathcal{P}$ can be expressed as the probability of the intersection of the events $\mathcal{E}_2, \mathcal{E}_3, \mathcal{F}$. By using simple probability calculus (and similarly with the intersection security proof), this can be written as

$$\mathcal{P} = \Pr\left[\mathcal{E}_2 \cap \mathcal{E}_3 \cap \mathcal{F}\right] = \Pr\left[(\mathcal{E}_{2,0} \cup \mathcal{E}_{2,1}) \cap \mathcal{E}_3 \cap \mathcal{F}\right] \leq \Pr[\mathcal{E}_{2,0}] + \Pr\left[\mathcal{E}_3 | \mathcal{E}_{2,1} \cap \mathcal{F}\right] .$$

We compute each such probability separately:

1. $\Pr[\mathcal{E}_{2,0}]$ is $\mathsf{neg}(k)$ by Lemma 3.4;

2. $\Pr[\mathcal{E}_3 | \mathcal{E}_{2,1} \cap \mathcal{F}]$: For this event we note that the event $\mathcal{E}_3$ is conditioned on the event $\mathcal{E}_{2,1} \cap \mathcal{F}$. This condition allows us to replace β_{10} with $\mathsf{acc}(S_1)$ and β_{20} with $\mathsf{acc}(S_2)$ in the event $\mathcal{E}_3$. Therefore the event $\mathcal{E}_3 | \mathcal{E}_{2,1} \cap \mathcal{F}$ is the event

$$e\left(\mathsf{W}_{S_1,S_2}, \mathsf{acc}(S_1)\right) = e\left(\mathsf{acc}(S_2), g\right) \wedge S_1 \nsubseteq S_2 .$$

   This event implies breaking the bilinear q-strong Diffie-Hellman assumption (Assumption 3.2), by Lemma 5.1. Therefore the probability $\Pr[\mathcal{E}_3 | \mathcal{E}_{2,1} \cap \mathcal{F}]$ is $\mathsf{neg}(k)$;

This concludes the security proof for the case of the positive subset query. For a negative answer, the adversary Adv outputs an incorrect answer true and also a proof that consists of the following elements:

1. Proof Π_1 and Π_2;

2. An element y;

3. A membership witness W_y for element y;

4. A non-membership witness A_y and B_y.

We define now the following events, related to the choice of the proof above made by the adversary. Our goal will be to the express the probability of the security definition (Definition 2.5) as a function of the following events.

- $\mathcal{E}_2$: Same as in the positive answer;
- $\mathcal{E}_3$: The values β_{10}, W_y and y picked by Adv are such that $e(\mathsf{W}_y, g^s g^y) = e(\beta_{10}, g)$. Event $\mathcal{E}_3$ can be partitioned into two mutually exclusive events $\mathcal{E}_{3,0}$ and $\mathcal{E}_{3,1}$, i.e, $\mathcal{E}_3 = \mathcal{E}_{3,0} \cup \mathcal{E}_{3,1}$:
  - $y \in S_1$;
  - $y \notin S_1$.
- $\mathcal{E}_4$: The values y, A_y, B_y and β_{20} picked by Adv are such that $e(g^y g^s, \mathsf{A}_y) e(\beta_{20}, \mathsf{B}_y) = e(g, g)$;
- $\mathcal{F}$: $S_1 \subseteq S_2$.

Similarly with the intersection security proof, let $\mathcal{P}$ be the probability of Definition 2.5. We observe that for the case of the negative subset query, $\mathcal{P}$ can be expressed as the probability of the intersection of the events $\mathcal{E}_2, \mathcal{E}_3, \mathcal{E}_4, \mathcal{F}$. By using simple probability calculus (and similarly with the intersection security proof), this can be written as

$$\begin{aligned} \mathcal{P} &= \Pr\left[\mathcal{E}_4 \cap \mathcal{E}_3 \cap \mathcal{E}_2 \cap \mathcal{F}\right] = \Pr\left[\mathcal{E}_4 \cap (\mathcal{E}_{3,0} \cup \mathcal{E}_{3,1}) \cap (\mathcal{E}_{2,0} \cup \mathcal{E}_{2,1}) \cap \mathcal{F}\right] \\ &\leq \Pr[\mathcal{E}_{2,0}] + \Pr[\mathcal{E}_{3,0} | \mathcal{E}_{2,1}] + \Pr\left[\mathcal{E}_4 | \mathcal{E}_{3,1} \cap \mathcal{E}_{2,1} \cap \mathcal{F}\right] . \end{aligned}$$

We compute each such probability separately:

1. $\Pr[\mathcal{E}_{2,0}]$ is $\mathsf{neg}(k)$ by Lemma 3.4;
2. $\Pr[\mathcal{E}_{3,0} | \mathcal{E}_{2,1}]$: For this event we note that the event $\mathcal{E}_{3,0}$ is conditioned on the event $\mathcal{E}_{2,1}$. This condition allows us to replace β_{10} with $\mathsf{acc}(S_1)$ in the event $\mathcal{E}_{3,0}$. Therefore the event $\mathcal{E}_{3,0} | \mathcal{E}_{2,1}$ is the event

$$e\left(\mathsf{W}_y, g^s g^y\right) = e\left(\mathsf{acc}(S_1), g\right) \wedge y \notin S_1 .$$

This event implies breaking the bilinear q-strong Diffie-Hellman assumption (Assumption 3.2), by Lemma 5.1. Therefore the probability $\Pr[\mathcal{E}_{3,0}|\mathcal{E}_{2,1}]$ is $\mathsf{neg}(k)$;

3. $\Pr[\mathcal{E}_4|\mathcal{E}_{3,1} \cap \mathcal{E}_{2,1} \cap \mathcal{F}]$. Due to the condition on $\mathcal{E}_{3,1} \cap \mathcal{E}_{2,1} \cap \mathcal{F}$ this is the event

$$e\left(g^y g^s, \mathsf{A}_y\right) e\left(\mathsf{acc}(S_2), \mathsf{B}_y\right) = e(g,g) \wedge S_1 \subseteq S_2\,.$$

Since we have the condition on $\mathcal{E}_{3,1} \cap \mathcal{F}$ ($y \in S_1$ and $S_1 \subseteq S_2$), it must be that $y \in S_2$. By the security of the non-membership witness (Lemma 3.3), this implies breaking the q-strong Diffie-Hellman assumption (Assumption 3.2), which happens with probability $\mathsf{neg}(k)$. Thus the total probability $\mathcal{P}$ is $\mathsf{neg}(k)$. This concludes the proof for the security of a negative subset query.

Set difference. Let the difference query be $\mathsf{D} = S_1 - S_2$. The adversary Adv outputs an incorrect answer $\mathsf{D} = \{e_1, e_2, \ldots, e_\delta\} \neq S_1 - S_2$ and also a proof that consists of the following elements:

1. Coefficients $\gamma_\delta, \gamma_{\delta-1}, \ldots, \gamma_0$;
2. Proofs Π_1 and Π_2;
3. A subset witness $\mathsf{W}_{\mathsf{D},S_1}$;
4. A proof $(\mathsf{W}_{S_1-\mathsf{D},1}, \mathsf{W}_{S_1-\mathsf{D},2}, \mathsf{F}_{S_1-\mathsf{D},1}, \mathsf{F}_{S_1-\mathsf{D},2})$ for the intersection $S_1 \cap S_2$.

We define now the following events, related to the choice of the proof above made by the adversary. Our goal will be to the express the probability of the security definition (Definition 2.5) as a function of the following events.

- $\mathcal{E}_1$: Same as in intersection;
- $\mathcal{E}_2$: Same as in subset;

- $\mathcal{E}_3$: The values $\gamma_\delta, \gamma_{\delta-1}, \ldots, \gamma_0$, $\mathsf{W}_{\mathsf{D},S_1}$ and β_{10} (contained in Π_1) picked by Adv satisfy

$$e\left(\mathsf{W}_{\mathsf{D},S_1}, \prod_{i=0}^{\delta}\left(g^{s^i}\right)^{\gamma_i}\right) = e(\beta_{10}, g)\,.$$

  Event $\mathcal{E}_3$ can be partitioned into two mutually exclusive events $\mathcal{E}_{3,0}$ and $\mathcal{E}_{3,1}$, i.e, $\mathcal{E}_3 = \mathcal{E}_{3,0} \cup \mathcal{E}_{3,1}$:

  - $\mathcal{E}_{3,0}$: $\mathsf{D} \nsubseteq S_1$;
  - $\mathcal{E}_{3,1}$: $\mathsf{D} \subseteq S_1$;

- $\mathcal{E}_4$: The values $\mathsf{W}_{\mathsf{D},S_1}, \beta_{10}, \beta_{20}, \mathsf{W}_{S_1-\mathsf{D},1}, \mathsf{W}_{S_1-\mathsf{D},2}, \mathsf{F}_{S_1-\mathsf{D},1}, \mathsf{F}_{S_1-\mathsf{D},2}$ picked by Adv are such that the respective tests for the intersection of S_1 and S_2 are satisfied, i.e.,

  1. $e\left(\mathsf{W}_{S_1-\mathsf{D},1}, \mathsf{W}_{\mathsf{D},S_1}\right) = e(\beta_{10}, g)$;
  2. $e\left(\mathsf{W}_{S_1-\mathsf{D},2}, \mathsf{W}_{\mathsf{D},S_1}\right) = e(\beta_{20}, g)$;
  3. $e\left(\mathsf{W}_{S_1-\mathsf{D},1}, \mathsf{F}_{S_1-\mathsf{D},1}\right) e\left(\mathsf{W}_{S_1-\mathsf{D},2}, \mathsf{F}_{S_1-\mathsf{D},2}\right) = e\left(g, g\right)$.

- $\mathcal{F}$: The difference D is incorrect, i.e., $\mathsf{D} \neq S_1 - S_2$.

Similarly with the intersection security proof, let $\mathcal{P}$ be the probability of Definition 2.5. We observe that for the case of the difference query, $\mathcal{P}$ can be expressed as the probability of the intersection of the events $\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3, \mathcal{E}_4, \mathcal{F}$. By using simple probability calculus (and similarly with the intersection security proof), this can be written as

$$\begin{aligned}
\mathcal{P} &= \Pr\left[\mathcal{E}_4 \cap \mathcal{E}_3 \cap \mathcal{E}_2 \cap \mathcal{E}_1 \cap \mathcal{F}\right] \\
&= \Pr\left[\mathcal{E}_4 \cap (\mathcal{E}_{3,0} \cup \mathcal{E}_{3,1}) \cap (\mathcal{E}_{2,0} \cup \mathcal{E}_{2,1}) \cap (\mathcal{E}_{1,0} \cup \mathcal{E}_{1,1}) \cap \mathcal{F}\right] \\
&\leq \Pr[\mathcal{E}_{1,0}] + \Pr[\mathcal{E}_{2,0}] + \Pr[\mathcal{E}_{3,0} | \mathcal{E}_{2,1} \cap \mathcal{E}_{1,1}] + \Pr[\mathcal{E}_4 | \mathcal{E}_{3,1} \cap \mathcal{E}_{2,1} \cap \mathcal{F}]\,.
\end{aligned}$$

We compute each such probability separately:

1. $\Pr[\mathcal{E}_{1,0}]$ is $\mathsf{neg}(k)$ by Lemma 5.5;
2. $\Pr[\mathcal{E}_{2,0}]$ is $\mathsf{neg}(k)$ by Lemma 3.4;

3. $\Pr[\mathcal{E}_{3,0}|\mathcal{E}_{2,1} \cap \mathcal{E}_{1,1}]$. For the event $\mathcal{E}_{3,0}|\mathcal{E}_{2,1} \cap \mathcal{E}_{1,1}$, by replacing the values of the conditions, we get

$$e\left(\mathsf{W}_{\mathsf{D},S_1}, g^{\prod_{x \in \mathsf{D}}(x+s)}\right) = e(\mathsf{acc}(S_1), g) \wedge \mathsf{D} \nsubseteq S_1 \,.$$

This event implies breaking the bilinear q-strong Diffie-Hellman assumption (Assumption 3.2), by Lemma 5.1. Therefore the probability $\Pr[\mathcal{E}_{3,0}|\mathcal{E}_{2,1} \cap \mathcal{E}_{1,1}]$ is $\mathsf{neg}(k)$;

4. $\Pr[\mathcal{E}_4|\mathcal{E}_{3,1} \cap \mathcal{E}_{2,1} \cap \mathcal{F}]$. By the conditions, since $\mathsf{D} \subseteq S_1$, we can write

$$\mathsf{W}_{\mathsf{D},S_1} = g^{\prod_{x \in S_1 - \mathsf{D}}(x+s)} \,.$$

Therefore the event is equivalent to the conjunction of the following events:

- $e\left(\mathsf{W}_{S_1-\mathsf{D},1}, g^{\prod_{x \in S_1 - \mathsf{D}}(x+s)}\right) = e\left(\mathsf{acc}(S_1), g\right)$;
- $e\left(\mathsf{W}_{S_1-\mathsf{D},2}, g^{\prod_{x \in S_1 - \mathsf{D}}(x+s)}\right) = e\left(\mathsf{acc}(S_2), g\right)$;
- $e\left(\mathsf{W}_{S_1-\mathsf{D},1}, \mathsf{F}_{S_1-\mathsf{D},1}\right) e\left(\mathsf{W}_{S_1-\mathsf{D},2}, \mathsf{F}_{S_1-\mathsf{D},2}\right) = e(g, g)$.

We have already proved (intersection proof) that the probability that the above event holds and $S_1 - \mathsf{D} \neq S_1 \cap S_2$ is $\mathsf{neg}(k)$. However, the event $S_1 - \mathsf{D} \neq S_1 \cap S_2$ is equivalent with the event $\mathsf{D} \neq S_1 - S_2$, which is our event $\mathcal{F}$. Therefore the probability $\Pr[\mathcal{E}_4|\mathcal{E}_{3,1} \cap \mathcal{E}_{2,1} \cap \mathcal{F}]$ is $\mathsf{neg}(k)$.

This completes the proof of security for all the queries of the sets collection data structure. □

Theorem 5.1 *Consider a collection of m sets $S_1, \ldots, S_m$ and let $M = \sum_{i=1}^{m} |S_i|$ and $0 < \epsilon < 1$. For a query operation involving t sets (intersection/union/subset/difference), let N be the sum of the sizes of the involved sets and δ be the answer size. Let now k be the security parameter. Then there exists a publicly-verifiable authenticated data structure scheme $\mathcal{ASC} = \{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$ for a data structure scheme defined for dynamic sets collection data structure D such that:*

1. *It is correct and secure according to Definitions 2.4 and 2.5 and based on the bilinear q-strong Diffie-Hellman assumption;*

2. *The access complexity of* setup() *is* $O(m+M)$*, outputting an authenticated data structure* auth(D) *of* $O(m+M)$ *group complexity;*

3. *The access complexity of* update() *is* $O(1)$*, outputting update information* upd *of* $O(1)$ *group complexity;*

4. *The access complexity of* refresh() *is* $O(1)$*;*

5. *For all queries q (intersection/union/subset/difference), the access complexity of* query() *is* $O(N \log^2 N \log\log N + tm^{\epsilon} \log m)$*, outputting a proof* $\Pi(q)$ *of* $O(t+\delta)$ *group complexity;*

6. *For all queries (intersection/union/subset/difference), the access complexity of* verify() *is* $O(t+\delta)$*.*

Proof: The result follows from Lemmata 5.2, 5.3, 5.4, 5.7, 5.8, 5.9 and 5.10. □

5.6.1 Protocols

Three-party protocol. By using Theorem 2.1 we can easily derive the following corollary that describes the use of the authenticated data structure scheme $\mathcal{ASC}$ of Theorem 5.1 in the three-party model:

Corollary 5.1 *Consider a collection of* m *sets* $S_1, \ldots, S_m$ *and let* $M = \sum_{i=1}^{m} |S_i|$ *and* $0 < \epsilon < 1$*. For a query operation involving* t *sets (intersection/union/subset/difference), let* N *be the sum of the sizes of the involved sets and* δ *be the answer size. Let now* k *be the security parameter and assume that the bilinear q-strong Diffie-Hellman assumption holds. Then there exists a three-party authenticated data structures protocol (see Protocol 2.1) for verifying intersection, union, subset and difference queries* q *on a dynamic sets collection data structure such that:*

1. *The setup at the source has $O(m + M)$ access complexity;*

2. *The update at the source has $O(1)$ access complexity;*

3. *The space needed at the source has $O(m + M)$ group complexity;*

4. *The communication between the source and the server has $O(1)$ group complexity;*

5. *The update at the server has $O(1)$ access complexity;*

6. *For all queries (intersection/union/subset/difference), the query at the server has*

$$O(N \log^2 N \log\log N + tm^{\epsilon} \log m)$$

   access complexity;

7. *The space needed at the server has $O(m + M)$ group complexity;*

8. *For all queries (intersection/union/subset/difference), the communication between the server and the client has $O(t + \delta)$ group complexity;*

9. *For all queries (intersection/union/subset/difference), the verification at the client has $O(t + \delta)$ access complexity;*

10. *For a query q (intersection/union/subset/difference) sent by the client to the server at any time (even after updates), let α be an answer and let π be a proof returned by the server. With probability $\Omega(1 - \mathsf{neg}(k))$, the client accepts the answer α if and only if α is correct.*

Two-party protocol. Since the authenticated data structure scheme $\mathcal{ASC}$ uses the authenticated data structure scheme $\mathcal{BHT}$, for which we have proved that Assumption 2.1 is true (see Corollary 3.6), by Theorems 2.2 and 5.1, we can state the final result for the two-party model:

Corollary 5.2 *Consider a collection of m sets $S_1, \ldots, S_m$ and let $M = \sum_{i=1}^{m} |S_i|$ and $0 < \epsilon < 1$. For a query operation involving t sets (intersection/union/subset/difference), let N be the sum of the sizes of the involved sets and δ be the answer size. Let now k be the security parameter and assume that the bilinear q-strong Diffie-Hellman assumption holds. Then there exists a two-party authenticated data structures protocol (see Protocol 2.2) for verifying intersection, union, subset and difference queries q on a dynamic sets collection data structure such that:*

1. *The protocol requires one round of interaction during updates;*
2. *The setup at the client has $O(m + M)$ access complexity;*
3. *The update at the client has $O(1)$ access complexity;*
4. *For all queries (intersection/union/subset/difference), the verification at the client has $O(t + \delta)$ access complexity;*
5. *The space needed at the client has $O(1)$ group complexity;*
6. *The communication between the client and the server has $O(1)$ group complexity during updates and $O(t + \delta)$ group complexity during queries;*
7. *The update at the server has $O(m^{\epsilon} \log m)$ access complexity;*
8. *For all queries (intersection/union/subset/difference), the query at the server has*

$$O(N \log^2 N \log\log N + tm^{\epsilon} \log m)$$

   access complexity;

9. *The space needed at the server has $O(m + M)$ group complexity;*
10. *For a query q (intersection/union/subset/difference) sent by the client to the server at any time (even after updates), let α be an answer and let π be a proof returned by the*

server. With probability $\Omega(1 - \mathsf{neg}(k))$*, the client accepts the answer* α *if and only if* α *is correct.*

5.7 Applications

In this section we discuss on some applications of the presented authenticated sets collection data structure.

5.7.1 Keyword-search

First of all, we notice that our scheme could be easily used to authenticate *keyword-search queries* implemented by the *inverted index* data structure [9]: Each term in the dictionary corresponds to a set in our sets collection data structure which contains as elements all the documents that include this term. A usual text query for terms m_1 and m_2 returns those documents that are included in both the sets that are represented by m_1 and m_2, i.e., their intersection. By using our scheme, we can easily authenticate any such keyword-search query with costs that are proportional to the size of the answer of the query and not proportional to the amount of data that the algorithm reads in order to process the query. Moreover, the derived *authenticated inverted index* can be efficiently updated as well. We continue now with an extension of the authenticated inverted index, the *timestamped keyword-search*.

5.7.2 Timestamped keyword-search

Apart from applications in web search engines, the inverted index is used in other applications that employ keyword-search as well, such as *email-search*. In email-search, a word dictionary is again maintained, the terms of which are mapped into sets of email messages that contain the specific term. Therefore when we are searching our inbox for emails containing terms m_1 and m_2, an inverted index query is executed. However, it is always desirable in email search to be able to introduce a "second" dimension in searching. For example, a query could be

"give me the emails that contain terms m_1 and m_2 and which were received between time t_1 and t_2", where $t_1 < t_2$. We call this procedure timestamped keyword-search.

One solution for the verification of timestamped keyword-search would be to embed a timestamp in the documents (e.g., each email message) and have the client do the filtering locally, after he has verified—using our scheme—the intersection of the sets that correspond to terms m_1 and m_2. However, this is not *operation-sensitive* at all: The intersection can be a lot bigger than the set resulted after the application of the local filtering, making this straightforward solution inefficient.

We now describe an algorithmic construction to solve this problem. Let $t_1, t_2, \ldots, t_r$ be the discrete timestamps that we are interested in (t_i can be viewed as a certain day of the month). We define a new sets collection data structure as follows: Imagine $t_1, t_2, \ldots, t_r$ are the leaves of a binary tree. We build a *segment tree* [97] on top of these timestamps as follows: Each leaf storing timestamp t_i contains the documents (e.g., email messages) that were received at time t_i. Moreover, the internal nodes of the binary tree contain the documents that correspond to the union (note that this union does not have any common elements) of the documents contained in the children's nodes, recursively defining in this way sets of documents for all the nodes of the tree. Therefore we end up with a new sets collection data structure that is built on top of these $2r - 1$ sets (one set per internal tree node of the tree), namely the sets $T_1, T_2, \ldots, T_{2r-1}$. The timestamped keyword-search is therefore verified by two sets collection data structures, one built on the text terms, namely the sets $S_1, S_2, \ldots, S_m$, and one built on top of the sets of the timestamps, namely the sets $T_1, T_2, \ldots, T_{2r-1}$. Define now the extension of two timestamps $\mathsf{ext}(t_1, t_2)$ to be the *set of sets* T_i that "cover" the interval $[t_1, t_2]$, i.e., namely the set that contains sets the union of which equals the set of all timestamps in $[t_1, t_2]$. One can easily see that for every $1 \leq t_1 \leq t_2 \leq r$, it is $|\mathsf{ext}(t_1, t_2)| = O(\log r)$.

Suppose now we want to verify the documents that contain terms m_1 and m_2 and which were received between t_1 and t_2. Namely our query is described by the parameters

m_1, m_2, t_1, t_2 (in the general case our query is described by t terms $m_1, m_2, \ldots, m_t$ and two timestamps t_1 and t_2—see Corollary 5.3). All we have to do is to verify the intersection of the following sets: (a) the union of sets in $\mathsf{ext}(t_1, t_2)$, (b) S_1 (set that refers to term m_1) and, (c) S_2 (set that refers to term m_2). Let $T_1, T_2, \ldots, T_\ell$ be the disjoint sets that are contained in $\mathsf{ext}(t_1, t_2)$, where $\ell = O(\log r)$. The answer to the query is the set $(S_1 \cap S_2) \cap (T_1 \cup T_2 \cup \ldots \cup T_\ell)$ which can be written as $(S_1 \cap S_2 \cap T_1) \cup (S_1 \cap S_2 \cap T_2) \cup \ldots \cup (S_1 \cap S_2 \cap T_\ell)$. Since T_i are disjoint, each term of the union contributes at least one new term to the answer, and therefore we can verify this query in a nearly *operation-sensitive* way by authenticating $\log r$ intersections separately (note there is an extra $O(\log r)$ multiplicative factor in the complexities of Corollary 5.3).

Corollary 5.3 *Consider a collection of m sets $S_1, \ldots, S_m$, let $M = \sum_{i=1}^{m} |S_i|$, $0 < \epsilon < 1$ and $t_1, t_2, \ldots, t_r$ be discrete timestamps. For a query operation involving in a time interval $[t_1, t_2]$, let t be the number of involved sets, N be the sum of the sizes of the involved sets, and δ be the answer size. There exists an authenticated data structure scheme $\mathcal{TKS} =$ {*genkey*, *setup*, *update*, *refresh*, *query*, *verify*} for a data structure scheme defined for a timestamped keyword-search data structure D with the following properties:*

1. *It is correct and secure according to Definitions 2.4 and 2.5 and based on the bilinear q-strong Diffie-Hellman assumption;*

2. *The access complexity of* $\mathsf{setup}()$ *is $O(m + r + M)$, outputting an authenticated data structure* $\mathsf{auth}(D)$ *of $O(m + M + r)$ group complexity;*

3. *The access complexity of* $\mathsf{update}()$ *is $O(\log r)$, outputting information* upd *of $O(1)$ group complexity;*

4. *The access complexity of* $\mathsf{refresh}()$ *is $O(\log r)$;*

5. *For a time-stamped keyword-search query q, algorithm* $\mathsf{query}()$ *has $O(N \log^2 N \log\log N + t(m+r)^\epsilon \log(m+r))$ access complexity, outputting a proof $\Pi(q)$ of $O(t \log r + \delta)$ group*

complexity;

6. *For a time-stamped keyword-search query, the access complexity of* verify() *is* $O(t \log r + \delta)$.

Note that in the above theorem we do not have a result concerning the verification of union with timestamps. This is due to the following: Using the same notation as we did for the intersection, the answer to the union query, would be the set $(S_1 \cup S_2) \cap (T_1 \cup T_2 \cup \ldots \cup T_\ell)$. The nature of the answer does not allow for any further algebraic processing and therefore in order to authenticate the whole expression, one needs to verify the two unions separately. This leads to a solution that is not operation-sensitive (we recall that the size of out query is $O(t)$), therefore the operation-sensitive verification of this type of queries cannot be achieved with our method—at least in a way similar to the techniques we have used so far. The same applies for the difference queries.

5.8 Analysis

In this section we analyze the costs needed by our solution and compare with experimental results from other works. For bilinear maps and generic-group operations in the bilinear-map accumulator, we used the PBC library [1], a library for pairing-based cryptography, interfaced with C.

5.8.1 System setup

We choose our system parameters as follows. First of all, type A pairings are used, as described in [70]. These pairings are constructed on the curve $y^2 = x^3 + x$ over the base field $\mathbb{F}_q$, where q is a prime number. The multiplicative cyclic group $\mathbb{G}$ we are using is a subgroup of points in $E(\mathbb{F}_q)$, namely a subset of those points of $\mathbb{F}_q$ that belong to the elliptic curve E. Therefore this pairing is symmetric. The order of $E(\mathbb{F}_q)$ is $q + 1$ and the order of the

group $\mathbb{G}$ is some prime factor p of $q+1$. The group of the output of the bilinear map $\mathcal{G}$ is a subgroup of $\mathbb{F}_{q^2}$.

In order to instantiate type A pairings in the PBC library, we have to choose the size of the primes q and p. The main constraint in choosing the bit-sizes of q and p is that we want to make sure that discrete logarithm is difficult in $\mathbb{G}$ (that has order p) and in $\mathbb{F}_{q^2}$. Typical values are 160 bits for p and 512 bits for q. We use the typical value for the size of q, i.e., 512 bits. Note that with this choice of parameters the size of the elements in $\mathbb{G}$ (which have the form (x, y), i.e., points on the elliptic curve) is 1024 bits. Finally, let's assume that the accumulation tree that is built on top of the set digests, has two levels, i.e., $\epsilon = 0.5$.

5.8.2 Communication cost

Here we analyze the communication cost that our scheme has for an intersection of *two* sets. Let's assume that the size of the reported intersection is δ. According to the described query() algorithm for the intersection, the proof (apart from the answer itself), consists of the following values: (a) Two subset witnesses, two completeness witnesses and two proofs (each one of the proofs consist of two proof elements of two group elements each). Therefore the size of all these elements, which are all elements of group $\mathbb{G}$, is not dependent on the size of the intersection and is equal to $2 \times (1024 + 1024 + 4 \times 1024)/8 = 1536$ bytes; (b) The coefficients $b_i \in \mathbb{Z}_p$ (we recall p is 160 bits long) of the intersection, for $i = 1, \ldots, \delta$. These have size $160\delta/8 = 20\delta$ bytes. Therefore the total communication cost is a linear function of δ, i.e., the function $1536 + 20\delta$ (in bytes). We now compare the communication cost of our scheme with the analysis made in [79]. In Table 5.2 we compare with the results presented in Table IV of [79] where various set sizes n_1 and n_2 are used and the size of the intersection δ is always $0.01n_2$. Note that in most cases, our communication cost is a lot less than the one reported in [79]. More importantly, it is *not* dependent on the size of the sets participating in the intersection. In cases that our cost is worse, it is due to the big constants enforced by the use of bilinear pairings and accumulators.

Table 5.2: Comparison of a 2-intersection communication overhead (proof size) of the scheme presented by Morselli et al. [79] with our scheme. Here n_1 and n_2 are the sets sizes that are intersected and δ is the size of the intersection.

n_1	n_2	δ	KB [79]	KB (this work)
1000	1000	10	3.34	1.73
1000	100	1	1.68	1.55
1000	10	0	1.01	1.53
1000	1	0	0.46	1.53
10000	10000	100	26.88	3.53
10000	1000	10	12.15	1.73
10000	100	1	6.86	1.55
10000	10	0	3.08	1.53
100000	100000	1000	263.25	21.53
100000	10000	100	116.13	3.53
100000	1000	10	63.18	1.73
100000	100	1	26.69	1.55

5.8.3 Verification cost

Let exp, mult, add be the times needed to perform an exponentiation, a multiplication and an addition respectively, all modulo p. Let also EXP, MULT be times required for exponentiation and multiplication in group $\mathbb{G}$ and let $\mathcal{EXP}$, $\mathcal{MULT}$ be the respective times in the target group of the bilinear map $\mathcal{G}$. Finally let MAP be the time needed to perform the operation $e(.,.)$. We benchmarked all these operations using the PBC library [1] (version $\mathsf{pbc} - 0.5.7$), on a 64-bit, 2.8GHz Intel based, dual-core, dual-processor machine with 4GB main memory, running Debian Linux, and derived the following times, i.e., $\mathsf{MAP} = 5$ms, $\mathcal{MULT} = 0.005$ms, $\mathsf{exp} = 0.02$ms, $\mathsf{add} = 0.002$ms and $\mathsf{mult} = 0.002$ms.

We analyze now the verification cost of a 2-intersection, required by our scheme. Let S_i and S_j be the sets of the intersection. The verification algorithm, on input the proof has to perform the following tasks: (a) First it verifies the proofs Π_i and Π_j, which requires two bilinear-map computations for each value, therefore taking time $4\mathsf{MAP}$; (b) Then algorithm $\mathsf{certify}()$ is executed. The time needed for this part is $\delta(2\mathsf{mult} + 2\mathsf{add} + \mathsf{exp})$; (c) Then the algorithm checks the *subset condition* which takes time $4\mathsf{MAP}$; (d) Finally it checks the *completeness condition* that takes times $2\mathsf{MAP} + \mathcal{MULT}$. Therefore we see that the total

cost for verification of a 2-intersection of size δ is

$$10\mathsf{MAP} + \delta(2\mathsf{mult} + 2\mathsf{add} + \mathsf{exp}) + \mathcal{MULT}\,,$$

which is a linear function in δ, namely the function $50 + 0.028\delta$ (in ms).

Chapter 6

Optimality with multilinear forms

In the previous chapters of this thesis, we introduced authenticated data structures schemes based on several well-accepted cryptographic primitives such as *accumulators*, *bilinear maps* and *lattices*. Some of these schemes present desirable efficiency characteristics, such as operation sensitivity and parallel algorithms, which would not be achievable with traditional hash-based techniques.

However, none of the authenticated data structure schemes presented so far is *optimal* (i.e., adding *no* extra asymptotic overhead to the respective plain data structure scheme), according to our natural definition of optimality given in Section 2 (see Definition 2.8). One authenticated data structure scheme that is *almost optimal* is the scheme $\mathcal{ASC}$, used for verifying set operations and presented in Chapter 5: Although the verification and communication costs were showed to be optimal ($O(t + \delta)$), the query costs were increased by a polylogarithmic factor (see Theorem 5.1).

As such, in this chapter, we pose the following natural question: Can we construct an optimal authenticated data structure scheme? The answer is yes, but, assuming the existence of a cryptographic primitive that does not exist yet! Moreover, and less importantly, the derived authenticated data structure scheme is *not* publicly verifiable, thus not allowing its use by a three-party protocol (Protocol 2.1). This shows the complication of the problem of achieving optimality in authenticated data structures.

To show the realization of an optimal authenticated data structure, we present an *authenticated dictionary* data structure that is based on a new cryptographic primitive that was proposed by Silverberg and Boneh, namely *multilinear forms* [19], the construction of which remains however an open problem to date. The use of such a primitive gives an authenticated dictionary with constant communication and constant verification complexity, while maintaining all other complexities logarithmic. To the best of our knowledge (see Table 6.1), this is the *first* optimal authenticated dictionary to appear in the literature, as it exactly matches the respective complexities (update, query, and communication complexity) of the optimal dictionary data structure (e.g., implemented as a red-black tree).

The multilinear form cryptographic primitive that is used in our construction can be described as the "multi" version of the well-known *bilinear map*. Although initially used to attack elliptic curve systems [76], bilinear maps (also extensively used in the previous chapters of the thesis), being literally an efficient a tool for solving the decisional Diffie-Hellman problem, eventually proved to be a very useful tool in cryptography (e.g., [16, 17, 18]) after their first appearance in the literature for a "good purpose" [60]. However, the main limitation of bilinear maps is the fact that they cannot be applied twice, i.e., the output element cannot be fed back into the map $e(.,.)$ in an efficient way. Finding such maps, i.e., self-bilinear maps, which could be used in a recursive way to construct multilinear forms, was recently proved to be infeasible for groups that are of interest in cryptography, i.e., groups where the computational Diffie-Hellman problem is hard [27].

However, since *cryptographically interesting* multilinear form generators[1] are not known to exist to date, one can view our work from a different (and more theoretical) angle: A proof through a complexity lower bound of the nonexistence of *optimal* authenticated dictionaries would imply the nonexistence of cryptographically interesting multilinear form generators (see Theorem 6.2). This reveals yet another important relation between two

[1] I.e., multilinear form generators for groups where the discrete log problem is hard, e.g., elliptic curve groups. We call these generators *admissible* later in the paper.

fields—combinatorics and cryptography—and becomes more promising (towards proving nonexistence of cryptographically interesting multilinear form generators) given recent advances in the derivation of general complexity lower bounds for memory checking [35] and authenticated data structures [106].

About mulitlinear forms. Multilinear forms were proposed as a possible useful tool in cryptography in 2003 by Silverberg and Boneh [19]. Since then, no efficient construction of interest in cryptography has appeared. A work similar in nature with ours, where an efficient construction for a cryptographic application based on multilinear forms is presented, is proposed by Lee et al. [64]. The impossibility of deriving multilinear forms through *self-bilinear* maps is investigated by Cheon and Lee [27].

Table 6.1: Asymptotic access and group complexities of various authenticated data structure schemes for a dynamic dictionary storing n elements, compared with the optimal authenticated dictionary $\mathcal{MFD}$ based on multilinear forms and derived in this chapter. Parameter $0 < \epsilon < 1$ is a constant and "M. q-DH" stands for "Multilinear q-strong Diffie-Hellman". The various acronyms used for variables and assumptions have all been defined in Table 3.1. Note that our construction requires two assumptions, namely the assumptions M. q-DH and Generic CR.

	[15, 48, 75, 81]	[11]	[83]	[23, 101]	[51]	[90]	$\mathcal{MFD}$
setup()	n	n	n	n	n	n	n
update()	$\log n$	1	1	1	n^ϵ	1	$\log n$
refresh()	$\log n$	1	n	$n \log n$	n^ϵ	1	$\log n$
query()	$\log n$	n	1	1	n^ϵ	n^ϵ	$\log n$
verify()	$\log n$	n	1	1	1	1	1
proof $\Pi(q)$	$\log n$	n	1	1	1	1	1
info. upd	1	1	1	1	n^ϵ	1	$\log n$
publicly verifiable	yes	yes	yes	yes	yes	yes	no
optimal	no	no	no	no	no	no	yes
assumption	Generic CR	D. Log	B. q-DH	Strong RSA			M. q-DH Generic CR

6.1 Dictionary data structure

In this chapter, the underlying data structure we are using (and for which we are designing an authenticated data structure scheme for) is a *dictionary*. Let $\mathcal{X}$ be a collection of n elements from a *totally-ordered* universe $\mathcal{U}$. Note that the total order *is* a requirement for the dictionary data structure, a property that distinguishes it from a hash table (and thus the difference in their complexities). The data structure scheme {**update**, **query**, **check**} as defined in Definition 2.2 for a dictionary $\mathbf{D}(\mathcal{X})$ is as follows:

1. $\mathbf{y} \leftarrow$ **query**$(a, b, \mathbf{D}(\mathcal{X}))$: Given two elements $a, b \in \mathcal{U}$, with $a \leq b$, return the sorted list of *successive* elements $\mathbf{y} = [y_1\ y_2 \ldots y_{w-1}\ y_w] \subseteq \mathcal{X}$ such that

$$y_1 \leq a \leq y_2 \leq \ldots \leq y_{w-1} \leq b < y_w\,.$$

   This is a general *range search* query. Note that for $a = b$ this query reduces to a membership (or non-membership) query for a outputting the interval of $\mathcal{X}$ containing (or not) a. Answering a range search query can be implemented to have $O(\log n + w)$ worst-case complexity with a red-black tree data structure [29], outputting an answer of size $O(w)$;

2. $\mathbf{D}(\mathcal{X}') \leftarrow$ **update**$(x, \mathbf{D}(\mathcal{X}))$: Given an element $x \in \mathcal{U}$ such that $x \notin \mathcal{X}$, *insert* element x into $\mathcal{X}$ and output $\mathbf{D}(\mathcal{X}')$; Given an element $x \in \mathcal{U}$ such that $x \in \mathcal{X}$, *delete* element x from $\mathcal{X}$ and output $\mathbf{D}(\mathcal{X}')$. Both insertions and deletions can be implemented to have $O(\log n)$ worst-case complexity [29];

3. $\{\mathsf{accept}, \mathsf{reject}\} \leftarrow$ **check**$(a, b, \mathbf{y}, \mathbf{D}(\mathcal{X}))$: If $\mathbf{y} = [y_1\ y_2 \ldots y_{w-1}\ y_w] \subseteq \mathcal{X}$ is a sorted list of w *successive* elements such that $y_1 \leq a \leq y_2 \leq \ldots \leq y_{w-1} \leq b < y_w$, return accept. Else return reject.

6.1.1 Non-optimal authenticated dictionaries

To verify range search queries on a dictionary of n elements, we can use many authenticated data structure schemes extensively described in the literature, e.g., various hierarchical hashing constructions [15, 48, 75, 81, 92], the security of which is based on generic collision-resistant hashing. Let $\mathcal{HBD} = \{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$ be such a scheme, described below in Corollary 6.1. All these schemes, are however, non-optimal: When the output range is of size $w = o(\log n)$, the output proof complexity as long as the verification complexity are both $\Omega(\log n)$, not satisfying in this way the definition of optimality (see Definition 2.8). Nevertheless, for reasons that will become clear later, we are using such an authenticated dictionary in our construction:

Corollary 6.1 *Let k be the security parameter. Then there exists a non-optimal, publicly-verifiable authenticated data structure scheme $\mathcal{HBD} = \{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$ for a data structure scheme defined for a dynamic dictionary D storing n elements such that:*

1. *It is correct according to Definition 2.4 and secure according to Definition 2.5 and assuming the existence of generic collision-resistant hash functions;*
2. *The access complexity of $\mathsf{setup}()$ is $O(n)$, outputting an authenticated data structure $\mathsf{auth}(D)$ of $O(n)$ group complexity;*
3. *The access complexity of $\mathsf{update}()$ is $O(\log n)$, outputting update information upd of $O(1)$ group complexity;*
4. *The access complexity of $\mathsf{refresh}()$ is $O(\log n)$;*
5. *For a range search query q outputting an answer of size w we have:*
   (a) *The access complexity of $\mathsf{query}()$ is $O(\log n + w)$;*
   (b) *The access complexity of $\mathsf{verify}()$ is $O(\log n + w)$;*

(c) The group complexity of the proof $\Pi(q)$ *is* $O(\log n + w)$.

We continue with describing the cryptographic primitive to be used in our construction, the *multilinear form*:

6.2 Multilinear forms

Let $\mathbb{G}$, $\mathcal{G}$ be two cyclic groups of prime order p and let g be a generator of $\mathbb{G}$. We let the bit-size of p (the order of both $\mathbb{G}$ and $\mathcal{G}$) to be a polynomial in the security parameter k. We are now ready to define an admissible t-multilinear form. The definition is similar to the one presented in the original paper by Silverberg and Boneh [19]:

Definition 6.1 *We say that a map* $e : \mathbb{G}^t \to \mathcal{G}$ *is an admissible* t*-multilinear form if it satisfies the following properties:*

1. $\mathbb{G}$ *and* $\mathcal{G}$ *are cyclic groups of the same prime order* p*;*
2. *The discrete logarithm problem is hard both in* $\mathbb{G}$ *and* $\mathcal{G}$*;*
3. *For all* $a_1, a_2, \ldots, a_t \in \mathbb{Z}_p^*$ *and* $x_1, x_2, \ldots, x_t \in \mathbb{G}$ *it is*

$$e(x_1^{a_1}, x_2^{a_2}, \ldots, x_t^{a_t}) = e(x_1, x_2, \ldots, x_t)^{a_1 a_2 \ldots a_t} \in \mathcal{G};$$

4. *The map is non-degenerate: If* $g \in \mathbb{G}$ *generates* $\mathbb{G}$ *then* $e(g, g, \ldots, g) \in \mathcal{G}$ *generates* $\mathcal{G}$.

We call the groups $\mathbb{G}$ and $\mathcal{G}$ for which there exists an admissible t-multilinear form *admissible* t*-multilinear groups.*

Definition 6.2 *An* admissible t-multilinear form generator *is a probabilistic polynomial-time algorithm that takes as input a natural number* t *and the security parameter* 1^k *and outputs a uniformly random tuple of multilinear pairing parameters* $(p, \mathbb{G}, \mathcal{G}, e, g)$*, where* $\mathbb{G}$ *and* $\mathcal{G}$ *are admissible* t*-multilinear groups for which there exists an admissible* t*-multilinear form* $e(., \ldots, .) : \mathbb{G}^t \to \mathcal{G}$ *of* t *inputs.*

To prove security of our construction, we are going to use the following assumption, which can be described as the "multi" version of the bilinear q-strong Diffie-Hellman assumption (see Assumption 3.2):

Assumption 6.1 (Multilinear q-strong Diffie-Hellman assumption) *Let k be the security parameter and let $(p, \mathbb{G}, \mathcal{G}, e, g)$ be a uniformly randomly generated tuple of multilinear pairings parameters, output by an admissible t-multilinear form generator. Given the elements $g, g^s, \ldots, g^{s^q} \in \mathbb{G}$ for some s chosen at random from $\mathbb{Z}_p^*$, where $q = \mathsf{poly}(k)$, there is no polynomial-time algorithm that can output the pair $(a, e(g, g, \ldots, g)^{1/(s+a)}) \in \mathbb{Z}_p \times \mathcal{G}$ except with negligible probability $\mathsf{neg}(k)$.*

6.3 An optimal authenticated dictionary

In this section, we describe an authenticated dictionary data structure scheme based on admissible multilinear form generators that achieves communication and verification complexity that is proportional to the size of the reported output range w. More importantly, these complexities are combined with *logarithmic* update and query costs.

Let $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ be a set of elements from a totally-ordered universe $\mathcal{U}$ contained in the dictionary, where $x_1 < x_2 < \ldots < x_n$. Each element is represented with $k/2$ bits. The actual set we are going to store, in order to also support efficient range search and non-membership queries, is the set of k-bit intervals, i.e., the set $\mathcal{A} = \{a_0, a_1, a_2, \ldots, a_n\}$, where, for $i = 1, \ldots, n-1$, it is $a_i = x_i || x_{i+1}$, for $i = 0$, it is $a_0 = -\infty || x_1$, and, for $i = n$, it is $a_n = x_n || + \infty$. Note that the total order on $\mathcal{X}$ imposes a natural total order on $\mathcal{A}$. In our construction we use a red-black tree, with data at the leaves (i.e., internal nodes data navigates the searches and does not correspond to actual data) [29].

We now describe $\mathcal{MFD} = \{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$, our authenticated data structure scheme for a dictionary data structure on the totally-ordered set

$\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$. We note that the construction uses several features from the accumulators constructions in Chapter 3, as long as the authenticated data structure scheme for a dictionary $\mathcal{HBD}$ from Corollary 6.1. Again, the actual set on which we are going to build our data structure is the set of intervals $\mathcal{A} = \{a_0, a_1, a_2, \ldots, a_n\}$.

Algorithm $\{\mathsf{sk}, \mathsf{pk}\} \leftarrow \mathsf{genkey}(1^k)$**:** Using a k-admissible multilinear form generator from Definition 6.2, the algorithm outputs a k-bit prime p, k-admissible multilinear groups $\mathbb{G}$ and $\mathcal{G}$, $g \in \mathbb{G}$ that generates $\mathbb{G}$ and a k-admissible multilinear form $e(., \ldots, .) : \mathbb{G}^k \rightarrow \mathcal{G}$[2]. Then it randomly picks a number $s \in \mathbb{Z}_p^*$ (s is the trapdoor). An upper bound q of the total number of elements to be stored in the data structure is decided and the algorithm also computes the elements of $\mathbb{G}$ $g^s, g^{s^2}, \ldots, g^{s^q}$. It also calls $\{\mathsf{sk}, \mathsf{pk}\} \leftarrow \mathcal{HBD}.\mathsf{genkey}(1^k)$. The algorithm outputs $s \in \mathbb{Z}_p^*$ and $\mathcal{HBD}.\mathsf{sk}$ as sk and everything else as pk.

Algorithm $\{\mathsf{auth}(D_0), d_0\} \leftarrow \mathsf{setup}(D_0, \mathsf{sk}, \mathsf{pk})$**:** Let $D_0 = \mathcal{A} = \{a_0, a_1, a_2, \ldots, a_n\}$ be the set of sorted intervals, that corresponds to the underlying set of elements $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$. Initially, the algorithm computes

$$\mathsf{acc}(\mathcal{A}) = g^{(a_0+s)(a_1+s)\ldots(a_n+s)} \in \mathbb{G}\,. \tag{6.1}$$

Let now T be the red-black tree built on top of the intervals a_i for $i = 0, \ldots, n$. Note that there is a natural notion of order imposed on $\mathcal{A}$, based on the order imposed on $\mathcal{X}$. Let $v_0, v_1, \ldots, v_n$ be the leaves of the tree, storing the intervals $a_1, a_1, \ldots, a_n$ respectively. We define the *label* of v_i as

$$\mathsf{label}(v_i) \quad = \quad g^{a_i+s} \in \mathbb{G} \text{ for all } i = 0, \ldots, n\,. \tag{6.2}$$

Also, let v_A be the internal node of T that is the root of the subtree T_A of T that contains the elements of some set $A \subseteq \mathcal{A}$. For every internal node v_A (and for the root of the tree as

[2]Note that the number of inputs of the multilinear form is equal to the security parameter k. This is because our construction will require $O(\log n)$ inputs for the multilinear form, and, since we are in the computational model, it is always $\log n < k$.

well), the algorithm sets

$$\mathsf{label}(v_A) \;=\; g^{\prod_{a\in A}(a+s)} \in \mathbb{G}\,. \tag{6.3}$$

All the labels $\mathsf{label}()$ are stored with tree T. Subsequently, the algorithm calls the algorithm $\mathcal{HBD}.\mathsf{setup}(\mathcal{A}, \mathsf{sk}, \mathsf{pk})$, building in this way a new authenticated dictionary based on hashing on top of $\mathcal{A}$. It sets $d_0 = \{\mathsf{acc}(\mathcal{A}), \mathsf{hash}(\mathcal{A})\}$, where $\mathsf{hash}(\mathcal{A}) = \mathcal{HBD}.d_0$. The structure $\mathsf{auth}(D_0)$ contains the tree T, as well as the authenticated structure $\mathcal{HBD}.\mathsf{auth}(\mathcal{A})$. We now make the following important remark:

Remark 6.1 *The hashing scheme employed by $\mathcal{HBD}$ includes in the hashing computation all the labels $\mathsf{label}(v)$ (defined in Relation 6.3) of the internal nodes v. Namely the hash value h_v at some internal node v that has left child u and right child w is computed as*

$$h_v = h(h_u||v||\mathsf{label}(v)||h_w)\,,$$

where $h(.)$ is the used generic collision-resistant hash function (e.g., SHA-2).

Lemma 6.1 *Algorithm $\mathsf{setup}()$ of the authenticated data structure scheme $\mathcal{MFD}$ has $O(n)$ access complexity. Moreover, the authenticated data structure $\mathsf{auth}(D_0)$ output by $\mathsf{setup}()$ has $O(n)$ group complexity.*

Proof: First of all, $\mathcal{HBD}.\mathsf{setup}()$ has $O(n)$ complexity, outputting an authenticated data structure of $O(n)$ group complexity, by Corollary 6.1. Denote now with v_A an internal node of tree T, which is the root of a subtree T_A. For each leaf of the tree v_i the algorithm computes $P_i = a_i + s$ and then sets $\mathsf{label}(v_i) = g^{P_i}$. Note now that for every other internal node of the tree v_A with left child v_B and right child v_C, it is $P_A = P_B P_C$ and $\mathsf{label}(v_A) = g^{P_A}$, since $A = B \cup C$. The described recursive computation has $O(n)$ complexity (a postorder traversal of T). Moreover, since for each node of T, we are storing one label, the total group complexity of the labels is $O(n)$. This completes the proof. $\square$

Algorithm $\{D_{h+1}, \mathsf{auth}(D_{h+1}), d_{h+1}, \mathsf{upd}\} \leftarrow \mathsf{update}(u, D_h, \mathsf{auth}(D_h), d_h, \mathsf{sk}, \mathsf{pk})$**:** We distinguish two cases:

(1) *Insertion of element* $x \in \mathcal{X}$: Let $a_1 = u||z \in \mathcal{A}$ be an interval stored in the dictionary such that $u < x < z$. Then the insertion of x is equivalent with deleting the interval a_1 and inserting the intervals $b_1 = u||x$ and $b_2 = x||z$;

(2) *Deletion of element* $x \in \mathcal{X}$: Let $a_1 = u||x \in \mathcal{A}$ and $a_2 = x||z \in \mathcal{A}$ be successive intervals stored in the dictionary. Then the deletion of x is equivalent with deleting the intervals a_1 and a_2 and inserting the interval $b = u||z$.

We have therefore reduced the update of *elements* in $\mathcal{X}$ to a constant number of updates of *intervals* in $\mathcal{A}$. Thus we continue the description of the update algorithms with reference to intervals. Let a be the interval of the update u. Interval a defines a logarithmic number of nodes in T that needs to be accessed and modified in order for the update to be performed (this follows from red-black tree properties). Let $p(a)$ be the set of those nodes. For every node $v \in p(a)$, the algorithm updates the labels $\mathsf{label}(v)$ and outputs the updated labels as information upd. The algorithm also stores the new (updated) labels on tree T, which is updated to T'. Finally, the algorithm calls $\{\mathcal{A}', \mathsf{auth}(\mathcal{A}'), d'\} \leftarrow \mathcal{HBD}.\mathsf{update}(u, \mathcal{A}, \mathsf{auth}(\mathcal{A}), d_h, \mathsf{sk}, \mathsf{pk})$ and outputs the following structures:

1. The new digest d_{h+1}, which contains $\mathsf{acc}(\mathcal{A}') = \mathsf{acc}(\mathcal{A})^{a+s}$ (in the case of a deletion it is $\mathsf{acc}(\mathcal{A}') = \mathsf{acc}(\mathcal{A})^{(a+s)^{-1}}$) and the new digest $\mathsf{hash}(\mathcal{A}') = \mathcal{HBD}.d'$, as output by calling $\mathcal{HBD}.\mathsf{update}()$;

2. The new authenticated data structure $\mathsf{auth}(D_{h+1})$, which contains T' and $\mathsf{auth}(\mathcal{A}')$;

3. Information upd, which contains the updated labels.

Lemma 6.2 *Algorithm* $\mathsf{update}()$ *of the authenticated data structure scheme* $\mathcal{MFD}$ *has* $O(\log n)$ *access complexity. Moreover, the update information* upd *output by* $\mathsf{update}()$ *has* $O(\log n)$ *group complexity.*

Proof: This result follows from the properties of the red-black tree [29]: There is only a logarithmic number of nodes that changes during a red-black tree update. Moreover, the label $\mathsf{label}(v)$ of each such node v can be updated with $O(1)$ complexity since $\mathsf{update}()$ has access to the secret key s. Finally, by Corollary 6.1, $\mathcal{HBD}.\mathsf{update}()$ has $O(\log n)$ access complexity. This makes the total update complexity equal to $O(\log n)$. □

Algorithm $\{D_{h+1}, \mathsf{auth}(D_{h+1}), d_{h+1}\} \leftarrow \mathsf{refresh}(u, D_h, \mathsf{auth}(D_h), d_h, \mathsf{upd}, \mathsf{pk})$**:** The algorithm updates T to T' by using the information contained in upd[3]. Finally, the algorithm calls $\{\mathcal{A}', \mathsf{auth}(\mathcal{A}'), d'\} \leftarrow \mathcal{HBD}.\mathsf{refresh}(u, \mathcal{A}, \mathsf{auth}(\mathcal{A}), d_h, \mathsf{sk}, \mathsf{pk})$ and outputs the following structures:

1. The new digest d_{h+1}, which contains $\mathsf{acc}(\mathcal{A}')$ (contained in upd) and the new digest $\mathsf{hash}(\mathcal{A}') = \mathcal{HBD}.d'$, as output by calling $\mathcal{HBD}.\mathsf{refresh}()$;
2. The new authenticated data structure $\mathsf{auth}(D_{h+1})$, which contains T' and $\mathsf{auth}(\mathcal{A}')$.

Lemma 6.3 *Algorithm* $\mathsf{refresh}()$ *of the authenticated data structure scheme* $\mathcal{MFD}$ *has* $O(\log n)$ *access complexity.*

Proof: The algorithm performs a computation proportional to the size of the information upd. Therefore, by Lemma 6.2, this part has $O(\log n)$ access complexity. Moreover, by Corollary 6.1, $\mathcal{HBD}.\mathsf{refresh}()$ has $O(\log n)$ access complexity. Summing up, the total access complexity of the algorithm is $O(\log n)$. □

6.3.1 Dictionary queries and verification

In this section we show the construction of proofs for the dictionary queries using the authenticated data structure scheme $\mathcal{MFD}$. As it was defined in Section 6.1, a dictionary range search query is described by two arguments, namely $a, b \in \mathcal{U}$, with $a \leq b$. Moreover,

[3] Algorithm $\mathsf{refresh}$ could perform this task without access to information upd. However, since the algorithm does not have access to the secret key sk, this would require linear complexity.

the answer to the query is the sorted list of w successive elements $\mathbf{y} = [y_1\ y_2 \ldots y_{w-1}\ y_w] \subseteq \mathcal{X}$ such that $y_1 \leq a \leq y_2 \leq \ldots \leq y_{w-1} \leq b < y_w$. For reasons to be made clear later, we will distinguish two cases:

- If $w = \Omega(\log n)$, the proof is constructed by using the authenticated data structure scheme $\mathcal{HBD}$. In this case, the size of the answer is $\Omega(\log n)$, and therefore the logarithmic-sized proofs of $\mathcal{HBD}$ (see Corollary 6.1) achieve optimality, according to Definition 2.8;

- If $w = o(\log n)$, the proof needs to be constructed in a different way, so that optimality can be achieved. Specifically, and as we will see later, optimality will be achieved only for $w = O(1)$. This is where the multilinear forms need to be employed.

We continue with describing algorithm $\mathsf{query}()$ formally.

Algorithm $\{\Pi(q), \alpha(q)\} \leftarrow \mathsf{query}(q, D_h, \mathsf{auth}(D_h), \mathsf{pk})$**:** Let the query q be two elements $a, b \in \mathcal{U}$, with $a \leq b$. Suppose the answer to the query is the sorted list of w successive elements $\mathbf{y} = [y_1\ y_2 \ldots y_{w-1}\ y_w] \subseteq \mathcal{X}$ such that $y_1 \leq a \leq y_2 \leq \ldots \leq y_{w-1} \leq b < y_w$.

If $w = \Omega(\log n)$, let $\mathcal{HBD}.\Pi(q)$ and $\mathcal{HBD}.\alpha(q)$ be the proof and the answer respectively, output by calling $\mathcal{HBD}.\mathsf{query}(q, D_h, \mathsf{auth}(D_h), \mathsf{pk})$, where q contains intervals $y_1||y_2$ and $y_{w-1}||y_w$ (note that $y_1||y_2 \leq y_{w-1}||y_w$). Then we set $\Pi(q) = \mathcal{HBD}.\Pi(q)$. Namely in this case, the proof is constructed by using the authenticated data structure scheme $\mathcal{HBD}$.

Let us now examine the most interesting case, where $w = o(\log n)$. In this case the proof $\Pi(q)$ consists of $w - 1$ group elements in $\mathcal{G}$, namely the witnesses

$$\mathcal{W}_{a_i} = e(g, g, \ldots, g)^{\prod_{a \in \mathcal{A} - a_i} (a+s)} \in \mathcal{G}\,, \tag{6.4}$$

where $a_i = y_i||y_{i+1}$, for $i = 1, \ldots w-1$. Note that $\mathcal{W}_{a_i}$ is similar to the membership witness for accumulators, described in Section 3.

Lemma 6.4 *Algorithm* query() *of the authenticated data structure scheme* $\mathcal{MFD}$ *has* $O(\log n + w)$ *access complexity when* $w = \Omega(\log n)$ *and* $O(w \log n)$ *access complexity when* $w = o(\log n)$. *Moreover, in both cases, it outputs a proof* $\Pi(q)$ *of* $O(w)$ *group complexity.*

Proof: If $w = \Omega(\log n)$, then the authenticated data structure scheme $\mathcal{HBD}$, by Corollary 6.1, outputs proofs of $O(\log n + w)$ group complexity with $O(\log n + w)$ access complexity. However, since $w = \Omega(\log n)$, it is $O(\log n + w) = O(w)$.

For the case $w = o(\log n)$, note that each witness $\mathcal{W}_{a_i}$ in Relation 6.4 can be constructed with $O(\log n)$ access complexity: Let $v_{i0}, v_{i1}, \ldots, v_{il}$ be the path in tree T from the leaf node storing the interval a_i to the root of tree T, v_{il}, where $l = O(\log n)$. Let also $w_{i0}, w_{i1}, \ldots, w_{i(l-1)}$ be the *sibling nodes* of nodes $v_{i0}, v_{i1}, \ldots, v_{i(l-1)}$ respectively (note that w_{i0} might not exist). By the construction of tree T (it can be viewed as *segment tree* [97]), for each $j = 0, 1, \ldots, l-1$, it is

$$\mathsf{label}(w_{ij}) = g^{P_{ij}}\,, \tag{6.5}$$

where

$$\prod_{j=0}^{l-1} P_{ij} = \prod_{a \in \mathcal{A} - a_i} (a + s)\,. \tag{6.6}$$

This means that information about the whole set can be retrieved by accessing $O(\log n)$ memory locations. Therefore, the algorithm constructs the witness $\mathcal{W}_{a_i}$ by computing

$$\begin{aligned} e\left(\mathsf{label}(w_{i0}), \mathsf{label}(w_{i1}), \ldots, \mathsf{label}(w_{i(l-1)}), g, \ldots, g\right) &= e\left(g^{P_{i0}}, g^{P_{i1}}, \ldots, g^{P_{i(l-1)}}, g, \ldots, g\right) \\ &= e(g, g, \ldots, g)^{\prod_{j=0}^{l-1} P_{ij}} \\ &= e(g, g, \ldots, g)^{\prod_{a \in \mathcal{A} - a_i} (a+s)} \\ &= \mathcal{W}_{a_i}\,. \end{aligned}$$

The above four equalities follow from Relation 6.5, the properties of the multilinear form $e(., \ldots, .)$ and Relations 6.6 and 6.4 respectively. Since computing one such witness $\mathcal{W}_{a_i}$ requires $O(\log n)$ inputs from the authenticated data structure, and since the proof construction requires the computation of $w - 1$ such witnesses we conclude that for the case

$w = o(\log n)$, computing the proof has $O(w \log n)$ access complexity. Finally, since the proof contains $w-1$ such witnesses (which are elements in $\mathcal{G}$), we conclude that the group complexity of the proof in the case $w = o(\log n)$ is also $O(w)$. $\square$

We now formally describe the verification algorithm. The verification algorithm will take as input a proof and an answer and will either accept or reject the answer. Note that the verification algorithm needs to have access to the secret key sk. Therefore the authenticated data structure scheme $\mathcal{MFD}$ *is not* publicly verifiable.

Algorithm $\{\mathsf{accept}, \mathsf{reject}\} \leftarrow \mathsf{verify}(q, \alpha, \Pi, d_h, \mathsf{sk}, \mathsf{pk})$**:** Let the query q be two elements $a, b \in \mathcal{U}$, with $a \leq b$. Suppose the input answer α is the sorted list of w successive elements $\mathbf{y} = [y_1\ y_2 \ldots y_{w-1}\ y_w]$ such that $y_1 \leq a \leq y_2 \leq \ldots \leq y_{w-1} \leq b < y_w$. If $w = \Omega(\log n)$ the algorithm verifies the answer by running algorithm $\mathcal{HBD}.\mathsf{verify}(q, \alpha, \Pi, d_h, \mathsf{sk}, \mathsf{pk})$.

Otherwise, i.e., if $w = o(\log n)$, the input proof Π is the list of $w-1$ witnesses

$$\mathsf{W}_1, \mathsf{W}_2, \ldots, \mathsf{W}_{w-1} \in \mathcal{G}\,.$$

The algorithm outputs accept if all of the following relations are true:

$$\mathsf{W}_i^{(s+a_i)} = e(d, g, \ldots, g) \text{ for all } i = 1, \ldots, w-1\,, \tag{6.7}$$

where $a_i = y_i || y_{i+1}$ and $d = \mathsf{acc}(\mathcal{A})$ is contained in digest d_h.

Lemma 6.5 *Algorithm* $\mathsf{verify}()$ *of the authenticated data structure scheme* $\mathcal{MFD}$ *has* $O(w)$ *access complexity.*

Proof: For $w = \Omega(\log n)$ the complexity is due to the authenticated data structure scheme $\mathcal{HBD}$ and the proof follows the same argument as in Lemma 6.4. For $w = o(\log n)$, the complexity is $O(w)$ since the algorithm needs to check Relations 6.7. Checking one such relation requires one exponentiation in $\mathcal{G}$ and since $w-1$ such relations need to be checked, the result follows. Note that the required exponentiations in Relations 6.7 can be performed by the algorithm because the algorithm has access to the secret key s. $\square$

Lemma 6.6 *The authenticated data structure scheme* $\mathcal{MFD} = \{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$ *is correct according to Definition 2.4.*

Proof: Let D_0 be any dictionary storing the collection of intervals $\mathcal{A}$, corresponding to an elements collection $\mathcal{X}$ (of n elements) from a totally-ordered universe $\mathcal{U}$. Fix the security parameter k and output $\mathsf{pk} = \{\mathbb{G}, \mathcal{G}, e(.,\ldots,.), p\}$ and $\mathsf{sk} = s \in \mathbb{Z}_p^*$ by calling algorithm $\mathsf{genkey}()$. Then output an authenticated data structure $\mathsf{auth}(D_0)$ and the respective digest d_0, by calling algorithm $\mathsf{setup}()$. Pick a polynomial number of updates—namely, pick a polynomial number of elements from $\mathcal{U}$ for insertion or deletion—and update $\mathsf{auth}(D_0)$ and d_0 by calling algorithm $\mathsf{refresh}()$. Let D_h be the final dictionary, $\mathsf{auth}(D_h)$ be the produced authenticated data structure and d_h be the final digest. Let now q be a range search query corresponding to elements a and b from $\mathcal{U}$ with $a \leq b$. Algorithm query outputs an answer $\alpha(q)$ which is the sorted list of w successive elements $\mathbf{y} = [y_1\ y_2 \ldots y_{w-1}\ y_w] \subseteq \mathcal{X}$ (note that $[y_1||y_2\ y_2||y_3 \ldots y_{w-1}||y_w] \subseteq \mathcal{A}$) such that $y_1 \leq a \leq y_2 \leq \ldots \leq y_{w-1} \leq b < y_w$. Let also $\Pi(q)$ be the proof output by algorithm $\mathsf{query}()$. We distinguish two cases:

1. If $w = \Omega(\log n)$, the proof $\Pi(q)$ is computed by algorithm $\mathcal{HBD}.\mathsf{query}()$. By the correctness of the authenticated data structure scheme $\mathcal{HBD}$, $\mathsf{verify}()$ does not reject in this case;

2. If $w = o(\log n)$, the proof $\Pi(q)$ consists of the witnesses $\mathcal{W}_{a_i}$ for $i = 1, \ldots, w-1$, and where $a_i = y_i||y_{i+1}$. Algorithm $\mathsf{verify}()$ does not reject since

$$\begin{aligned} \mathcal{W}_{a_i}^{a_i+s} &= \left(e(g,g,\ldots,g)^{\prod_{a\in\mathcal{A}-a_i}(a+s)}\right)^{(a_i+s)} \\ &= e(g,g,\ldots,g)^{\prod_{a\in\mathcal{A}}(a+s)} \\ &= e(\mathsf{acc}(\mathcal{A}),g,\ldots,g)\,, \end{aligned}$$

   by the definition of $\mathcal{W}_{a_i}$ in Relation 6.4 and since $\mathsf{acc}(\mathcal{A})$ is always maintained to be the accumulation of all the intervals in $\mathcal{A}$ through algorithm $\mathsf{refresh}()$—see Relation 6.1.

This completes the proof. □

Lemma 6.7 *The authenticated data structure scheme* $\mathcal{MFD} = \{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$ *is secure according to Definition 2.5.*

Proof: Fix the security parameter k and output $\mathsf{pk} = \{\mathbb{G}, \mathcal{G}, e(.,\ldots,.), p\}$ and $\mathsf{sk} = s \in \mathbb{Z}_p^*$ by calling algorithm $\mathsf{genkey}()$. Let Adv be a polynomially-bounded adversary. Adv picks an initial collection of n elements $\mathcal{X}$, all belonging to a totally-ordered universe $\mathcal{U}$. Let $\mathcal{A}$ be the respective collection of intervals, stored in a dictionary D_0. Adv outputs an authenticated data structure $\mathsf{auth}(D_0)$, by calling algorithm $\mathsf{setup}()$ through oracle access. Then Adv picks a polynomial number of updates—namely, he picks a polynomial number of elements from $\mathcal{U}$ for insertion or deletion. Let D_h be the final dictionary after the updates, let the updated final collection of intervals and elements be $\mathcal{A}$ and $\mathcal{X}$ respectively, and let d_h be the final digest as produced by the adversary through oracle access to algorithm $\mathsf{update}()$. Let q be a dictionary query picked by the adversary, consisting of two elements $a, b \in \mathcal{U}$, with $a \leq b$. Suppose the adversary outputs an *incorrect* answer α which is however the sorted list of w successive elements $[z_1\ z_2 \ldots z_{w-1}\ z_w]$ such that $z_1 \leq a \leq z_2 \leq \ldots \leq z_{w-1} \leq b < z_w$ and a respective proof Π. We will compute the probability that $\mathbf{check}(q, \alpha, D_h)$ rejects, while $\mathsf{verify}(q, \alpha, \Pi, \mathsf{sk}, \mathsf{pk})$ accepts, as required by Definition 2.5. If $w = \Omega(\log n)$, by the security of the scheme $\mathcal{HBD}$ the event in question happens with probability $\mathsf{neg}(k)$. If $w = o(\log n)$, the proof Π consists of $w-1$ witnesses $\mathsf{W}_1, \mathsf{W}_2, \ldots, \mathsf{W}_{w-1}$, each one referring to the intervals $b_1, b_2, \ldots, b_{w-1}$, where $b_i = z_i || z_{i+1}$, respectively. Since answer α is not correct, it should be the case that there exists $b_i \notin \mathcal{A}$[4] such that

$$\mathsf{W}_i^{b_i+s} = e(g, g, \ldots, g)^{(a_0+s)(a_1+s)(a_2+s)\ldots(a_n+s)},$$

where $\mathcal{A} = \{a_0, a_1, \ldots, a_n\}$. Since $b_i \notin \{a_0, a_1, a_2, \ldots, a_n\}$ we can write

$$(a_0+s)(a_1+s)\ldots(a_n+s) = \mathcal{P}(b_i+s) + \lambda,$$

where the coefficients of polynomial $\mathcal{P}$ and quantity λ are computable in polynomial time in

[4]Note that $b_i \notin \mathcal{A}$ is equivalent to either adding extra elements in the reported range or omitting certain elements from the reported range.

n (polynomial division). Therefore the adversary Adv can compute

$$e(g,g,\ldots,g)^{\frac{1}{b_i+s}} = [\mathsf{W}_i e(g,g,\ldots,g)^{-\mathcal{P}}]^{\lambda^{-1}},$$

since $e(g,g,\ldots,g)^{s^i} \in \mathcal{G}$ can efficiently be computed from $g^{s^i} \in \mathbb{G}$ by using the admissible multilinear form $e : \mathbb{G}^k \to \mathcal{G}$ for all $i = 0,\ldots,q$. However, by Assumption 6.1, this happens with probability $\mathsf{neg}(k)$. $\square$

We note here that the second part of the proof of security above (the case $w = o(\log n)$) follows the same logic of the security of the bilinear-map accumulator in Lemma 5.1. However, in a multilinear setting, where $e(.,\ldots,.)$ is not used for verification, if we are to use only Assumption 6.1 we *cannot* prove security for *subsets* of elements, but only for *one* element. Proving security for subsets of elements in the multilinear setting would require a stronger assumption. Moreover, due to this limitation, we will only be able to prove optimality of the presented authenticated data structure scheme $\mathcal{MFD}$ for specific values of w, i.e., for $w = O(1)$ or $w = \Omega(\log n)$.

Lemma 6.8 *For range search queries outputting an answer of size w such that $w = O(1)$ or $w = \Omega(\log n)$, the authenticated data structure scheme $\mathcal{MFD} = \{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$ is optimal according to Definition 2.8.*

Proof: According to Definition 2.8, the authenticated data structure scheme $\mathcal{MFD} = \{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$ for a dictionary D of n elements is optimal as long as $w = O(1)$ or $w = \Omega(\log n)$. This is because all of the following are true:

1. The authenticated data structure scheme $\mathcal{MFD}$ is correct and secure, by Lemmata 6.6 and 6.7 respectively;

2. For the group complexity of the authenticated data structure we have $|\mathsf{auth}(D)| = |D| = O(n)$, by Lemma 6.1;

3. By Lemmata 6.2 and 6.3, and for the update access complexity we have $|\mathsf{update}()| + |\mathsf{upd}| + |\mathsf{refresh}()| = O(|\mathbf{update}()|) = O(\log n)$;

4. For the query access complexity, we distinguish two cases:

   - When $w = O(1)$ or $w = \Omega(\log n)$, by Lemma 6.4, we have

     $$|\mathsf{query}()| = O(|\mathbf{query}()|) = O(\log n + w)\,.$$

     So in this case optimality is achieved;

   - When $w = \omega(1)$ and $w = o(\log n)$, by Lemma 6.4, we have $|\mathsf{query}()| = O(w \log n)$ and $|\mathbf{query}()| = O(\log n + w)$. Therefore $|\mathsf{query}()|$ is *not* $O(|\mathbf{query}()|)$ which means that the query complexity constraint is not satisfied in this case.

5. For the group complexity of the proof we have $|\Pi(q)| = O(|q| + |\alpha(q)|) = O(w)$, by Lemma 6.4;

6. For the access complexity of the verification algorithm we have $|\mathsf{verify}()| = O(|q| + |\alpha(q)|) = O(w)$.

Therefore the authenticated data structure scheme $\mathcal{MFD}$ is optimal for range search queries returning an answer of size w such that $w = O(1)$ or $w = \Omega(\log n)$. □

6.3.2 Main results

Theorem 6.1 *Let k be the security parameter and assume the existence of an admissible $\Theta(k)$-multilinear form generator, as defined in Definition 6.2. Then there exists an authenticated data structure scheme $\mathcal{MFD} = \{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$ for a data structure scheme defined for a dynamic dictionary D storing n elements such that:*

1. *It is correct according to Definition 2.4 and secure according to Definition 2.5 and (i) under the multilinear q-strong Diffie-Hellman assumption; (ii) assuming the existence of generic collision-resistant hash functions;*

2. *It is optimal only for range search queries outputting an answer of size w such that $w = O(1)$ or $w = \Omega(\log n)$, according to Definition 2.8;*

3. *It is not publicly-verifiable according to Definition 2.9;*

4. *The access complexity of* $\mathsf{setup}()$ *is* $O(n)$*, outputting an authenticated data structure* $\mathsf{auth}(D)$ *of* $O(n)$ *group complexity;*

5. *The access complexity of* $\mathsf{update}()$ *is* $O(\log n)$*, outputting update information* upd *of* $O(\log n)$ *group complexity;*

6. *The access complexity of* $\mathsf{refresh}()$ *is* $O(\log n)$*;*

7. *For a range search query* q *outputting an answer of size* w *we have:*

   (a) *The access complexity of* $\mathsf{query}()$ *is* $O(\log n + w)$ *when* $w = \Omega(\log n)$ *and* $O(w \log n)$ *when* $w = o(\log n)$*;*

   (b) *The access complexity of* $\mathsf{verify}()$ *is* $O(w)$*;*

   (c) *The group complexity of the proof* $\Pi(q)$ *is* $O(w)$*.*

Proof: This result follows directly from Lemmata 6.1, 6.2, 6.3, 6.4, 6.5, 6.6, 6.7 and 6.7. Note that the scheme is not publicly-verifiable since algorithm $\mathsf{verify}()$ requires the secret key as an input. Finally, we have to assume existence of generic collision-resistant hash functions since we are using the authenticated data structure scheme $\mathcal{HBD}$. □

We now present the final result of this chapter that relates optimality of an authenticated data structure scheme for a dictionary with the existence of admissible multilinear form generators.

Theorem 6.2 *Let* k *be the security parameter and let* $\mathcal{D}$ *be a data structure scheme for a dictionary of* n *elements that supports range search queries* q *outputting an answer of size* w *such that* $w = O(1)$ *or* $w = \Omega(\log n)$*. If no optimal authenticated data structure scheme for* $\mathcal{D}$ *exists, then no admissible* $\Theta(k)$*-multilinear form generator exists either.*

Proof: Let's assume this is not the case and an admissible $\Theta(k)$-multilinear form generator does exist in the absence of an optimal authenticated data structure scheme for $\mathcal{D}$. This is

a contradiction since we can use the construction of Theorem 6.1, which uses an admissible $\Theta(k)$-multilinear form generator, to derive an optimal authenticated data structure scheme for $\mathcal{D}$. $\square$

Finally we need to make the following important observation. Theorem 6.2 does not exclude the existence of some instance of a multilinear form, even in the absence of optimal authenticated dictionaries (say for example an instance of a multilinear form for three inputs). The result holds for all admissible $\Theta(k)$-multilinear forms, where k is the security parameter.

6.3.3 Application in the two-party protocol

Due to the fact that the authenticated data structure scheme $\mathcal{MFD}$ is *not* publicly-verifiable, it can only be used by a two-party protocol, since a three-party protocol always requires a publicly-verifiable authenticated data structure scheme (see Protocol 2.1). However, in order to be able to use the authenticated data structure scheme $\mathcal{MFD}$ of Theorem 6.1 in a black-box way with Theorem 2.2—and derive a two-party authenticated data structures protocol, we have to ensure that Assumption 2.1 holds for the authenticated data structure scheme $\mathcal{MFD}$:

Lemma 6.9 *Assumption 2.1 is true for the authenticated data structure scheme $\mathcal{MFD}$. Moreover, for every update u, $|Q_u|$ has $O(1)$ complexity.*

Proof: Let an update u refer to element e, i.e., either *insert* element e to the dictionary or *delete* element e from the dictionary. The respective set of queries Q_u required for Assumption 2.1 *simply* contains one query q for the range $[e, e']$ such that there are $w = \Omega(\log n)$ elements between e and e'. Let $\{\Pi(q), \alpha(q)\} \leftarrow \mathsf{query}(q, D_h, \mathsf{auth}(D_h), \mathsf{pk})$. Since $w = \Omega(\log n)$, $\Pi(q)$ and $\alpha(q)$ are output by algorithm $\mathcal{HBD}.\mathsf{query}()$.

We now describe function $z(.)$ from Assumption 2.1. Function $z(.)$ extracts $\delta_u(D_h)$ and $\delta_u(\mathsf{auth}(D_h))$ from $\Pi(q)$: Due to the hashing scheme employed in Remark 6.1, $\Pi(q)$ contains all the structure of the red-black tree $\delta_u(D_h)$ that is accessed during update u, along

with the labels $\mathsf{label}(.)$ (i.e., the ones that need to be updated by $\mathsf{update}()$), that belong to the accessed authenticated data structure $\delta_u(\mathsf{auth}(D_h))$. Extracting that information has $O(\log n)$ complexity, equal to the verification complexity, as required by Assumption 2.1. This completes the proof. □

By Theorems 2.2 and 6.1 and Lemma 6.9, we can now state the final result for the two-party model:

Corollary 6.2 *Let k be the security parameter and assume (i) the existence of an admissible $\Theta(k)$-multilinear form generator; (ii) that the multilinear q-strong Diffie-Hellman assumption holds; (iii) the existence of generic collision-resistant hash functions. Then there exists a two-party authenticated data structures protocol (see Protocol 2.2) for verifying range search queries q on a dynamic dictionary storing n elements, and where w is the size of the answer to a range search query q, such that:*

1. *The protocol is interactive;*
2. *The setup at the client has $O(n)$ access complexity;*
3. *The update at the client has $O(\log n)$ access complexity;*
4. *The verification at the client has $O(w)$ access complexity;*
5. *The space needed at the client has $O(1)$ group complexity;*
6. *The communication between the client and the server has $O(\log n)$ group complexity during updates and $O(w)$ group complexity during queries;*
7. *The update at the server has $O(\log n)$ access complexity;*
8. *The query at the server has $O(\log n + w)$ access complexity when $w = \Omega(\log n)$ and $O(w \log n)$ access complexity when $w = o(\log n)$;*
9. *The space needed at the server has $O(n)$ group complexity;*

10. *For a query q sent by the client to the server at any time (even after updates), let α be an answer and let π be a proof returned by the server. With probability $\Omega(1 - \mathsf{neg}(k))$, the client accepts the answer α if and only if α is correct.*

6.4 Summary

In this chapter, we have presented the first *optimal* authenticated dictionary (Theorem 6.1) supporting range search queries outputting answers of size w such that $w = O(1)$ or $w = \Omega(\log n)$, having verification and proof complexity equal to $O(w)$ (as opposed to $O(\log n + w)$, see Table 6.1). Its design is based on multilinear forms, a recently-proposed cryptographic primitive [19] whose construction remains an open problem to date.

However, since multilinear forms are not known to exist yet, this work can be viewed from a different angle (Theorem 6.2): if one could prove that such optimal authenticated dictionaries cannot exist in the computational model, irrespectively of cryptographic primitives, then our result would imply that certain admissible multilinear form generators cannot exist as well (i.e., it can be viewed as a reduction). Thus, we provide an alternative avenue towards proving the nonexistence of multilinear form generators in the context of general lower bounds for authenticated data structures [106] and for memory checking [35].

Chapter 7

Conclusions

This thesis studies the problem of efficiently verifying data and computations stored and performed respectively by untrusted parties. This research direction, lying under the framework of *cloud cryptography*, has become very relevant nowadays, given the great amount of information and computation that is outsourced remotely at untrusted repositories, due to the increasing adoption of *cloud computing* in our everyday digital interactions. We therefore explore in depth the field of *authenticated data structures*, constructing a firm theoretical foundation in Chapter 2 and then continue, in the subsequent chapters, with the development of five different authenticated data structure schemes, each one constructed for a different problem. All our solutions are fully dynamic.

A common feature shared by all the authenticated data structures designed in this thesis is the use of advanced cryptography. A common goal of all the solutions has been how to exploit the offered cryptographic tools in order to derive highly-desirable efficiency features, such as constant communication complexity (Chapter 3), parallel algorithms (Chapter 4), operation sensitivity (Chapter 5) and optimality (Chapter 6), which could not be achieved otherwise, e.g., with the use of traditional hash-based techniques. We prove the security of our constructions only under well-accepted—by the cryptography community—computational assumptions (e.g., strong Diffie-Hellman and RSA problems and polynomial approximation of lattices problems).

Table 7.1: Asymptotic access and group complexities of the authenticated data structure schemes presented in this thesis, applied to the fundamental problem of verifying read/write operations on an array of n entries, and compared with the first result on *dynamic* authenticated data structures by Naor and Nissim [81]. We note that, since all complexities for the plain table data structure are constant, no authenticated data structure scheme presented is optimal. Moreover, based on the recent lower bound for memory checking by Dwork et al. [35], it seems unlikely that such a scheme could be derived.

	$\mathcal{HBD}$ [81]	$\mathcal{RHT}$ Chapter 3	$\mathcal{BHT}$ Chapter 3	$\mathcal{LBT}$ Chapter 4	$\mathcal{MFD}$ Chapter 6
setup()	n	n	n	$n \log n$	n
update()	$\log n$	1	1	1	$\log n$
refresh()	$\log n$	1	1	$\log n$	$\log n$
query()	$\log n$	n^ϵ	$n^\epsilon \log n$	$\log n$	$\log n$
verify()	$\log n$	1	1	$\log n$	1
proof $\Pi(q)$	$\log n$	1	1	$\log n$	1
info. upd	1	1	1	1	$\log n$
publicly verifiable	yes	yes	yes	yes	no
optimal	no	no	no	no	no
assumption	Generic CR	Strong RSA	Bilinear q-DH	GAPSVP	Multilinear q-DH and Generic CR

The findings of this thesis indicate that understanding and employing advanced cryptographic tools can lead to significant complexity gains in authenticated data structures and more generally in verifiable computations. Perhaps the most persuading justification for the validity of this statement is Chapter 5 itself, where non-trivial computations over outsourced data (set operations) are verified with optimal costs, due to the use of bilinear maps. Moreover this result provides evidence that more complicated functionalities (other than traditional set-membership computations) could be possibly verified efficiently in a public-key setting using authenticated data structures techniques, initiating in this fashion the quest for schemes that apply to other interesting problems (e.g., geometric computations).

7.1 Overview of thesis results and discussion

In this thesis, we observed that using different cryptography allows for various complexity trade-offs in authenticated data structures. In Table 7.1, we apply all our schemes (except for the scheme of Chapter 5) in the fundamental problem of verifying read/write operations on an array of n entries. This is a data structure where all our authenticated data structure schemes can be easily employed. Table 7.1 also includes a column referring to the seminal result by Naor and Nissim [81], where an authenticated dictionary was presented, based on 2-3 tree implementation, and with logarithmic complexities.

From the results in Table 7.1, we draw the following conclusion: As of now, there is *no* optimal authenticated data structure (as defined in Definition 2.8) for the simplest functionality of reading and writing entries on a table (similar to the memory checking model). We note however, that this does not come as a surprise: It would seem that deriving an optimal authenticated data structure scheme for a table (a RAM array) would violate existing $\Omega(\log n/\log\log n)$ bounds that have appeared in the memory checking model [35]. This observation naturally raises an open problem: Can we design an authenticated table of $\Theta(\log n/\log\log n)$ complexities? This construction would potentially yield an optimal online

memory checker and could be derived from the realm of more advanced cryptography.

7.2 Future work

It is our belief that providing security in the cloud is going to play a major role in adopting cloud computing as a new computing discipline. Concerning *cloud integrity*, future work includes a further investigation of the field of authenticated data structures. More specifically, one can focus on the verification of outsourced computations, in an operation-sensitive way. Operation-sensitivity, a crucial efficiency property, has only been achieved so far in a practical and publicly-verifiable fashion for specific computations, such as range search [52] and set operations (see Chapter 5). On the other hand, it has been shown that in a privately-verifiable way—and under certain assumptions, it is feasible for general computations, i.e., any boolean circuit, e.g., by using the model of outsourced verifiable computation [41]. However, these constructions are currently not very efficient. Aiming at publicly-verifiable solutions that could be used by cloud applications without changing the user experience, the question that arises is evident: Which outsourced computations (e.g., shortest paths) can be practically and publicly verified in an operation-sensitive way?

Another aspect of cloud security that can be investigated is *cloud privacy*, i.e., protecting the confidentiality of data that is stored remotely. Resorting to a solution that merely encrypts our data before uploading it online defeats one of the main purposes of investing into cloud infrastructures: No advanced meaningful outsourced computations can be performed on encrypted data. Achieving both goals, namely storing encrypted data and at the same time being able to do significant processing with it, was recently achieved with the proposal of a fully-homomorphic encryption scheme [43]. Implementing however such a primitive has not led to efficient solutions yet—it has on the other hand ignited a lot of enthusiasm for cloud privacy research. Our belief is that we have to settle for simpler and more efficient constructions that refer to specific functionalities, e.g., see the work on *searchable* symmetric

encryption by Curtmola et al. [31]. Therefore, future directions could explore the computation of such specific functionalities (e.g., geometric queries, polynomial evaluation) on private data in an efficient way which will allow easy implementation and fast deployment.

Finally, another very interesting privacy topic that has emerged lately and lies at the intersection of algorithms and cryptography is the notion of *data-oblivious* algorithms [49, 109]. Data oblivious algorithms do not perform any data-dependent operations and therefore an adversary observing the flow of the circuit computation cannot distinguish between two different inputs. Applying oblivious algorithms in secure two-party computations can lead to considerable efficiency gains and practical protocols. This is because garbled circuits [113] can be used in this way only for primitive black boxes performing data-dependent operations (e.g., min, max), and not for the whole circuit. Recent results include highly efficient protocols for secure two-party sorting, selection, and permuting [49] as well as for various geometric problems [36]. Since the need for efficient secure two-party computation is now greater than ever, my belief is that there is a lot of research potential on transforming algorithms into oblivious algorithms so that they can be securely used by cloud applications.

More theoretically-oriented future research, and as mentioned in Section 7.1, can involve improving the asymptotic bounds of memory checking [35] by using advanced cryptographic primitives, exploring existence and limitations of optimal authenticated data structures, and studying the dynamization overhead of cloud cryptography[1].

From a practical perspective, designing more efficient authenticated data structures to be used in practice is definitely a big challenge: Most practical applications nowadays extensively use fast authenticated data structures such as Merkle trees, the security of which is however based on totally empirical assumptions (e.g., collision-resistance of SHA-2). This provides great efficiency at the cost of risking the security of the application—e.g., SHA-2 replaced MD-5 due to an attack [103], after MD-5 had been used extensively over the years

[1]So far, most cloud cryptography constructions work for static data only and updates can be handled in a secure way only through total recomputation, which is highly inefficient.

in many systems, such as authenticated file systems and authenticated storage systems. It would be great to come up with authenticated data structures, whose security will be based on a widely-accepted computational assumption (e.g., discrete log) and at the same time can favorably compete in practice with the widely-used Merkle trees.

Bibliography

[1] PBC: The pairing-based cryptography library. http://crypto.stanford.edu/pbc/.

[2] T-mobile sidekick disaster: Danger's servers crashed, and they don't have a backup. http://techcrunch.com/2009/10/10/.

[3] Miklós Ajtai. Generating hard instances of lattice problems (extended abstract). In *Proc. Symposium on Theory of Computing (STOC)*, pages 99–108, 1996.

[4] Aris Anagnostopoulos, Michael T. Goodrich, and Roberto Tamassia. Persistent authenticated dictionaries and their applications. In *Proc. Information Security Conference (ISC)*, pages 379–393, 2001.

[5] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. From secrecy to soundness: Efficient verification via secure computation. In *Proc. International Colloquium on Automata, Languages and Programming (ICALP)*, pages 152–163, 2010.

[6] Mikhail J. Atallah, YounSun Cho, and Ashish Kundu. Efficient data authentication in an environment of untrusted third-party distributors. In *Proc. International Conference on Data Engineering (ICDE)*, pages 696–704, 2008.

[7] Giuseppe Ateniese, Randal Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary Peterson, and Dawn Song. Provable data possession at untrusted stores.

In *Proc. International Conference on Computer and Communications Security (CCS)*, pages 598–609, 2007.

[8] Man Ho Au, Patrick P. Tsang, Willy Susilo, and Yi Mu. Dynamic universal accumulators for DDH groups and their application to attribute-based anonymous credential systems. In *Proc. Cryptographers' Track at the RSA Conference (CT-RSA)*, pages 295–308, 2009.

[9] Ricardo A. Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval: The Concepts and Technology behind Search*. Addison-Wesley, 2nd edition, 2010.

[10] Niko Baric and Birgit Pfitzmann. Collision-free accumulators and fail-stop signature schemes without trees. In *Proc. Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 480–494, 1997.

[11] Mihir Bellare and Daniele Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. In *Proc. Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 163–192, 1997.

[12] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proc. International Conference on Computer and Communications Security (CCS)*, pages 62–73, 1993.

[13] Josh Benaloh and Michael de Mare. One-way accumulators: A decentralized alternative to digital signatures. In *Proc. Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 274–285, 1993.

[14] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13:422–426, 1970.

[15] Manuel Blum, William S. Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the correctness of memories. *Algorithmica*, 12(2/3):225–244, 1994.

[16] Dan Boneh and Xavier Boyen. Short signatures without random oracles and the SDH assumption in bilinear groups. *Journal of Cryptology*, 21(2):149–177, 2008.

[17] Dan Boneh and Matthew K. Franklin. Identity-based encryption from the weil pairing. In *Proc. International Cryptology Conference (CRYPTO)*, pages 213–229, 2001.

[18] Dan Boneh, Ilya Mironov, and Victor Shoup. A secure signature scheme from bilinear maps. In *Proc. Cryptographers' Track at the RSA Conference (CT-RSA)*, pages 98–110, 2003.

[19] Dan Boneh and Alice Silverberg. Applications of multilinear forms to cryptography. *Contemporary Mathematics*, 324(1):71–90, 2003.

[20] Dan Boneh and Brent Waters. Conjunctive, subset, and range queries on encrypted data. In *Proc. Theoretical Cryptography Conference (TCC)*, pages 535–554, 2007.

[21] Andrei Z. Broder and Michael Mitzenmacher. Network applications of Bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2005.

[22] Jan Camenisch, Markulf Kohlweiss, and Claudio Soriente. An accumulator based on bilinear maps and efficient revocation for anonymous credentials. In *Proc. Public Key Cryptography (PKC)*, pages 481–500, 2009.

[23] Jan Camenisch and Anna Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *Proc. International Cryptology Conference (CRYPTO)*, pages 61–76, 2002.

[24] Jan Camenisch and Anna Lysyanskaya. A signature scheme with efficient protocols. In *Proc. Security and Cryptography for Networks (SCN)*, pages 268–289, 2002.

[25] Sébastien Canard and Aline Gouget. Multiple denominations in e-cash with compact transaction data. In *Proc. Financial Cryptography (FC)*, pages 82–97, 2010.

[26] Larry Carter and Mark N. Wegman. Universal classes of hash functions. In *Proc. Symposium on Theory of Computing (STOC)*, pages 106–112, 1977.

[27] Jung Hee Cheon and Dong Hoon Lee. A note on self-bilinear maps. *Korean Mathematical Society*, 46(2):303–309, 2009.

[28] Kai-Min Chung, Yael Kalai, and Salil Vadhan. Improved delegation of computation using fully homomorphic encryption. In *Proc. International Cryptology Conference (CRYPTO)*, pages 483–501, 2010.

[29] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

[30] Scott A. Crosby. *Efficient Tamper-Evident Data Structures for Untrusted Servers*. PhD thesis, Rice University, May 2010.

[31] Reza Curtmola, Juan A. Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *Proc. International Conference on Computer and Communications Security (CCS)*, pages 79–88, 2006.

[32] Ivan Damgård and Nikos Triandopoulos. Supporting non-membership proofs with bilinear-map accumulators. http://eprint.iacr.org/. Cryptology ePrint Archive, Report 2008/538, 2008.

[33] Premkumar Devanbu, Michael Gertz, Chip Martel, and Stuart G. Stubblebine. Authentic third-party data publication. In *Proc. Conference on Database Security (DBSEC)*, pages 101–112, 2000.

[34] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, 1994.

[35] Cynthia Dwork, Moni Naor, Guy Rothblum, and Vinod Vaikuntanathan. How efficient can memory checking be? In *Proc. Theoretical Cryptography Conference (TCC)*, pages 503–520, 2009.

[36] David Eppstein, Michael T. Goodrich, and Roberto Tamassia. Privacy-preserving data-oblivious geometric algorithms for geographic data. In *Proc. International Symposium on Advances in Geographic Information Systems (GIS)*, pages 13–22, 2010.

[37] C. Chris Erway, Alptekin Küpçü, Charalampos Papamanthou, and Roberto Tamassia. Dynamic provable data possession. In *Proc. International Conference on Computer and Communications Security (CCS)*, pages 213–222, 2009.

[38] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Networking*, 8(3):281–293, 2000.

[39] Michael J. Freedman, Kobbi Nissim, and Benny Pinkas. Efficient private matching and set intersection. In *Proc. Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 1–19, 2004.

[40] Joachim Von Zur Gathen and Jurgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, 2nd edition, 2003.

[41] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *Proc. International Cryptology Conference (CRYPTO)*, pages 465–482, 2010.

[42] Rosario Gennaro, Shai Halevi, and Tal Rabin. Secure hash-and-sign signatures without the random oracle. In *Proc. Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 123–139, 1999.

[43] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proc. Symposium on Theory of Computing (STOC)*, pages 169–178, 2009.

[44] Oded Goldreich, Shafi Goldwasser, and Shai Halevi. Collision-free hashing from lattice problems. In *Electronic Colloqium on Computational Complexity (ECCC)*, 3(56), 1996.

[45] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity for all languages in NP have zero-knowledge proof systems. *Journal of the ACM*, 38(3):691–729, 1991.

[46] Michael T. Goodrich, Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Athos: Efficient authentication of outsourced file systems. In *Proc. Information Security Conference (ISC)*, pages 80–96, 2008.

[47] Michael T. Goodrich and Roberto Tamassia. *Algorithm Design: Foundations, Analysis, and Internet Examples*. John Wiley & Sons, 2002.

[48] Michael T. Goodrich, Roberto Tamassia, and Andrew Schwerin. Implementation of an authenticated dictionary with skip lists and commutative hashing. In *Proc. DARPA Information Survivability Conference and Exposition II (DISCEX II)*, pages 68–82, 2001.

[49] Michael T. Goodrich. Randomized Shellsort: A simple oblivious sorting algorithm. In *Proc. Symposium on Discrete Algorithms (SODA)*, pages 1–16, 2010.

[50] Michael T. Goodrich, Charalampos Papamanthou, and Roberto Tamassia. On the cost of persistence and authentication in skip lists. In *Proc. Workshop on Experimental Algorithms (WEA)*, pages 94–107, 2007.

[51] Michael T. Goodrich, Roberto Tamassia, and Jasminka Hasic. An efficient dynamic and distributed cryptographic accumulator. In *Proc. Information Security Conference (ISC)*, pages 372–388, 2002.

[52] Michael T. Goodrich, Roberto Tamassia, and Nikos Triandopoulos. Super-efficient verification of dynamic outsourced databases. In *Proc. Cryptographers' Track at the RSA Conference (CT-RSA)*, pages 407–424, 2008.

[53] Michael T. Goodrich, Roberto Tamassia, and Nikos Triandopoulos. Efficient authenticated data structures for graph connectivity and geometric search problems. *Algorithmica*, 60(3):505–552, 2011.

[54] Eric Hall and Charanjit S. Julta. Parallelizable authentication trees. In *Proc. Selected Areas in Cryptography (SAC)*, pages 95–109, 2005.

[55] Brian Hayes. Cloud computing. *Communications of the ACM*, 51(7):9–11, 2008.

[56] Alexander Heitzmann, Bernardo Palazzi, Charalampos Papamanthou, and Roberto Tamassia. Efficient integrity checking of untrusted network storage. In *Proc. International Workshop on Storage Security and Survivability (STORAGESS)*, pages 43–54, 2008.

[57] Jeffrey Hoffstein, Nick Howgrave-Graham, Jill Pipher, Joseph H. Silverman, and William Whyte. NTRUSIGN: Digital signatures using the NTRU lattice. In *Proc. Cryptographers' Track at the RSA Conference (CT-RSA)*, pages 122–140, 2003.

[58] Andreas Hutflesz, Hans-Werner Six, and Peter Widmayer. Globally order preserving multidimensional linear hashing. In *Proc. International Conference on Data Engineering (ICDE)*, pages 572–579, 1988.

[59] Joseph F. JaJa. *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.

[60] Antoine Joux. A one-round protocol for tripartite Diffie-Hellman. *Journal of Cryptology*, 17(4):263–276, 2004.

[61] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography.* Chapman & Hall/CRC, 2007.

[62] Claire Kenyon and Jeffrey S. Vitter. Maximum queue size and hashing with lazy deletion. *Algorithmica*, 6:597–619, 1991.

[63] Dieter Kratsch, Ross M. McConnell, Kurt Mehlhorn, and Jeremy P. Spinrad. Certifying algorithms for recognizing interval graphs and permutation graphs. In *Proc. Symposium on Discrete Algorithms (SODA)*, pages 158–167, 2003.

[64] Hyung-Mok Lee, Kyung Ju Ha, and Kyo-Min Ku. ID-based multi-party authenticated key agreement protocols from multilinear forms. In *Proc. Information Security Conference (ISC)*, pages 104–117, 2005.

[65] Arjen K. Lenstra, Hendrik W. Lenstra Jr, and László Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, (261):515–534, 1982.

[66] Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid Reyzin. Dynamic authenticated index structures for outsourced databases. In *Proc. International Conference on Management of Data (SIGMOD)*, pages 121–132, 2006.

[67] Feifei Li, Ke Yi, Marios Hadjieleftheriou, and George Kollios. Proof-infused streams: Enabling authentication of sliding window queries on streams. In *Proc. Very Large Data Bases (VLDB)*, pages 147–158, 2007.

[68] Jiangtao Li, Ninghui Li, and Rui Xue. Universal accumulators with efficient nonmembership proofs. In *Proc. Applied Cryptography and Network Security (ACNS)*, pages 253–269, 2007.

[69] Nathan Linial and Ori Sasson. Non-expansive hashing. In *Proc. Symposium on Theory of Computing (STOC)*, pages 509–517, 1996.

[70] Ben Lynn. *On the Implementation of Pairing-Based Cryptosystems*. PhD thesis, Stanford University, November 2008.

[71] Vadim Lyubashevsky and Daniele Micciancio. Generalized compact knapsacks are collision resistant. In *Proc. International Colloquium on Automata, Languages and Programming (ICALP)*, pages 144–155, 2006.

[72] Kyriakos Mouratidis, Man Lung Yiu and Yimin Lin. Efficient verification of shortest path search via authenticated hints. In *Proc. International Conference on Data Engineering (ICDE)*, pages 237–248, 2010.

[73] Petros Maniatis. *Historic Integrity in Distributed Systems*. PhD thesis, Stanford University, August 2003.

[74] Petros Maniatis and Mary Baker. Enabling the archival storage of signed documents. In *Proc. USENIX Conference on File and Storage Technologies (FAST)*, pages 31–45, 2002.

[75] Charles U. Martel, Glen Nuckolls, Premkumar T. Devanbu, Michael Gertz, April Kwong, Stuart G. Stubblebine. A general model for authenticated data structures. *Algorithmica*, 39(1):21–41, 2004.

[76] Alfred Menezes, Scott Vanstone, and Tatsuaki Okamoto. Reducing elliptic curve logarithms to logarithms in a finite field. In *Proc. Symposium on Theory of Computing (STOC)*, pages 80–89, 1991.

[77] Ralph C. Merkle. A certified digital signature. In *Proc. International Cryptology Conference (CRYPTO)*, pages 218–238, 1989.

[78] Daniele Micciancio and Oded Regev. Worst-case to average-case reductions based on gaussian measures. *SIAM Journal on Computing*, 37(1):267–302, 2007.

[79] Ruggero Morselli, Samrat Bhattacharjee, Jonathan Katz, and Peter J. Keleher. Trust-preserving set operations. In *Proc. Conference on Computer Communications (INFOCOM)*, 2004.

[80] James K. Mullin. Spiral storage: Efficient dynamic hashing with constant-performance. *Computer Journal*, 28:330–334, 1985.

[81] Moni Naor and Kobbi Nissim. Certificate revocation and certificate update. In *Proc. USENIX Security Symposium (USENIX)*, pages 217–228, 1998.

[82] Moni Naor and Guy Rothblum. The complexity of online memory checking. *Journal of the ACM*, 56(1), 2009.

[83] Lan Nguyen. Accumulators from bilinear pairings and applications. In *Proc. Cryptographers' Track at the RSA Conference (CT-RSA)*, pages 275–292, 2005.

[84] Glen Nuckolls. Verified query results from hybrid authentication trees. In *Proc. Conference on Database Security (DBSEC)*, pages 84–98, 2005.

[85] National Institute of Standards and Technology. Secure hash standard (SHS). October 2008.

[86] Rafail Ostrovsky. Efficient computation on oblivious RAMs. In *Proc. Symposium on Theory of Computing (STOC)*, pages 514–523, 1990.

[87] Mark H. Overmars. *The Design of Dynamic Data Structures*. Springer-Verlag, LNCS 156, 1983.

[88] HweeHwa Pang and Kyriakos Mouratidis. Authenticating the query results of text search engines. *VLDB Endowment*, 1(1):126–137, 2008.

[89] HweeHwa Pang and Kian-Lee Tan. Authenticating query results in edge computing. In *Proc. International Conference on Data Engineering (ICDE)*, pages 560–571, 2004.

[90] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Authenticated hash tables. In *Proc. International Conference on Computer and Communications Security (CCS)*, pages 437–448, 2008.

[91] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Optimal authenticated data structures with multilinear forms. In *Proc. International Conference on Pairing-Based Cryptography (PAIRING)*, pages 246–264, 2010.

[92] Charalampos Papamanthou and Roberto Tamassia. Time and space efficient algorithms for two-party authenticated data structures. In *Proc. International Conference on Information and Communications Security (ICICS)*, pages 1–15, 2007.

[93] Charalampos Papamanthou and Roberto Tamassia. Cryptography for efficiency: Authenticated data structures based on lattices and parallel online memory checking. http://eprint.iacr.org/. Cryptology ePrint Archive, Report 2011/102, 2011.

[94] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Optimal verification of operations on dynamic sets. In *Proc. International Cryptology Conference (CRYPTO)*, 2011.

[95] Chris Peikert. Public-key cryptosystems from the worst-case shortest vector problem (extended abstract). In *Proc. Symposium on Theory of Computing (STOC)*, pages 333–342, 2009.

[96] Franco P. Preparata and Dilip V. Sarwate. Computational complexity of Fourier transforms over finite fields. *Mathematics of Computation*, 31(139):740–751, 1977.

[97] Franco P. Preparata and Michael I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

[98] Oded Regev. Lattice-based cryptography. In *Proc. International Cryptology Conference (CRYPTO)*, pages 131–141, 2006.

[99] Oded Regev. On the complexity of lattice problems with polynomial approximation factors. *The LLL algorithm*, pages 475–496, 2010.

[100] Tomas Sander. Efficient accumulators without trapdoor (extended abstract). In *Proc. International Conference on Information and Communications Security (ICICS)*, pages 252–262, 1999.

[101] Tomas Sander, Amnon Ta-Shma, and Moti Yung. Blind, auditable membership proofs. In *Proc. Financial Cryptography (FC)*, pages 53–71, 2001.

[102] Victor Shoup. *A Computational Introduction to Number Theory and Algebra*. Cambridge University Press, 2nd edition, 2008.

[103] Marc Stevens, Alexander Sotirov, Jacob Appelbaum, Arjen Lenstra, David Molnar, Dag Arne Osvik, and Benne Weger. Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate. In *Proc. International Cryptology Conference (CRYPTO)*, pages 55–69, 2009.

[104] Roberto Tamassia and Nikos Triandopoulos. Efficient content authentication in peer-to-peer networks. In *Proc. Applied Cryptography and Network Security (ACNS)*, pages 354–372, 2007.

[105] Roberto Tamassia. Authenticated data structures. In *Proc. European Symposium on Algorithms (ESA)*, pages 2–5, 2003.

[106] Roberto Tamassia and Nikos Triandopoulos. Computational bounds on hierarchical data processing with applications to information security. In *Proc. International Colloquium on Automata, Languages and Programming (ICALP)*, pages 153–165, 2005.

[107] Roberto Tamassia and Nikos Triandopoulos. Certification and authentication of data structures. In *Proc. Alberto Mendelzon Workshop on Foundations of Data Management*, 2010.

[108] Nikos Triandopoulos. *Efficient Data Authentication*. PhD thesis, Brown University, September 2006.

[109] Guan Wang, Tongbo Luo, Michael T. Goodrich, Wenliang Du, and Zutao Zhu. Bureaucratic protocols for secure two-party sorting, selection, and permuting. In *Proc. Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 226–237, 2010.

[110] Peishun Wang, Huaxiong Wang, and Josef Pieprzyk. A new dynamic accumulator for batch updates. In *Proc. International Conference on Information and Communications Security (ICICS)*, pages 98–112, 2007.

[111] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding collisions in the full SHA-1. In *Proc. International Cryptology Conference (CRYPTO)*, pages 17–36, 2005.

[112] Yin Yang, Dimitris Papadias, Stavros Papadopoulos, and Panos Kalnis. Authenticated join processing in outsourced databases. In *Proc. International Conference on Management of Data (SIGMOD)*, pages 5–18, 2009.

[113] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *Proc. Foundations of Computer Science (FOCS)*, pages 160–164, 1982.