Abstract of "Efficient Data Authentication" by Nikolaos Triandopoulos, Ph.D., Brown University, May 2007.

We address the problem of authenticating data in untrusted, or adversarial, computing environments: when the distributor of the data is not the source of the data, and thus is not trusted by the end-user, how can data received be proven authentic? Data authentication constitutes a fundamental problem in information security and an interesting new dimension in data management and data structure design. At the same time, the problem captures the security needs of many computing applications that exchange and use sensitive information in hostile distributed environments and its importance increases given the current trend in modern system design towards decentralized architectures with minimal trust assumptions. Solutions should not only be provably secure, but efficient and easily implementable.

This dissertation presents an extensive study of data authentication. We examine the problem for both structured and unstructured data, provide formal definitions, and design new efficient techniques for authenticating general classes of query problems, such as graph and geometric search problems, and data streams. We also study the complexity of data authentication, deriving lower bounds for the important special case of authenticating set membership queries, and design new optimal constructions. Moreover, we provide a new general framework for authenticating any query over structured data, which decouples the answer-validation and answer-generation procedures. Finally, we design totally decentralized authentication structures that provide authentication for data distributed over any peer-to-peer overlay network.

Efficient Data Authentication

by

Nikolaos Triandopoulos

Diploma, Computer Engineering & Informatics, University of Patras, 1999

Sc. M., Computer Science, Brown University, 2002

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 2007

This dissertation by Nikolaos Triandopoulos 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 _____ _____

Eli Upfal, Reader

Approved by the Graduate Council

Date _____ _____

Sheila Bonde
Dean of the Graduate School

Vita

Nikolaos (Nikos) Triandopoulos was born in Athens, Greece, in 1976. Nikos received his undergraduate diploma in Computer Engineering and Informatics at the University of Patras, Greece, in 1999 and his Sc.M. in Computer Science at Brown University in 2002. He completed his Ph.D. in Computer Science at Brown in 2006 and his dissertation studies the problem of authenticating information in hostile and adversarial computing environments. His primary research areas are in information security, cryptography, algorithms and data structures. He has been a recipient of the Paris Kanellakis Fellowship and the Technological Innovation Award from Brown University, and the recipient of the I3P Postdoctoral Research Award from the Institute for Information Infrastructure Protection.

Acknowledgments

I wish to specially thank my advisor Roberto Tamassia for introducing me to the main topic of this dissertation, spending time and sharing his thoughts while I was working on the subject and for his support whenever there was need. I am very happy that I have been interacting with Roberto during all these six years that I spent as a graduate student. His advice and guidance on research and academics have been a valuable resource for me.

I am also grateful to my committee members, Michael T. Goodrich, Anna Lysyanskaya and Eli Upfal, for the collaboration we have had together and the help and encouragement they provided me to achieve my thesis goals. Through my interaction with them, either in the classroom or while contacting research, I gained knowledge that significantly affected the results of this dissertation. Their enthusiasm for research has been a motivation for me.

The ideas and parts of the text presented in this thesis were developed in collaboration with: Roberto Tamassia, Michael T. Goodrich and Anna Lysyanskaya. This research was supported in part by the National Science Foundation under grants CCR–0311510 and IIS–0324846, in part by the Defense Advanced Research Projects Agency under grant F30602–00–2–0509, and also in part by a research gift by IAM Technologies, Inc.

I have been fortunate to live a great learning experience while at Brown CS department and I thank the faculty for providing an excellent academic environment and, also, the technical and administrative staff for their help. I thank Philip Klein, David Croston and Aris Anagnostopoulos for useful technical discussions on topics related to this thesis. Also, I feel honored to have been one of the Paris Kanellakis fellows and I wish to thank General and Mrs. Kanellakis for their encouragement towards the completion of this dissertation.

Finally, I would like to specially thank my parents and my sister for their love and continuous care and support throughout my graduate studies. To my wife Olga, I wish to express my gratitude for always being next to me and I send my biggest thanks, for it is hard to achieve goals without love. This thesis is dedicated to my grandfather, to whom I was unable to say goodbye: Γιατί έτσι μας το θέμε, κι έτσι μας το κάμε, παππού.

Contents

⋆ Parts of this dissertation have previously appeared as [59, 82, 142] or have been submitted for publication in conferences or journals.

List of Tables

List of Figures

Chapter 1

Introduction

This dissertation addresses the problem of authenticating data that is retrieved in untrusted, or adversarial, computing environments: when the distributor of the data is not the source of the data, and thus is not trusted by the end-user, how can data received be proven authentic? This question is the core of several security-related problems underlying any real-life computing application that involves dissemination of data over a communication or computing structure that can act unreliably. Clearly, data authentication—ensuring that received data can be accurately verified to be in its original form—is a fundamental problem in the area of information security, for information is valuable only when it is trustworthy.

Data authentication captures some primary security needs of today's computing reality. Indeed, with the growth of the Internet and distributed and pervasive computing, more and more data is queried and retrieved according to non-traditional patterns. That is, the distributor of the data is not necessarily the source of the data and, thus, it cannot be trusted by the end-user. At the same time, a growing number of computing applications exchange, share and use sensitive information in a totally distributed computing setting, where no central authority controls the data traffic and no trust between participating entities has been established prior to data communication. As data validity is critical for the end-user, data authenticity is crucial for various target applications. Accordingly, to achieve high degree of information assurance, we need methods that allow the recipient of information to verify whether the data received is authentic—that is, it has been retrieved intact as its source intended—or invalid—that is, it has been erroneously or maliciously altered by some entity acting as the "man in the middle". Admittedly, in these scenarios data authentication becomes the number one security goal because it is essential that information is verified before it is consumed by higher-level applications. The problem is of increasing importance

given the current trend in modern system design towards decentralized architectures with minimal trust assumptions.

Verifying data authenticity in such settings is a challenging task from both theoretical and practical perspectives: solutions should not only be provably secure with firm foundations, but also efficient and easily implementable. From a technical point of view, the problem of data authentication lies in the overlap of information security with algorithms and cryptography, where it is often hard to combine security with efficiency. In particular, when directly applied to the data authentication problem, traditional message authentication techniques are inadequate to provide efficient and viable solutions. For instance, since data can often be structured, dynamically evolving over time, and users query or receive only subsets of it, data cannot be treated as a whole, merely as a single message. At the same time, data authentication constitutes a relatively unexplored—yet, very interesting—new dimension in data management and data structure design, where new methodologies and new authentication schemes are required for achieving satisfactory solutions.

This dissertation presents an extensive study on the data authentication problem. We examine the problem for both structured data—that is being retrieved through queries—and unstructured data—that is being received as a stream. We provide formal problem definitions that carefully model the notions of security and efficiency in data authentication, and design new efficient techniques for securely authenticating general classes of query problems, such as queries on graphs and geometric search problems, and data streams. We also study the complexity of the problem and the computational and communication costs that are inherently associated with the authentication of data, deriving lower bounds for the important special case of authenticating set-membership queries, for which problem we also design an asymptotically optimal new construction. Moreover, in a general computational and data querying model, we provide a new framework for authenticating any query type over structured and dynamic data, proving a general, constructive, possibility result for data authentication, and we show that the authentication of general queries is reduced to the authentication of set-membership queries. By decoupling the answer-generation and answer-validation procedures, this framework provides us with an interesting new methodology for designing super-efficient data authentication schemes, where an answer to a query can be verified in time asymptotically less than the time spent to produce it. Finally, we address the problem of distributed data authentication, where we design totally decentralized authentication schemes that provide authentication for data stored, shared and retrieved over any distributed peer-to-peer network. These schemes are based on the design of an efficient distributed authentication tree, the first of this kind.

1.1 The Problem of Data Authentication

Informally, the *data authentication problem* is described as follows. The setting is very simple, consisting of a basic data querying and communication protocol between three parties: the *source*, the *distributor* and the *user* (see Figure 1.1). The source creates data D_s, which becomes available to the distributor, and then the user requests from the distributor and receives data D_r, which is related to data D_s. Here, the only assumption is that the source and the distributor are distinct entities and, thus, the user trusts only the source—otherwise, why participating in the protocol? The core problem that we study is the authentication of the data that reaches the user, that is, we wish to provide assurance that received data is authentic, meaning that data D_r is received as if it was directly produced and sent to the user by the data source.

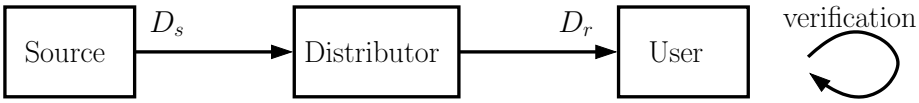

Figure 1.1: Data D_s, produced by the source, becomes available through the distributor. Data D_r, provided by the distributor, reaches the user who wishes to verify its authenticity.

In particular, the problem of data authentication involves the design of techniques and protocols, in this three-party model for data dissemination, that allow the user to locally perform a verification test on the received data. This verification procedure enables the user to either verify that the received data is authentic or identify that it has been erroneously or maliciously altered by the distributor—in which case, data is rejected as invalid. Accordingly, the main goal is to design a data authentication scheme (or, simply, an authentication scheme) that augments any existing data communication or data querying protocol between the three parties, such that data verification is successfully accomplished at the user's side and, consequently, data authentication is ensured. To achieve this goal, two requirements must be met: *security* and *efficiency*.

Security is a property that by definition should characterize any solution to the data authentication problem. That is, data authentication is achieved when the verification procedure correctly decides on the authenticity of the received data and independently of the distributor's behavior. In particular, to achieve high level of security, the data distributor is modeled as an entity that not only is untrusted by the user but also can act maliciously, aiming at falsifying the data that reaches the user. Following a cryptographic approach, the distributor is modeled as an entity that is controlled by an adversary of polynomial-time

computational power. In this setting, the security requirement guarantees that, subject to some general assumptions about the hardness of some computational problems and independently of the distributor's malicious action, data verification at the user's side is always reliable and correct: valid data is correctly identified as being authentic and invalid data is correctly identified as being forged. Data authentication by definition dictates the verification procedure to be secure, ensuring, in essence, that completeness and soundness (in a computational sense) are satisfied: data is verified to be valid if and only if it is authentic.

Efficiency is a property that characterize the feasibility and practicality of any (secure) technique for data authentication. A data authentication scheme is designed to become an augmentation or extension of some underlying data querying and communication protocol, offering an important security service. Albeit an integral component of the whole system, the authentication scheme should ideally be light-weigh and easy to implement, incurring no significant computational and communication overhead and, thus, not downgrading performance. In particular, the goal here is to authenticate data at the minimum possible computational and communication costs. And in this case, a satisfying notion of efficiency can be defined as follows: an authentication scheme is considered to be efficient if it asymptotically incurs no additional communication cost to the underlying communication protocol and no additional time or space costs to the three involved parties. Often, trade-offs between individual cost parameters (time, space, communication) can be allowed—never between security and efficiency, though.

Finally, we note the following important characteristic of the data authentication problem. Although the problem is defined in a three-party communication model, we require that any authentication scheme is oblivious of the user; instead, it should provide data verification to a large—potentially unbounded—number of users that query and retrieve data sent by the source through the distributor. We impose no assumption on the users.

1.1.1 Motivation and Applications

The above description corresponds to an abstract form of the data authentication problem and offers the most general problem formulation. Important categories of the problem are obtained and a rich set of application areas are covered as we consider concrete computational settings for the underlying data querying and communication model, as we instantiate the parties' individual computational roles, and also, as we introduce a large set of users.

Yet, even in this general form, data authentication captures the fundamental security problem that arises in the digital world whenever data is received not from its creator, but

from an intermediate third party that can act adversarially. In this data dissemination setting, authentication is indeed essential, since it is the basic requirement for attaining information assurance and performing trustworthy computation. As we argue below, this is already a commonly seen setting in today's computing reality. And we anticipate this to be an even more frequent practice in the near future. Consider pervasive computing [144], the next generation computing paradigm where computation is integrated into the environment with information communication technology being embedded into everyday objects. In a totally mobile and highly distributed computing environment, we envision that people will interact with information-processing devices located everywhere at all times; these devices will be constantly receiving data, who knows where from.

In its abstract form, data authentication involves the verification of data D_r received from an original data collection D_s through a (potentially adversarial) data distributor (see Figure 1.1). The only implicit assumption is that received data D_r is related to data D_s produced by the source, so that $D_r = f(D_s)$ for some function f. We distinguish two important cases in terms of data format and retrieval. First, data D_s can be *structured*, that is, it is an arbitrarily complex data set that is organized in a data structure and it is retrieved by the user through queries issued to the distributor, who maintains set D_s through update operations performed by the source. In this case, received data D_r should be the answer to the query q issued by the user, namely $D_r = f_q(D_s)$. Alternatively, data D_s can be *unstructured*, that is, it has the flat format of a data stream and is received by the user in blocks that come through the distributor. In this case, received data D_r should be the collection of blocks distributed by the data source. Authentication of structured and unstructured data constitute two important forms of the problem we study.

Using these forms and by considering concrete data dissemination scenarios and specific computational roles for the three parties, let us now describe how various real-life computing applications, built over popular data retrieval architectures, instantiate the data authentication problem. First, consider structured data (e.g., a database) that is produced by a server and then is replicated at remote untrusted and unprotected mirror servers, geographically distributed over a network. Client applications query the data by contacting some mirror server. This scenario models data replication and publication in the Internet and applications include content delivery networks, outsourced databases, data publication in portals, or any distributed information systems that cache data at proxy sites. Also, consider unstructured data that reaches users in a form of a stream through a computer network. Network nodes forward or process the transmitted stream, but they may be compromised or mis-configured, so that they can disrupt the transmission of the stream. This

scenario models the multicast data transmission setting covering a wide application area (e.g., information broadcast centers, software updates etc.) and generally any system that provides stream-based data dissemination (e.g., publish/subscribe systems). Finally, consider data distribution through peer-to-peer overlay networks or wireless mobile networks, where data (either structured or unstructured) produced by a source is dispersed to the nodes of a network through which users query and receive data. Since network nodes are untrusted (e.g., there is no control over the participating peers in a peer-to-peer network, or queried computing devices may have been compromised) data authentication is essential. This scenario models the data authentication problems in distributed storage systems (e.g., applications built on top of distributed hash tables) and other systems where data is retrieved from remote untrusted machines (e.g., geographic information systems).

1.1.2 Data Authentication Technique

To describe our main authentication technique, we first need to examine whether the data authentication problem is related to the important, and well-studied by the cryptographic community, problem of message authentication (see, e.g., [48], Chapter 6). Here, the setting is as follows: messages are transmitted by a sender to a receiver over an insecure communication channel that is controlled by an adversary. The goal is to authenticate the transmitted messages so that the receiver can identify whether a received message is the original message (sent by the sender) or a modified one (modified by the adversary). Message authentication schemes allow the receiver to verify whether or not a message received through an insecure channel is identical to the message that was originally sent by the sender over the channel. Digital signatures and message authentication codes are two important classes of message authentication schemes. Their main difference is that public-key cryptography is used by the former whereas secret-key cryptography is used by the latter; in particular, the verification key is public (known to everyone) for digital signatures but secret (known only to the legitimate receiver(s)) for message authentication codes.

Although they are related, the problem of data authentication and message authentication are distinct. What distinguishes data authentication from message authentication is that received data D_r is related to data D_s produced by the source through some function f, that is, $D_r = f(D_s)$. Instead, message authentication deals with verification of the entire message that produced by the sender. We see that message authentication instantiates data authentication when function f is the identity function, or better, that the data authentication problem constitutes a generalization of the message authentication problem.

Consequently, data D_s cannot be treated as a whole, merely as a single message. Moreover, data authentication is not only about data integrity: the verification procedure should not merely authenticate that data is received intact; it should also authenticate that data is received reliably, according to the specific querying patterns expressed by function f. Thus, data authentication typically involves verification of computational correctness as well.

That said, however, we still need to examine whether straightforward reductions from data authentication to message authentication exist that would potentially suggest the existence of efficient solutions to the data authentication problem. In general, this is not the case: directly applying message authentication schemes in data authentication provide inefficient or no viable solutions. For instance, employing message authentication codes can be quite problematic, almost impractical, because of the use of secret-keys and the fact that data authentication schemes should support large sets of users. On the other hand, merely applying digital signatures (e.g., signing every piece of information) is also problematic and rather inefficient. For instance, for structured data that is dynamically evolving over time, signing every data element incurs high computational and communication costs. Similarly, for unstructured data transmitted in blocks, signing every block also incurs high overhead.

Instead, more careful considerations should be made when using message authentication schemes and elegant data authentication techniques should be designed. The state-of-the-art data authentication technique, that can provide both security and efficiency, is what is known as *signature amortization*, briefly described as follows. Assuming a public-key infrastructure and, in particular, that the public-key of the source is known to the user, the technique amortizes only one digital signature over the entire data set D_s, independently of the data-retrieval patterns issued by the user. The signature is generated by the source on a special"message" $d = g(D_s)$, produced by applying some function g over data set D_s, and becomes available to the distributor. Function g is specially designed for the individual data authentication problem and incorporates cryptographic (or other) primitives such that string d satisfies certain security properties. Accordingly, the data D_r that reaches the user includes the signature on d and additional authentication information that allows the user to recompute string d. Given the security properties of function g and string d, data verification is then reduced to the verification of the signature on d.

When signature amortization is used, security and efficiency of any data authentication scheme are both properties of function g (used to produce the special message d that is signed). Designing signature amortization such that these properties are achieved is a challenging task. An important class of data authentication schemes rely on the use of hierarchical collision-resistant cryptographic hashing over data D_s for implementing function

g and computing the special message (often called digest) d that is signed. We refer to this solution concept as *hash-based data authentication* (or hash-based authentication schemes).

1.2 Overview and Thesis Structure

This dissertation presents an extensive study on data authentication. We formally define the data authentication problem for both structured data that is being retrieved through queries and unstructured data that is being received as a stream, capturing the notions of security and efficiency in data authentication. We use and extend the signature amortization technique to design new efficient authentication schemes for authenticating general classes of query problems, such as queries on graphs and geometric search problems, and data streams. We study the computational and communication costs that are inherently associated with hash-based data authentication, deriving lower bounds for the important special case of authenticating set-membership queries and designing an asymptotically optimal new construction. We address the problem of distributed data authentication, designing totally decentralized efficient authentication schemes that provide authentication for data stored and retrieved over any distributed peer-to-peer network. Finally, we provide a new general framework for authenticating any query type over structured data, proving general possibility and characterization results for efficient (or super-efficient) data authentication. We next overview our contributions and describe how they are organized in this dissertation.

For authenticating structured data, authenticated data structures [103, 141] provide a general solution concept modeling the data-querying paradigm in which an untrusted entity (not the creator of the stored data) maintains a data structure and answers queries on it. Using signature amortization, they support authenticated queries: answers to queries contain cryptographic proofs that allow verification of their authenticity. Prior research has focused on authenticating set-membership queries or search queries on ordered data sets. However, more complex queries over highly structured data sets do not necessarily fall into this category. For instance, how can we authenticate aggregation queries over sequences, connectivity queries on graphs, or point location queries over subdivisions of the plane?

In Chapter 2, we introduce a new hash-based authentication scheme for efficiently authenticating a broad class of queries, namely, any decomposable query over a sequence of data elements. This class includes queries about any property of subsequences of elements that are can be answered by applying an associative operator over the elements, e.g., aggregation queries, and it also includes searching over ordered sets. The authentication scheme incurs costs that are logarithmic on the sequence size. Using a divide-and-conquer

approach, we show how to employ the above technique for efficiently authenticating queries about properties of paths in graphs (i.e., given two nodes, report a certain decomposable property that is related to a corresponding connecting path) and iterated search queries on multi-catalogs that are organized as a graph (i.e., locate a query element in each catalog of a set of catalogs), thus, providing authentication for a large set of queries over data of topological structure. For instance, for graphs of size n, queries about properties of a path of size k are authenticated at $O(\log n)$ or $O(\log n + k)$ cost, depending on the property type. Also, iterated search in m catalogs of total size n is authenticated at cost $O(\log n + m)$. These results allow the design of new, efficient authenticated data structures for a rich class of path and connectivity queries over graphs (e.g., whether two nodes are connected with a path of length 4) and complex queries on 2-dimensional geometric objects (e.g., which region contains a given point). In essence, the authentication of these queries is reduced to the authentication of path property and iterated search queries on appropriate data sets.

With respect to the efficiency and simplicity of authenticated data structures, it is important to examine the overhead introduced due to authentication. Moreover, for hash-based data authentication—the popular solution concept that is based on signatures and cryptographic hashing according to the "hash and sign" paradigm—it is important to study how complex the problem is in this case and what are the limitations that inherently exist in designing efficient authentication schemes. No such study has been carried out before.

In Chapter 3, we study the computational, communication and storage costs that are inherently associated with the model of authenticated data structures, for the first time investigating lower bounds on these costs. For the fundamental problem of authenticating membership queries on a set of size n, we prove that any hash-based authentication scheme incurs an $\Omega(\log n)$ authentication cost, answering an open problem posed in [103]. Actually, this logarithmic lower bound is tight in the following sense: considering parameterized signature amortization, where $k \geq 1$ data digests are signed, we get $\Omega(\log(n/k))$ authentication costs, thus, even $O(n^{1-\epsilon})$ signed digests do not suffice to break the logarithmic bound. In view of this lower bound, we design a new authentication scheme for set-membership queries that achieves costs that are very close to the theoretical optimal. Our upper and lower bounds hold more generally for hierarchical data processing, a new class of problems that involve certain types of computations on directed acyclic graphs. In particular, they apply to: (i) data-structure design, where we get a new version of the skip-list data structure that achieves the best known expected search complexity, namely, searching in a set of size n results in $1.25 \log n$ comparisons on average, and (ii) the problem of multicast key distribution for secure group communication, where we get the first exact worst-case

logarithmic lower bound on the communication cost of protocols based on key-graphs. For completeness, we include these additional results in our exposition.

For authenticating unstructured data, we consider stream-based data dissemination and a new data-authentication model, in which data is transmitted in a flat form as a stream, over a unreliable, erroneous or adversarial medium, and thus it is delivered with losses or injections, altered and unordered. In this setting, the problem is formulated by considering a fully adversarial data distributor who can simultaneously inflict losses, injections and alterations on the data stream. Prior research has focused on less general forms of the problem or has imposed strong assumptions on the data communication model.

In Chapter 4, we provide a formal definition of the data-stream authentication problem in a very general data communication setting and assuming a powerful, fully adversarial, data distributor that can simultaneously inject, delete, modify or reorder data blocks. In this model, we design a new authentication technique for providing efficient data-stream authentication. We enhance signature amortization by using a novel combination of error-correcting codes with signatures and cryptographic hashing and we design an authentication scheme that tolerates any number of symbol additions, modifications and deletions, subject to some minimum assumptions about the reliability of the underlying channel which are expressed by two constant parameters. In essence, we design an authenticated error-correcting code and we prove that, in the bounded computational model for communication channels, list-decoding can be transformed to unambiguous decoding. Applying our technique to the problem of multicast authentication, we get the first efficient authentication scheme in this new adversarial model. Our scheme incurs only constant per-packet communication overhead and allows the detection of whether the data-stream is under attack.

With the growth of distributed computing, it is important to find totally decentralized solutions for the data authentication problem. New design patterns for system implementations increasingly make use of distributed data storage and computing resources. For instance, many systems are built on top of distributed search structures implemented over peer-to-peer networks, which lack central management or control. Clearly, authenticated data structures do not perfectly fit in this computing and data querying model, since they are based on the client-server query model, where data distribution occurs simply by replicating the entire data structure to remote untrusted servers. Instead, in distributed environments authentication of data should also be implemented in a totally decentralized fashion and authentication schemes should be distributed (on a per data item basis) over remote, untrusted participating peers of an overlay network. Prior research has examined distributed data authentication using a naive "sign-everything" approach which, in this data storage

and computation setting, can be either inefficient or insecure.

In Chapter 5, we introduce a distributed data authentication model in which both query answering and cryptographic proofs are distributed over a peer-to-peer network. The model captures the security needs of many distributed systems or emerging applications built on them. Over any distributed object-location system (for instance, any distributed hash table) we realize the first distributed version of a Merkle tree, a fundamental authentication construction, which yields distributed versions of many security protocols based on it. For instance, in a network of n nodes, we can authenticate membership in a dynamic set of m data elements in $O(\log n \log m)$ time using $O(\log m)$ storage (per network node). Based on this, we design an efficient authenticated distributed hash table. Our authentication structures do not depend on the implementation of the distributed location system, which allows them to leverage existing peer-to-peer architectures. Additionally, we present another implementation of a distributed Merkle tree that by construction achieves load-balance. As we discuss next, our results on distributed authentication generalize to any data query type.

Finally, for structured data, it is interesting to study whether possibility results exist for general query types. Prior research has shown possibility results for a specific class of search problems over static data and in the pointer machine model of computation, where answer verification is performed in time proportional to the time spent to answer the query.

In Chapter 5, we also introduce a general computational and data querying model for authentication of structured dynamic data. We provide a formal problem definition and present a new framework for authenticating any type of query, proving a general, constructive, possibility result. Interestingly, our construction also shows that the authentication of general queries is reduced to the authentication of set-membership queries. This reduction is of special importance, given the significant progress research has made studying security aspects of the set-membership problem. In particular, by combining this reduction with our distributed Merkle tree implementation, we immediately get that distributed data authentication is feasible for general queries. Moreover, by decoupling the answer-generation and answer-validation procedures, our new framework provides sufficient conditions for the design of super-efficient data authentication schemes, where answer verification can be performed in time asymptotically less than the time spent to answer the query.

Attempt has been made so that the chapters are self-contained in terms of the specific problem they study and the background technical material needed for the results they present. Also, the rich set of previous and related work of the problems this dissertation studies is contained in the individual chapters.

Chapter 2

Authentication of Graph and Geometric Search Queries

2.1 Introduction

Verifying information that at first appears authentic is an often neglected task in data structure and algorithm usage. Fortunately, there is a growing literature on correctness checking that aims to rectify this omission. Following early work on program checking and certification (e.g., [10, 139, 140]), several researchers have developed efficient schemes for checking the results of various data structures (e.g., [14, 15, 16, 42, 92]), graph algorithms (e.g., [36, 70]), and geometric algorithms (e.g., [35, 88]). These schemes are directed mainly at defending the user against an inadvertent error made during implementation. In addition, these previous approaches have primarily assumed that usage is limited to a single user on an individual machine.

With the advent of Web services and Internet computing, data structures and algorithms are no longer being used just by a single user on an individual machine. Indeed, with the development of content distribution services (e.g., Akamai) spreading content across the Internet, decentralized large-scale data-management systems or systems designed for distributed and pervasive computing, the machine responding to a user's query could be unknown to both the data-structure author and the user. More generally, in today's computing reality, more and more, the source and the distributor of the data are different entities or machines with distinct identities. Consequently, it is very common that the owner of a data

set does not control the data structure that is used to answer queries on this set. We must recognize that, although they benefit efficiency, such scenarios open the possibility that an agent hosting a data structure or an algorithm could deliberately falsify query responses to users. Such falsification could cause significant adverse consequences, especially when the information represented by the response to a query is crucial to the target application (e.g., it has security or financial implications). We want to guard against this possibility.

In this chapter we study a new dimension in data structure and algorithm checking—how can we design sophisticated data structures and algorithms so that their responses can be verified as accurately as if they were coming from their author, even when the response is coming from an untrusted host? Examples of the kind of information we want to authenticate include dynamic documents, online catalog entries and the responses to queries in geographic information systems, financial databases, medical information systems and scientific databases. In particular, we are interested in efficiently verifying information related to paths and connectivity in computer networks (or other combinatorial graph structures), even when the network is changing. In addition, we are interested in verifying complex geometric queries, such as range searching queries, ray shooting queries, and point location queries, which are used extensively in spatial databases or geographic information systems.

Digital signatures, used in a per-query basis, can be used to verify simple static documents, but are inefficient for dynamic data structures. Indeed, the main challenge in providing an integrity and authentication service in the above contexts is that the space of possible answers is much larger than the data size itself. For example, there are $O(n^2)$ different paths in a tree of n nodes, and many of these paths have $O(n)$ size. Requiring an authenticator to sign every possible answer-response pair is therefore prohibitive, especially when the data is changing due to the insertion or deletion of data elements in the set. We therefore need new techniques for authenticating the answers of complex data structures.

The state-of-art solution for this problem is *signature amortization*. Ideally, we would like our authenticator to sign just one single digest of our data structure, that is, a short secure description of it, such that the answer to any query can be securely and efficiently verified subject to the verification of this single signature. In our work, collision-resistant hashing is the cryptographic primitive used to produce the data digest, the latter being built from the careful combination of cryptographic hashes over subsets of the data set. Thus, we consider *hash-based data authentication*. The computation of the data digest and the hashes of partial data must be performed in accordance with the type of issued queries: it should be specifically designed such that data authentication can be efficiently supported. In essence, if we can achieve such a scheme, then the verification of the answer to a query

in the database can be reduced to the problem of collecting the appropriate partial hashes (and partial data) that allow a user to certify the answer, recompute the digest of the entire structure and, finally, compare this with the digest that is signed by the authenticator.

Even when we follow this approach, however, we are faced with the challenge of how to subdivide the data in a way that allows efficient assembly of partial cryptographic hashes and efficient computation of the digest of the entire structure for any possible query. For simple data structures, such as dictionaries, this subdivision is fairly straightforward (say, using a linear ordering and a Merkle hash tree [94, 95]; see also [52, 103]), but for complex data structures, such as graphs, geometric structures, or structures built using the fractional cascading paradigm, this subdivision method is far from obvious. For instance, for these problems there is no linear ordering among the data items, upon which one could build a (single) hash tree.

2.1.1 A Model for Authenticated Data Structures

Our data authentication model involves three parties: a trusted *source*, an untrusted *responder*, and a *user*. The *source* holds a structured collection S of objects, where we assume that a repertoire of *query operations* are defined over S. If S is fixed over time, we say that it is *static*; otherwise, we say that S is *dynamic* and assume that a repertoire of *update operations* is defined that modify S.

For example, S can represent a network whose nodes and edges store data items on which the following two query operations are defined: a *connectivity query* on S asks whether two given nodes of S are in the same connected component and a *path query* returns a path (or information related to the path), if any, between two given nodes. We can also define update operations of S that add and/or remove nodes and edges. As a second example, S can be a collection of line segments in the plane forming a polygonal chain, where an *intersection query* returns all the segments intersected by a given query line. In this case we can define update operations that insert and/or remove segments.

The *responder* maintains a copy of collection S together with *structure authentication information*, which consists of time-stamped statements about S signed by the source. If S is dynamic, the responder receives, together with each update on S, some *update authentication information*, which consists of signed time-stamped statements about the update and the current state of S. The *user* performs queries on S, but instead of contacting the source directly, it queries the responder. The responder provides the user with an answer to the query together with *answer authentication information*, which yields a cryptographic

proof of the validity of the answer. The answer authentication information should include a signed time-stamp—indicating the freshness of the answer—and information derived from the structure authentication information. The user then verifies the proof relying solely on the time-stamp, the answer authentication information and the information derived from statements signed by the source. Accordingly, the user accepts or rejects the answer. In terms of security, we allow the untrusted responder to be controlled by a polynomial-time adversary and require that, subject to standard cryptographic assumptions, the responder cannot cheat the user, i.e., no false answer to some query can be accepted as authentic.

The data structures used by the source and the responder to store collection S, together with the protocols and algorithms for queries, updates and verifications executed by the various parties, form what we call an *authenticated data structure* [52, 86, 103, 141]. In a practical deployment of an authenticated data structure, there would be various instances of geographically distributed responders. Such a distribution scheme reduces latency, allows for load balancing, and reduces the risk of denial-of-service attacks. Scalability is achieved by simply increasing the number of responders. Indeed, since the responders are not trusted parties, they do not require physical security, such as brick enclosures, guards, etc.

The design of authenticated data structures should address the following goals:

- *Low computational cost:* the computations performed internally by each party (source, responder and user) should be simple and fast; also, the memory space used by the data structures supporting the computation should be as small as possible;
- *Low communication overhead:* source-to-responder communication (updates and update authentication information) and responder-to-user communication (answer and answer authentication information) should be kept as small as possible;
- *Security:* the authenticity of the data provided by a responder should be verifiable with a high degree of reliability.

Cost parameters that measure the performance of an authenticated data structure are:

1. space used by the data structures maintained by the source, responder, and user;
2. time spent by the source and responder to perform an update initiated by the source;
3. size of the update authentication information sent by the source after an update (source-to-responder communication);
4. time spent by the responder to answer a query and compute the answer authentication information, as the proof of the answer;

5. size of the answer authentication information sent by the responder along with the answer (responder-to-user communication);

6. time spent by the user to verify (accept or reject) the answer to a query.

The complexity of authenticated data structures is addressed in more detail in Section 3.4. A formal definition of authenticated data structures is provided in Section 5.5.

2.1.2 Previous and Related Work

Previous work related to authenticated data structures was initially motivated by its applications to the *certificate revocation* problem in public key infrastructure (e.g., [1, 17, 46, 69, 72, 96, 103]), where the underlying problem involves the authentication of (non-)membership in sets. Therefore, this work is mostly concerned with *authenticated dictionaries*, which are authenticated structures for data sets on which membership queries are performed.

The *hash tree* scheme introduced by Merkle [94, 95] can be used to implement a static authenticated dictionary. A hash tree T for a set S is a (balanced) binary tree that stores hash values which are recursively computed over the elements of S using a collision-resistant hash function as follows: the leaves of T store hashes of the elements of S and each internal node v of T stores value h_v, which is the result of hashing over the hash values of v's children. The authenticated dictionary for S consists of the hash tree T computed over the ordered elements of S, plus the signature of the value h_r stored at the root r of T. An element e is proven to belong in S by considering the path in T from the leaf storing e to the root r and reporting the hashes stored at the nodes in T that have siblings on this path. With this approach, space is linear, and the query authentication information and the query and verification time are logarithmic in the size of S. Kocher [72] also advocates a static hash tree approach for realizing an authenticated dictionary, but simplifies somewhat the processing done by the user to verify that an item is not in set S, by storing at the leaves intervals instead of individual elements. Other certificate revocation schemes based on variations of hash trees are described in [17, 46].

Naor and Nissim [103] use techniques that allow dynamic operations on hash trees and support element insertion and deletion in logarithmic time, thus implementing a dynamic authenticated dictionary. In their scheme, the source and the responder maintain identically-implemented 2–3 trees (Seidel's randomized search trees). The update authentication information has $O(1)$ size and the answer authentication information has $O(\log n)$ size. Goodrich and Tamassia [52] present an authenticated dictionary based on other randomized structures, the skip-lists [121]. They introduce the notion of commutative hashing

and show how to embed in the nodes of a skip-list a computational DAG (directed acyclic graph) of cryptographic computations based on commutative hashing. This data structure matches the asymptotic performance of the Naor-Nissim approach [103], while simplifying the details of an actual implementation of a dynamic authenticated dictionary. In related works, Goodrich *et al.* [58] present the software architecture and implementation of an authenticated dictionary based on the above approach and Anagnostopoulos *et al.* [2] introduce the notion of *persistent authenticated dictionaries*, where the user can issue historical queries of the type "was element e in set S at time t?". Work related to the issue of persistence and historical queries appears in [83, 84].

Goodrich *et al.* [57] show how to use the RSA one-way accumulator [4, 5] to realize a dynamic authenticated dictionary for a set of n elements with $O(1)$ query authentication information size and verification time. Their scheme allows a tradeoff between the query and update times. For example, one can balance the two time complexities and achieve $O(\sqrt{n})$ query and update time and $O(\sqrt{n})$ update authentication information. Related work appears by Camenisch and Lysyanskaya in [18], where a dynamic accumulator is designed based on the RSA strong assumption. Accumulators have been studied in [4, 5, 18, 105].

A first step towards the design of more general authenticated data structures (beyond dictionaries) is made by Devanbu *et al.* [33, 34]. Using an extension of hash trees, they show how to authenticate operations *select*, *project* and *join* in a relational database. Moreover, they present an authenticated data structure for a set of multidimensional points that supports orthogonal range queries. This latter result goes beyond simple authenticated dictionaries, but it is restricted to hashing in range trees. More recent schemes for authenticating SQL queries in the related model of *outsourced database systems* (essentially, for authenticating range queries over indexes of databases published at remote sites) have been proposed by Li *et al.* [78], Mykletun *et al.* [102] and Nuckolls [106]. Also, related work includes the authentication of XML documents by Devanbu *et al.* [32] and Bertino *et al.* [7].

Martel *et al.* [86] initiated a study of authenticated queries beyond tree structures and skip-lists. They consider the class of data structures in the pointer machine model, such that (i) the links of the structure form a directed acyclic graph G of bounded degree with a single source node, and (ii) queries on the data structure correspond to a traversal of a subdigraph of G starting at the source. They show that such data structures can be authenticated by means of a hashing structure that digests the entire digraph G into a hash value at its source. With this scheme, the size of the answer authentication information and the verification time are proportional to the size of the subdigraph traversed. Thus, this general method for designing authenticated data structures applies to the pointer machine model

for the above class of search problems, essentially authenticating the entire query answering algorithm. They show how this technique can be applied to the design of static authenticated data structures for pattern matching in tries and for orthogonal range searching in a multidimensional set of points. They also begin an initial treatment of authenticating fractional cascading structures, but only for range-tree data structures, where catalogs are arranged as unions in a tree.

Recently, the study of an additional security property for data querying that is related to authenticated data structures has been initiated. Assuming a more adversarial for the user setting, consider the case where the data source can act unreliably. Then, a new requirement is data consistency, namely, the inability of the source (and, thus, also of the responder) to provide different, i.e., contradictory, verifiable answers to the same query. Buldas *et al.* [17] study this issue for hash trees and show how to enforce data consistency by augmenting hash trees. Micali *et al.* [98] introduce zero-knowledge sets, where a prover commits to a value for a set and membership queries can be verified by a verifier consistently. Ostrovsky *et al.* [108] extend consistency proofs to range queries and also give sufficient conditions for schemes to achieve consistency. The works in [98, 108] additionally provide privacy-preserving verification but involve computationally more expensive operations.

2.1.3 Contributions

In this chapter, we present general techniques for building authenticated data structures for a broad class of query problems on graphs and geometric objects.

In particular, we describe an authentication scheme for structured data that represents a general graph G and a generic type of queries that are related to properties about paths in G. This, in turn, allows the authentication of a broad class of graph queries, including the following, where v, w are nodes of graph G:

- areConnected(v, w): Are v and w in the same connected component?
- areBiconnected(v, w): Are v and w in the same biconnected component?
- areTriconnected(v, w): Are v and w in the same triconnected component?
- path(v, w): Return a path from v to w, if any.
- pathLength(v, w): Return the length of the path connecting v to w, if any.

Our scheme also supports efficient update operations that involve insertions of nodes and edges in G. For graphs of n nodes, our authenticated data structure uses linear space and

supports update operations and connectivity queries in $O(\log n)$ time and path queries in $O(\log n + k)$ time, where k is the length of the reported path. The update authentication information has $O(1)$ size. Similarly, the size of the answer authentication information and the verification time are each $O(\log n)$ for connectivity queries and $O(\log n + k)$ for path queries. If the graph is planar, the data structure is fully dynamic and supports arbitrary series of intermixed insertions and deletions of nodes/edges. For general graphs, the data structure supports insertions but no deletions. These results have applications to the authentication of network management systems.

In addition, we address several geometric search problems by showing how to authenticate the full, general version of the powerful *fractional cascading* technique [23], which supports efficient searching in multi-catalogs. Our technique provides a general framework for designing authentication schemes for any data structure built using the fractional cascading technique. In particular, we can efficiently authenticate any query for the *iterative search problem*, where we have a collection of k catalogs (i.e., dictionaries) of total size n, stored at the nodes of a connected graph, and we want to search for an element in each catalog in a connected subgraph of this graph. Fractional cascading yields a data structure of linear space that supports iterative search in $O(\log n + k)$ time, thus, clearly outperforming the $O(k \log n)$ time that results when performing k separate searches or the $O(kn)$ space that results when searching in a merged master-dictionary. A number of two-dimensional geometric searching problems—related to many data management problems—can be solved with data structures based on fractional cascading [24]. These problems include:

- *line intersection queries* on a polygon P, to report the edges of P intersected by a query line;
- *ray shooting queries* on a polygon P, to report the first edge of P intersected by a query ray;
- *point location* on a planar subdivision, to report the region containing a query point;
- *orthogonal range search* on a set of points in $\mathbf{R}^2$, to report the points inside a query rectangle;
- *orthogonal point enclosure* on a set of rectangles, to report the rectangles that contain a query point;
- *orthogonal intersection queries* on a set of rectangles, to report the rectangles intersected by a query rectangle.

Our authenticated fractional cascading data structure can be extended to yield efficient authenticated data structures for all the above problems. Our authentication schemes have applications to database management and geographic information systems. No authentication schemes for the above types of queries had been previously known, with the exception of orthogonal range search, for which an authenticated data structure is given in [86].

Our authenticated data structures are based on a new hash-based authentication scheme, the *path hash accumulator*, designed for efficiently authenticating various properties of structured data that is represented as paths. The path hash accumulator supports the authentication of any decomposable query over paths and, in particular, the authentication of queries about sequences of elements (or data of topological structure), where the queries are related to applying any associative operator over the data elements. This class of queries includes data aggregation queries and searching over ordered sets. Our authentication primitive can be considered an extension of Merkle's hash tree [95], supporting authentication of a rich set of properties of data stored in the tree. By building on the hash path accumulator, we achieve efficiency and modularity: our schemes are simple to implement and easy to analyze in terms of complexity and security. Overall, by supporting authentication of the abstract class of property queries over paths in graphs and of the iterative search problem, our authentication schemes can be used a general-purpose tool in the design of more complex authenticated data structures.

The efficiency of our authenticated data structures asymptotically matches the efficiency of the corresponding data structure used to answer the query. However, our hash-based authentication techniques use a "bottom-up" hashing approach, where the answer verification is authenticated rather than the query answering process. In particular, as opposed to the "top-down" hashing approach in [86], where the exact traversal of the data structure and the exact data processing at any visited node is being sequentially authenticated, in our authentication framework data is being authenticated on a per-answer basis. That is, although the data structure is used to facilitate the way data is hashed, an answer is generally verified independently of its searching access path. This way we achieve more efficiency, since only information that is necessary for the verification purpose is being authenticated; not all the information stored in the data structure.

The security of our schemes is based on standard cryptographic primitives, such as collision-resistant cryptographic hashing and digital signatures; hence, our schemes are practical and do not need any new cryptographic assumptions.

2.1.4 Chapter Structure

The chapter is organized as follows. In Section 2.2, we define our data authentication model, state our cryptographic assumptions and present the general authentication framework that we use. In Section 2.3, we present the path hash accumulator, an authentication scheme for properties on paths and associative queries on sequences. In Section 2.4, we present authenticated data structures for various path and connectivity queries on graphs. In Section 2.5, we present an authentication scheme for the fractional cascading algorithmic paradigm, which leads to the authentication of various geometric data structures. We conclude in Section 2.6.

2.2 Hash-Based Data Authentication

In this section, we present the general technique that is used in our authenticated data structures and discuss the security requirements related to our authentication model.

Let $S = \{e_1, \ldots, e_n\}$ be a data set owned by the source and let $\mathcal{Q}$ be the set of all possible queries (of some type) that a user can issue about S. In an extreme solution, the source can just digitally sign the answer to every possible query $q \in \mathcal{Q}$, that is, pairs of the form (q, a) denoting that a is the answer to query q. Since $\mathcal{Q}$ can be infinitely large or it can be a dynamic set that frequently changes over time, this solution is almost always not viable. Thus, directly applying well-known message authentication techniques for data authentication (e.g., signing query-answer pairs or whatever piece of information needs authentication) is not suitable *per se*, but rather, additional machinery is needed.

Alternatively, consider the following solution concept. Let us assume that $\mathcal{C}(S)$ is a set of statements $s_1, \ldots, s_c$ that completely describe data set S with respect to queries in $\mathcal{Q}$, meaning that the answer a to any query $q \in \mathcal{Q}$ can be securely certified to be authentic by providing some minimal (with respect to q) subset $A(q, a) \subset \mathcal{C}(S)$ of such statements to the user and by proving that the statements in $A(q, a)$ are valid. If such a set $\mathcal{C}(S)$ exists, then, intuitively, the source can authenticate only this set, for instance by signing all statements in $\mathcal{C}(S)$ and providing these signed statements to the responder, so that they are appropriately forwarded to the user along with the answer to a query. Of course, for efficiency, set $\mathcal{C}(S)$ should be chosen to have the smallest possible size. For instance, if S represents an ordered sequence of n elements and $\mathcal{Q}$ is the set of all one-dimensional range queries about S (i.e., "report elements of S in range $[x_1, x_2]$"), then $\mathcal{C}(S)$ could consist of statements of the form "e_{i+1} is the successor of e_i in S".

Our approach, which constitutes the state-of-the-art solution for data authentication and is common in concept in most works on authenticated data structures [103, 141], is *signature amortization*. The idea is to compute a *digest* of data set S, that is, a short, secure cryptographic description of S, and to have this digest be the only statement signed by the source[1]. This signature is forwarded to the responder. On a query, along with the answer, the user is provided by the responder with this signed digest and with some auxiliary information (a proof) that is sufficient for the computation of the digest and the verification of the answer. In this way, one signature is amortized over all queries about S.

But what is being signed? The digest of S should, essentially, securely represent a corresponding set of statements $\mathcal{C}(S)$ that can provide secure answer verification for the type of queries that are issued. In particular, the digest should be computed such that it carries certain cryptographic properties and encodes certain structural properties of S, that allow the authentication of set $\mathcal{C}(S)$ (through the secure transmission of trust from the digest to the elements of $\mathcal{C}(S)$, or S), independently of the behavior of the responder, and the efficient verification of the answer, independently of the specific query.

In this chapter, we consider the case where the digest is computed using collision-resistant hashing. We call this authentication technique *hash-based data authentication*. We note that the use of one-way accumulators (as in [18, 57]) to produce the digest constitutes an alternative—though, in practice computationally more expensive—approach. We next describe more formally our approach, starting from the cryptographic primitives in use.

2.2.1 Cryptographic Primitives

In our authentication model the basis of trust is the assumption that the user trusts the data source. In the public-key cryptographic model this is expressed by means of a *digital signature scheme*. That is, the user knows the public key of the source and trusts that anything signed under the corresponding private key is authentic. For completeness we present the standard definition of the signature-scheme primitive as in [50]. Schemes that satisfy the following security requirement are known as signature schemes secure against *adaptive chosen-message attack*. A function $\nu : \mathbb{N} \rightarrow \mathbb{R}$ is *negligible* if for every positive polynomial $p(\cdot)$ and for sufficiently large k, $\nu(k) < \frac{1}{p(k)}$.

Definition 2.2.1 (Signature Scheme)**.** *The triplet of probabilistic polynomial-time algorithms* $(G(\cdot), \mathsf{Sign}_{(\cdot)}(\cdot), \mathsf{Verify}_{(\cdot)}(\cdot,\cdot))$*, where* G *is the key generation algorithm producing a*

[1] We note that in [86] the data digest is not signed (not authenticated); it is, instead, assumed to be available to the users, through a registration phase with the source. Since only static data is considered, this registration phase is performed only once and, thus, is not problematic.

pair (PK, SK) *of public and secret keys on input a security parameter* k, Sign *the signature algorithm, and* Verify *the verification algorithm, constitute a digital signature scheme for a family (indexed by the public key* PK*) of message spaces* $\mathcal{M}_{(\cdot)}$ *if the following two hold:*

Correctness *If a message* m *is in the message space for a given public key* PK*, and* SK *is the corresponding secret key, then the output of* $\mathsf{Sign}_{SK}(m)$ *will always be accepted by the verification algorithm* Verify_{PK}*. More formally, for all values* m *and* k*:*

$$\Pr[(PK, SK) \leftarrow G(1^k); \sigma \leftarrow \mathsf{Sign}_{SK}(m) \ : \ m \leftarrow \mathcal{M}_{PK} \wedge \neg \mathsf{Verify}_{PK}(m, \sigma)] = 0.$$

Security *Even if an adversary has oracle access to the signing algorithm that provides signatures on messages of the adversary's choice, the adversary cannot create a valid signature on a message not explicitly queried. More formally, for all families of probabilistic polynomial-time oracle Turing machines* $\{A_k^{(\cdot)}\}$*, there exists a negligible function* $\nu(k)$ *such that*

$$\begin{aligned}\Pr[(PK, SK) \leftarrow G(1^k); (Q, m, \sigma) \leftarrow A_k^{\mathsf{Sign}_{SK}(\cdot)}(1^k) : \\ \mathsf{Verify}_{PK}(m, \sigma) = 1 \wedge \neg(\exists \sigma' \mid (m, \sigma') \in Q)] \quad = \quad \nu(k).\end{aligned}$$

The hash-based data authentication technique makes use of a *cryptographic hash function* to produce the digest of the data set. A cryptographic hash function h operates on a variable-length message M producing a hash value $h(M)$ of short and fixed length. Moreover, a cryptographic hash function h is called *collision-resistant*, if it is infeasible to find two different strings $x \neq y$ that hash to the same value, i.e., form a collision $h(x) = h(y)$. For completeness, we give a standard definition of a family of collision-resistant hash functions.

Definition 2.2.2 (Collision-resistant Hash Function)**.** *Let* $\mathcal{H}$ *be a probabilistic algorithm that, on input* 1^k*, runs in polynomial time and outputs an algorithm* $h \ : \ \{0,1\}^* \mapsto \{0,1\}^k$*. Then* $\mathcal{H}$ *defines a family of collision-resistant hash functions if:*

Efficiency *For all* $h \in \mathcal{H}(1^k)$*, for all* $x \in \{0,1\}^*$*, it takes polynomial time in* $k + |x|$ *to compute* $h(x)$*.*

Collision-resistance *For all families of probabilistic polynomial-time Turing machines* $\{A_k\}$*, there exists a negligible function* $\nu(k)$ *such that*

$$\Pr[h \leftarrow \mathcal{H}(1^k); (x_1, x_2) \leftarrow A_k(h) \ : \ x_1 \neq x_2 \wedge h(x_1) = h(x_2)] = \nu(k).$$

2.2.2 Hashing Scheme and Authentication Framework

We now describe the general authentication framework which our authentication schemes are based on. Using the public-key cryptographic model, given a security parameter, a signature scheme and a collision-resistant hash function h are available for use to the source and the user. As a result, there is always available information that allows the user to validate a signature produced by the source.

Let $S = \{e_1, \ldots, e_n\}$ be a data set owned by the source. To achieve signature amortization, what is signed by the source is a digest d of the data elements of S. In our authentication schemes, the collision-resistant hash function h is used to produce this digest of S. To achieve this, we assume some well-defined binary representation for any data element e of S, so that h can operate on e and produce hash value $h(e)$. Also, we assume that rules have been defined so that h can operate over any finite sequence of elements. That is, $h(e_{i_1}, \ldots, e_{i_k})$ represents a hash value computed over elements $e_{i_1}, \ldots, e_{i_k}$. For instance, $h(e_{i_1}, \ldots, e_{i_k})$ can denote that h operates on the concatenation $e_{i_1} \| \ldots \| e_{i_k}$ of $e_{i_1}, \ldots, e_{i_k}$, or on the concatenation $h(e_{i_1}) \| \ldots \| h(e_{i_k})$ of their hashes; in both cases, h essentially operates on a binary string s, where the cost of computing $h(s)$ is proportional to the length of s.

We next describe the notion of *hashing scheme*, our general approach for computing in a systematic way a digest d over the data elements of S. In our approach, the computation of a digest d of S is expressed by means of a single-sink directed acyclic graph (DAG) defined over S. Nodes of DAG G are associated with the data elements in S and are labeled with hash values.

Definition 2.2.3 (Hashing Scheme)**.** *Let $S = \{e_1, \ldots, e_n\}$ be a data set and let G be a single-sink directed acyclic graph. A* hashing scheme *for S using G is a node-labeling in G created as follows. Sequences of data elements of S are associated with nodes of G and each node $u \in G$ is assigned a* label *$L(u)$, a hash value, such that:*

- *if u is a source node of G and $e_{u_1}, \ldots, e_{u_m}$ the are elements of S associated with node u, where m is some constant integer, then*

$$L(u) = h(e_{u_1}, \ldots, e_{u_m}), \text{ otherwise}$$

- *if u is a non-source node of G, edges $(z_1, u), \ldots, (z_k, u)$ exist in G and $e_{u_1}, \ldots, e_{u_m}$ are the elements of S associated with node u, where m and k are some constant integers, then*

$$L(u) = h(e_{u_1}, \ldots, e_{u_m}, L(z_1), \ldots, L(z_k)).$$

If t is the sink node of G, the digest *of S using this hashing scheme is label $L(t)$.*

We often use the term hashing scheme we refer to the augmented graph G, including the association between data elements and graph nodes and the hash values, not only to the node-labeling of G. Observe that we assume that each node of the DAG G is associated with a constant number of data elements and that also any node in G has constant in-degree. Also note that, as the digest d of S is simply a hash value, it has short length. The DAG G is in general defined in accordance with the data structure used to answer queries about S, but it does not necessarily coincide with its (linking) structure.

Given a data set S and a signature scheme, signature amortization is implemented using the following *authentication scheme*. Let $\mathcal{Q}$ be the set of all possible queries q of a certain type that the user can issue about S. Using a hashing scheme that is specially designed for the query type $\mathcal{Q}$, the data source produces a digest d of S and, using a signature scheme, the source signs d. Once d is signed (and is forwarded to the user by the responder), it is used as the basis of trust for authenticating any query on S. That is, the answer to any query $q \in \mathcal{Q}$ about S is tested against the validity of the signed digest d. This is achieved using the collision-resistant property of h and the way digest d is computed through the hashing scheme G. In particular, G is constructed such that it expresses relations and structural information about S that corresponds to a set of statements $\mathcal{C}(S)$ capable of verifying the answers to queries in $\mathcal{Q}$, and, when the signed digest d is verified to be authentic, it is the collision-resistance property of h that transmits trust to the data elements of S—from the authentic d and through the graph G—essentially, authenticating the set $\mathcal{C}(S)$. Information encoded in this set is finally used to check the validity of the answer provided by the responder. The hashing scheme should be designed such that, depending on the type $\mathcal{Q}$ of queries answered, checking the validity of answers is feasible in a secure, correct and efficient manner, and independently of the corresponding specific query q.

For dynamic data, new digests are computed and signed by the source, as data evolves over time. To avoid *replay attacks* launched by the responder, that is, attempt for verification of answers subject to old, invalid data digests (that can be easily cached by the responder), the technique of *time-stamping* is used, as it was introduced in [103]. A digest is signed after a time-stamp is appended to it, which is used by the user to check the freshness of the signature on the digest. A verifiable answer is finally accepted only if it corresponds to a fresh signature, that is, only if the time-stamp is recent (according to some convention depending on the higher level application). Accordingly, the source periodically resigns the current digest, even if no changes have occur in the data set.

Overall, our authentication techniques presented in this chapter are based on the following general protocol. In the model of authenticated data structures, the source and the responder store identical copies[2] of the data structure representing S and maintain the same hashing scheme G on S. The source periodically signs the digest of S together with a time-stamp and sends the signed time-stamped digest to the responder. When updates occur on S, they are sent to the responder together with the new signed time-stamped digest. Note that, in this setting, the update authentication information has $O(1)$ size and the structure authentication information consists only of the digest.

When the user poses a query, the responder returns to the user the answer along with the answer authentication information, i.e., it returns (i) the answer to the query, (ii) the signed time-stamped digest of S and (iii) a proof, consisting of a small collection of labels or data elements from the hashing scheme G that allows the recomputation of the digest and the semantic verification of the answer. The user validates the answer by recomputing the digest (and checking its correctness as it is expressed by the hashing scheme and the corresponding set $\mathcal{C}(S)$), checking that it is equal to the signed one and verifying the signature on the digest. Accordingly, the user either verifies the authenticity of the answer and *accepts* it as authentic, or otherwise, the user *rejects* the answer. The total time spent for this process is called the *answer verification time*.

Note that, at the user side, the verification algorithm operates on the three inputs: the answer, the proof and signed digest. The answer and the proof are used to recompute the digest. In doing this, the user employs the collision-resistant hash function h in combination with the hashing scheme G. Both h and the structure of G are assumed to be available to the user as part of the public key (recall that we work on the public-key cryptographic model). Alternatively, we can think the subgraph of G used by the user to be part of the proof. The hashing scheme G encodes set $\mathcal{C}(S)$—necessary semantic information about data set S—and is the means by which the user associates the answer with the proof and the issued query q and finally verifies the answer.

We note that our authentication framework described above is general and appropriate for any type of queries $\mathcal{Q}$. In particular, Definition 2.2.3 of hashing schemes imposes no restriction on the exact structure of the used DAGs. As we will see next, the design of efficient and secure authenticated data structures is based on the correct and careful definition of a hashing scheme G in accordance with the type of queries $\mathcal{Q}$.

[2]When randomized data structures are in consideration, identical copies can be still maintained by having the source and the responder sharing the same randomness seed.

Remark 2.2.1. *For static search query problems, a similar technique to ours for computing the data digest is presented in [86] by Martel et al., where again a DAG is used to facilitate the hash-based authentication. However, in [86] the DAG in use coincides with the search DAG of the underlying data structure and a top-down hashing approach is used, such that, essentially, the traversal of the data structure due to a query rather than the answer to this query is authenticated. This top-down step-by-step authentication technique generally leads to less efficient authentication schemes.*

2.2.3 Security

Finally, we discuss the security requirement for any authentication scheme and how it is achieved using the above authentication method. Our authentication techniques follow the standard "hash and sign" authentication paradigm. Here we present a security definition appropriate for the model of authenticated data structures.

Starting from the basis that in the model of authenticated data structures the user trusts the data source but not the responder, it is the responder that can act adversarially. We first assume that the responder always participates in the three-party protocol, i.e., it communicates with the source and the user, as the protocol dictates. Thus, we do not consider denial-of-service attacks; they do not form an authentication attack but rather a data communication threat. Actually, although practically nothing prevents the responder from denying to participate in the protocols, e.g., it may refuse to respond to a user's query, in principle nothing prevents the user from redirecting the query to another responder. Indeed, a practical deployment of authenticated data structures utilizes responders as geographically distributed—widely spread in a network—mirror sites of the source, thus, the user can contact a different responder. Additionally, denial-of-service attacks could be prevented using some form of penalties applied to non-cooperative responders (e.g., charges may be applied in an answer is not given).

However, a responder can try to cheat, by not providing the correct answer to a query but attempting to forge a fake proof for a false answer. We model this scenario by assuming that the responder is controlled by a polynomial-time adversary $\mathcal{A}$. The adversary performs a type of attack that is similar to a adaptive chosen-message attack for signature schemes. That is, $\mathcal{A}$ has oracle access to the authentication technique and possesses the signed digest (using a particular hashing scheme) of any data set S' of his choice. Then, given a particular query $q \in \mathcal{Q}$ for a data set S, the goal of the adversary is to construct a false answer and a fake proof for this query that passes the verification check performed at the user.

Accordingly, the security requirement that the authentication scheme of any authenticated data structure should satisfy is as follows: given any query by a user, no polynomial-time responder can reply with a pair of answer and an associated proof, such that both the answer is not correct and the user (incorrectly) verifies the authenticity of the answer and accepts it. More formally, the basic security requirement that any authenticated data structure should satisfy is enforcing the following property.

Definition 2.2.4 (Security Requirement)**.** *An authenticated scheme for an authenticated data structure is* secure*, if for any query issued by a user, no polynomial-time adversary $\mathcal{A}$—controlling the responder that answers the query and having oracle access to the authentication scheme—has non-negligible on a security parameter advantage in causing a user to accept, i.e., to verify as correct, an incorrect answer.*

For hash-based data authentication, the above property is achieved by relying on the security properties of signatures and collision-resistant hashing. This means that, if the hashing scheme is carefully designed such that it encodes statements about the data set S (set $\mathcal{C}(\mathcal{S})$) that allow answer verification, then the authentication scheme is secure: in a standard way, any attack against the scheme is effectively reduced to an attack either on the signature scheme or on the collision-resistant hash function in use. Like efficiency, the security of an authentication scheme is characterized by the design of the hashing scheme.

We close our discussion by noting that for search query problems, where the answer to a query is a subset of the data elements in S, we can further characterize the properties that the underlying hashing scheme should satisfy. Let $a(q)$ be the answer of a search query $q \in \mathcal{Q}$ for a data set S. If $a(q)$ contains elements $e_{a_1}, \ldots, e_{a_l}$ and answer $a'(q) = \{e_{a'_1}, \ldots, e_{a'_k}\}$ is given by the responder to the user, then the hashing scheme G should be chosen such that it can be used to check whether $a'(q)$ is correct. And in this case, answer $a'(q)$ is said to be correct, if it is both *sound*, i.e., it contains only elements that satisfy the query parameters of q, and *complete*, i.e., it contains all elements that satisfy the query parameters of q, that is, if $a'(q) = a(q)$ (see also [86]). According to this definition, a correct answer is unique.

2.3 Authenticated Path Properties

We now present our first authentication scheme, the *path hash accumulator*, which is a general authentication scheme used to provide authentication for various types of queries on a data set S. Here S is a sequence $(e_1, e_2, \ldots, e_n)$, i.e., a collection of n elements, where the notion of *predecessor* and *successor* are defined on elements of S and the notion of

first and *last* are defined on S. Our path hash accumulator will serve as an authentication primitive used in the rest of the chapter (Sections 2.4 and 2.5). We start by introducing some notation related to paths, the central technical concept in this chapter.

2.3.1 Paths and Path Properties

An abstract notion of a *path* is used to represent S. We use and extend the notation used in [25]. A *path* consists of one or more *nodes* and is always directed. This is in accordance with the need for capturing the predecessor-successor relationship. More formally:

Definition 2.3.1 (Path)**.** *A* path *p is an ordered sequence of one or more nodes. The first and last nodes of a path p are called the* head *and* tail *of p, and are denoted as $head(p)$ and $tail(p)$. A path is considered to have a direction: each node is connected to its successor by a directed edge.*

Paths can be joined to form a *concatenation path* and we also use the notion of *subpath*.

Definition 2.3.2 (Subpaths)**.** *If p' and p'' are paths, the* concatenation *$p = p' \| p''$ is a path formed by adding a directed edge from $tail(p')$ to $head(p'')$. A* subpath *$\bar{p}(v, u) = \bar{p}$ of a path p is the path consisting of the collection of consecutive nodes of p $v, w_1, \ldots, w_l, u$ with $head(\bar{p}) = v$, $tail(\bar{p}) = u$.*

The data set S is associated with a path through the notion of *node attributes* and *node properties* that are stored in the nodes of the path.

Definition 2.3.3 (Node Attribute)**.** *A* node attribute *$N(v)$ of node v is a value related to and stored at v. $N(v)$ can assume arbitrary values and occupies only $O(1)$ storage. A* node property *$\mathcal{N}(v)$ of node v is a sequence $N_1(v), \ldots, N_r(v)$ of node attributes, where r is a constant. For a node v, we require that v (or some id representing v) is included in any node property $\mathcal{N}(v)$ of v as a node attribute of v.*

Similarly, *path attributes* and *path properties* are defined to extend the notion of node attribute and node property for a collection of consecutive nodes, i.e., for paths.

Definition 2.3.4 (Path Attribute)**.** *A* path attribute *$P(p)$ of path p is a value that is related to p and occupies only $O(1)$ storage. A* path property *$\mathcal{P}(p)$ of p is a sequence of path attributes $P_1(p), \ldots, P_s(p)$, where s is a constant. For a path p, we require that $head(p)$ and $tail(p)$ are included in any path property $\mathcal{P}(p)$ of p as path attributes of p.*

The path attributes (and thus the path properties) of a path depend on the node properties of the path and possibly on the structural properties of the path (e.g., path size, node ordering etc.). The definition of path attributes and path properties are naturally extended when subpaths of paths are considered. In this setting and omitting some of the details (see Remark 2.3.1, end of section), we can view a path property as a *mapping* $\mathcal{P}$ from paths (and, actually, node properties of their nodes) to sequences of values.

We are interested in path properties that satisfy the *concatenation criterion*.

Definition 2.3.5 (Concatenation Criterion)**.** *Let p be a path and let p' and p'' be any subpaths of p such that $p = p' \| p''$. A path property $\mathcal{P}$ satisfies the* concatenation criterion *if $\mathcal{P}(p) = \mathcal{F}(\mathcal{P}(p'),\mathcal{P}(p''))$, where $\mathcal{F}$ is a function defined on pairs of sequences of values (path attributes) that can be computed in $O(1)$ time. Function $\mathcal{F}$ is called the* concatenation function *of $\mathcal{P}$.*

In other words, a path property satisfies the concatenation criterion when this path property evaluated for a path p can be computed in constant time given the corresponding path properties of any two paths whose path concatenation equals the path in consideration p. Overall, a path property satisfying the concatenation criterion with its corresponding concatenation function admits a computational evaluation that is inherently associative.

We wish to be able to locate nodes of a path that are of our interest. This is achieved by a *node selection query* by means of a *path selection function*. Analogously, a *path selection query* extends a node selection query using a *path advance function*.

Definition 2.3.6 (Node- and Path-Selection Queries)**.** *Let $\mathcal{P}$ be a path property that satisfies the concatenation criterion. Given a path p and a query argument q:*

- *a* node selection query *$NODE_{\mathcal{P}}$ maps p into a node $v = NODE_{\mathcal{P}}(p, q)$ of p; a node selection query is always associated with some* path selection function*: given that $p = p' \| p''$, a* path selection function *$\sigma(p, q)$ for $NODE_{\mathcal{P}}$ determines in $O(1)$ time whether v is in p' or p'' using q and values $\mathcal{P}(p')$ and $\mathcal{P}(p'')$; and*

- *a path selection query $PATH_{\mathcal{P}}$ maps p into a subpath $\bar{p} = PATH_{\mathcal{P}}(p, q)$ of p; a path selection query is always associated with some* path advance function*: given that $p = p'|p''$, a* path advance function *$\alpha(p, q)$ for $PATH_{\mathcal{P}}$, using values $\mathcal{P}(p')$ and $\mathcal{P}(p'')$, returns in $O(1)$ time the subpath(s) among p', p'' (possibly none) for which the query argument q holds.*

Let p be a path, $\mathcal{P}$ be any path property that satisfies the concatenation criterion and let $\mathcal{N}$ be any node property. We are interested in authenticating the following query operations on p:

- property(subpath $\bar{p}(v,u)$): report the value of path property $\mathcal{P}$ for subpath $\bar{p}(v,u)$ of p ($\bar{p}$ may be equal to p);
- property(node v): report the value of node property $\mathcal{N}$ for node v;
- locate(path p, path selection function σ, argument q): find node v of p returned by the node selection query expressed by the path selection function σ;
- subpath(path p, path advance function α, argument q): find the subpath of p returned by the path selection query expressed by the path advance function α.

Remark 2.3.1. *A path property is a sequence of path attributes, that is, a sequence of values that relate to the path. In particular, a path attribute depends on the data stored at the nodes of the path as node properties, i.e., as sequences of node attributes, as well as possibly on structural properties of the path (e.g., the size of the path, ordering of nodes etc.). Accordingly, and given the fact that the definition of path properties is naturally extended to subpaths of paths, we can view the path property $\mathcal{P}(p)$ of a path p as a mapping from p to a sequence of values related to p (with respect both its structure and its data stored). Thus, we can treat this path property as a function $\mathcal{P}(\cdot)$ or refer to it simply as $\mathcal{P}$. Also, note that a node property $\mathcal{N}(u)$ can be viewed as a corresponding path property $\mathcal{P}(\bar{p}(u,u))$ or vice versa.*

2.3.2 Path Hash Accumulator

We now present our first authentication scheme for the above query operations on paths, discussing the details of the path representation and its associated hashing scheme. Let $\mathcal{P}$ and $\mathcal{N}$ be the path property satisfying the concatenation criterion and the node property of our interest in terms of authentication.

We represent a path p as a balanced binary tree $T(p)$ as follows. A leaf of $T(p)$ represents a node of p, such that the left-to-right ordering of the leaves of $T(p)$ correspond to path p. An internal node v of $T(p)$ represents the subpath $p(v)$ of p associated with the leaves in the subtree of v. When referring to a leaf node, we do not distinguish between the node of the path and the leaf of the tree. Each leaf u stores the corresponding node property $\mathcal{N}(u)$ and each non leaf node v stores the corresponding path property $\mathcal{P}(p(v))$.

Let h be a collision-resistant hash function. The *path hash accumulator* for a path p is the hashing scheme for the node and path properties of p using a DAG induced by tree $T(p)$. Specifically, the hashing scheme is defined as follows. Consider the data set consisting of: (1) for each leaf node v of $T(p)$, the node property $\mathcal{N}(v)$ and (2) for each internal node u of $T(p)$, the path property $\mathcal{P}(p(u))$, where $p(u)$ is the subpath of p associated with the leaves in the subtree of u. For what follows, we denote the path property $\mathcal{P}(p(u))$ of the subpath defined by node u simply as $\mathcal{P}(u)$. Let G be the DAG obtained from $T(p)$ by directing each edge towards the parent node. For a node v of p, let $pred(v)$ and $succ(v)$ denote the predecessor and the successor of v in p, respectively. In particular, $pred(head(p))$ and $succ(tail(p))$ are some special (nil) values. Using G and h, we compute a label $L(u)$ for each node u of $T(p)$ as follows:

- If u is a source vertex of G, i.e., a leaf of $T(p)$, then

$$L(u) = h(\mathcal{N}(pred(u)), \mathcal{N}(u), \mathcal{N}(succ(u))); \tag{2.1}$$

- If u is a non source vertex of G and (z_1, u) and (z_2, u) are edges of G, then

$$L(u) = h(\mathcal{P}(u), L(z_1), L(z_2)). \tag{2.2}$$

The digest of the above data set is the label $L(r)$ of the sink r of G (i.e., r is the root of $T(p)$). This digest is called the *path hash accumulation* of path p.

We are interested in supporting the following two update operations on paths. Operation concatenate(path p', path p'') joins paths p' and p'' to the concatenation path $p = p'\|p''$ and operation split(node v) splits the path p that contains v in two subpaths p' and p'' such that $p = p'\|p''$ and $v = head(p'')$. These primitive operations are used in Section 2.4 to support more complex update operations of data structures built using path hash accumulators[3] and they cause the recomputation of the path hash accumulation of the paths involved in these operations.

We next present and prove our first result about the performance and the authentication properties of path hash accumulators, where we view these structures as data structures used in the three-party authentication model described in Section 2.1.1.

Lemma 2.3.1. *Let p be a path of length n. There exists an authenticated data structure for p that is based on the path hash accumulator hashing scheme and supports query operations*

[3] Although the term path hash accumulator is defined as the hashing scheme, i.e., the authentication structure, we use the same term to refer to the corresponding data structure, i.e., the binary tree; since the underlying DAG of the hashing scheme coincides with the binary tree, this brings no confusion.

property(subpath), property(node), locate and subpath and update operations concatenate and split with the following performance:

1. *query operations on p property(subpath), property(node), locate and subpath take each $O(\log n)$ time;*
2. *for every query operation the answer authentication information has size $O(\log n)$;*
3. *for every query operation the answer verification time is $O(\log n)$;*
4. *the total space used is $O(n)$;*
5. *for every update operation the update authentication information has size $O(1)$;*
6. *if q is a path of length m, update operation concatenate(p, q) takes $O(\log(\max\{n, m\}))$ and update operation split on p takes $O(\log n)$ time.*

Proof. (1) First consider the query property$(\bar{p}(v, u))$ on p. Let $\mathcal{A}(\bar{p})$ be the set of *allocation nodes* in $T(p)$ of subpath $\bar{p} = \bar{p}(v, u)$. For a tree node w, $w \in \mathcal{A}(\bar{p})$ if the leaves of the subtree defined by w are all nodes of $\bar{p}$ but the same is not the case for w's parent, if any. That is, $\mathcal{A}(\bar{p})$ is the minimal set of tree nodes defining subtrees that exactly cover $\bar{p}$ and no other nodes of p, i.e., subtrees whose leaves form a partition of $\bar{p}$ into subpaths, where each subpath consists of the leaves that correspond to one of the allocation nodes. So, each $w \in \mathcal{A}(\bar{p})$ corresponds to a subpath of path $\bar{p}$. Since $T(p)$ is a balanced binary tree, there are $O(\log n)$ allocation nodes for subpath $\bar{p}$ that can be found in $O(\log n)$ time by tracing the leaf-to-root tree paths in $T(p)$ from v and u up to the root r of $T(p)$. Since, the path property $\mathcal{P}$ satisfies the concatenation criterion, we have that the path property $\mathcal{P}(\bar{p})$ can be computed by using the tree structure and by applying $O(\log n)$ times the concatenation function $\mathcal{F}$ of $\mathcal{P}$ on the path properties of the subpaths of $\bar{p}$ stored at the allocation nodes of $\bar{p}$. Thus, query property(subpath) can be answered in $O(\log n)$ time.

(2) Clearly, the answer given to the user is the property $\mathcal{P}(\bar{p})$. For any node u of $T(p)$, let $(u_1, \ldots, u_k)$ be the node-to-root path connecting u with the root r, with u_1 being u and u_k being a child of r. We define the *verification sequence* of u to be the sequence $\mathcal{V}(u) = (s_1, s_2, ..., s_k)$, where, for $1 \leq j \leq k$,

$$s_j = (L(\bar{u}_j), \mathcal{P}(\bar{u}_j)), \tag{2.3}$$

and $\bar{u}_j$ is the sibling node of u_j, i.e., s_j is the pair of the label of the sibling $\bar{u}_j$ of node u_j and the path property of the path $p(\bar{u}_j)$ that corresponds to this sibling node $\bar{u}_j$. (Recall,

property $\mathcal{P}(p(\bar{u}_j))$) is denoted simply as $\mathcal{P}(\bar{u}_j)$.) Let z be the least common ancestor of v and u in $T(p)$. The answer authentication information except from the signed time-stamped digest consists of three parts (see Figure 2.1):

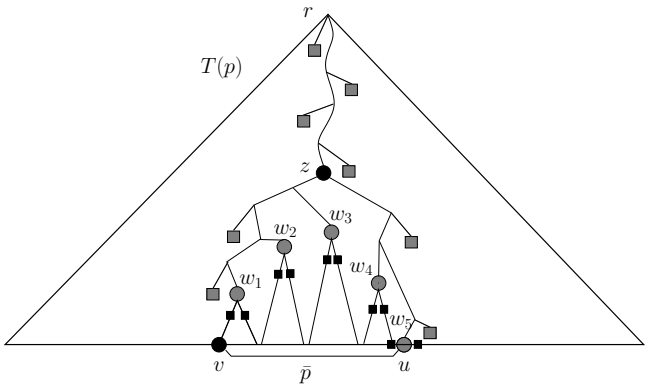

Figure 2.1: The answer authentication information for property$(\bar{p}(v,u))$ consists of: (1) the properties of allocation nodes $w_1,\ldots,w_5$ (grey circle nodes); (2) the labels of the children of the allocation nodes, if these children exist, or the node properties of their neighboring nodes in p, otherwise (black square nodes); (3) the labels and properties of sibling nodes of nodes in the leaf-to-root paths from v and u up to r that are not allocation nodes (grey square nodes).

1. for each allocation node $w \in \mathcal{A}(\bar{p})$, the property $\mathcal{P}(w)$, if w is not a leaf, or the property $\mathcal{N}(w)$ otherwise; these properties are given as a sequence $(\alpha_1,...,\alpha_m)$, such that the set of leaf nodes of any allocation nodes with properties α_i and α_{i+1}, $1 \leq i \leq m-1$, forms a subpath of $\bar{p}$;

2. for each allocation node $w \in \mathcal{A}(\bar{p})$, the labels of its children, if they exist, or the properties $\mathcal{N}(pred(w))$ and $\mathcal{N}(succ(w))$ otherwise;

3. and the labels and the corresponding path properties of the siblings of the nodes in the paths from the left most and right most allocation nodes up to the least common ancestor z, if these siblings are not allocation nodes themselves and the verification sequence of z.

Given that $T(p)$ is balanced, thus, for every $\bar{p}$, the set $\mathcal{A}(\bar{p})$ of allocation nodes of $\bar{p}$ has size $O(\log n)$, and given that a path property has constant size, the answer authentication information has size $O(\log n)$.

(3) To accept the answer, the user first recomputes $\mathcal{P}(\bar{p})$, by repeatedly applying the

concatenation function $\mathcal{F}$ on sequence $(\alpha_1, ..., \alpha_m)$. If $\mathcal{P}(\bar{p})$ is not verified, the answer is rejected. Otherwise, the verification process is completed by the computation and verification of the signed path hash accumulation. Observe that the user has all the necessary information needed for this procedure. The computation of the digest corresponds to tracing two leaf-to-root paths in $T(p)$ and at each node of the paths computing a hash label and possibly applying the concatenation function $\mathcal{F}$. The answer authentication information can be given in such a way so that this sequence of computations is well-defined for the user; e.g., $O(\log n)$ bits can be used to denote the left-right relation of siblings in $T(p)$. Clearly, since computing the path hash accumulation corresponds to tracing two paths of length $O(\log n)$, where at each node a constant amount of work is performed or, equivalently, to processing the answer authentication information which has size $O(\log n)$, the answer verification time is $O(\log n)$.

(1) − (3) (*Other query operations*) Considering the other three query operations of the path hash accumulator, we note the following. For a property(v) query, we proceed as above and the property($\bar{p}(v, u)$) case: observe that property(v) corresponds to property($\bar{p}(v, v)$). For a locate(p,σ,q) query, we locate the target node v by performing a top-down search in $T(p)$ starting from the root: at a node u with children w_1 and w_2, the path selection function σ is used to select either the path that corresponds to w_1 or the path that corresponds to w_2. Then, the answer is the located node v and the proof is the proof that corresponds to a property($\bar{p}(v, v)$) query. For a subpath(p,α,q) query, a similar top-down tree search is performed using the path advance function α to first compute the target subpath $\bar{p}(v, u)$; the proof is constructed by considering the corresponding allocation nodes. That is, the proof is the proof that corresponds to a property($\bar{p}(v, u)$) query. Thus, all these queries can be answered in $O(\log n)$ time, where the answer authentication information is of size $O(\log n)$ and the answer verification time is $O(\log n)$.

(4) Since a path property has constant size, the hash path accumulator occupies $O(n)$ space.

(5) Since the signed digest of the data set representing path p is simply a hash value and, after any update operation on the path hash accumulator, 1 (concatenate) or 2 (split) digests are signed by the source and sent to the responder, the update authentication information has constant size.

(6) Besides, the update operations can be implemented in $O(\log n)$ time using the following primitive update operations on trees (see, e.g., [134]): create_root (given two trees, create a root that merges them into one), delete_root (delete the root of a tree to create two new trees) and rotate (left or right rotation performed at a tree node). Observe that for

all these operations path properties can be computed and accordingly updated in constant time by applying the concatenation function $\mathcal{F}$. Thus, operation concatenate(p, q) involves updating the hash path accumulation for p and q, performing a create_root operation and then rebalancing the tree through rotations. Operation split involves performing rotations so that a delete_root operation creates the necessary target trees, then rebalancing the trees through rotations and finally updating the path hash accumulations of the trees. Note that for both operations, updating the path hash accumulations correspond to tracing at most two leaf-to-root paths and updating the hash labels stored in the visited nodes.

(*Security*) Considering the security provided by the hash path accumulator scheme, we note that we achieve the desired security results by reducing any attack from the responder against the user to a collision on the cryptographic hash function h or a successful attack against the signature scheme in use by the source and the user. With respect to the discussion on the security requirement in Section 2.2.3, we easily see that the hashing scheme of our path hash accumulator is well-defined with respect to path property queries, that is, it securely expresses these relations are these structural properties that are needed for verifying answer correctness: (1) path-related structural dependencies of elements of a path are expressed through the tree-based hashing scheme, and (2) the association of node and path properties with the corresponding subpaths is expressed through the inclusion of these properties in the hashing scheme. In particular, for every query, the answer consists of the set of properties stored at the set of allocation nodes $\mathcal{A}(\bar{p})$ of a subpath $\bar{p}$ of p. Any attack against the validity of the answer corresponds to forging set $\mathcal{A}(\bar{p})$, i.e., providing the properties stored at a different set $\mathcal{A}'(\bar{p})$ of allocation nodes, which, of course, may not correspond to any subpath of p. An attack against soundness corresponds to including at least one node $w \notin \mathcal{A}(\bar{p})$ in $\mathcal{A}'(\bar{p})$ and an attack against completeness corresponds to omitting at least one node $w \in \mathcal{A}(\bar{p})$ from $\mathcal{A}'(\bar{p})$. Since for every subpath $\bar{p}$ we have that $head(\bar{p}) \in \mathcal{P}(\bar{p})$ and $tail(\bar{p}) \in \mathcal{P}(\bar{p})$ and by the path hash accumulator hashing scheme that takes into account the left-right relation of siblings in $T(p)$ and path properties, we have that both soundness and completeness are achieved. Thus, if the set of allocation nodes that corresponds to the answer given to the user is not the correct one, for the user to accept the incorrect answer as correct, either an attack against the signature scheme or at least one collision on h must be computed by the responder. □

The path hash accumulator can be viewed as a generalization of the Merkle's hash tree. That is, it provides a tree-based authentication scheme capable in authenticating more sophisticated queries than membership queries and it also, as we will see in the next two

sections, a general framework for building more complex authenticated data structures. We end this section with a useful remarks concerning Lemma 2.3.1.

Remark 2.3.2. *Path property $\mathcal{P}(u)$ of subpath $p(u)$ is a sequence of constant size of path attributes. In Equation (2.2), $\mathcal{P}(u)$ participates in the hashing operation as the concatenation of a well-defined binary representation of the corresponding path attributes. That is, if $\mathcal{P}(u)$ consists of path attributes $a_1, \ldots, a_k$ that are related to path $p(u)$, then, choosing to implement the hash of a sequence of values as the hash of their concatenation (see Section 2.2.1), Equation (2.2) reads $L(u) = h(a_1 \| \ldots \| a_k \| L(z_1) \| L(z_2))$. Similarly, in Equation (2.1), any node property $\mathcal{N}(w)$ participates in the hashing operation as the concatenation of a well-defined binary representation of the corresponding node attributes of node w.*

Remark 2.3.3. *For some specific applications, the path hash accumulator can be defined using a slightly different hashing scheme. In particular, we can hash the path property stored at a node of the tree before this is included in the label computation through hashing. That is, in the definition of the hashing scheme, we can replace the hash operation in Equation (2.2) with the operation*

$$L(u) = h(h(\mathcal{P}(u)), L(z_1), L(z_2)). \tag{2.4}$$

This hashing scheme is more suitable in terms of performance in cases where the path property does not have constant size, but instead, it has size that is linear on the size of the path. We will encounter such a case for an specific type of queries in Section 2.4. We define the hash $h(\mathcal{P}(u))$ of the hash property of subpath $p(u)$ to be (as in one of the examples in Section 2.2.1) the hash of the concatenation of the hashes of a well-defined binary representation of the path attributes in $\mathcal{P}$. That is, if $a_1, \ldots, a_k$, where k is some constant, are the path attributes in $\mathcal{P}(u)$, then the hash of path property $\mathcal{P}(u)$ is $h(\mathcal{P}(u)) = h(a_1, \ldots, a_k) \triangleq h(h(a_1), \ldots, h(a_k))$. We note that the only changes that this modification of the hashing scheme brings to the results of Lemma 2.3.1 are: (1) *the answer authentication information is now slightly different, but still of logarithmic size, namely, Equation 2.3 now reads*

$$s_j = (L(\bar{u}_j), h(\mathcal{P}(\bar{u}_j))), \tag{2.5}$$

and (2) *according to the basic construction above, in the case where the path property has size that is proportional to the size of the path, the storage of the path hash accumulator becomes $O(n \log n)$. However, by introducing a special type of pointers into the data structure, a technique known with the name* threading*, we can actually reduce the storage needs of the data structure back to $O(n)$.*

2.4 Authenticated Graph Queries

In this section, we consider authenticated data structures for graph searching problems. We wish to authenticate search queries on graphs, like queries that ask for a path connecting two nodes in a graph (if any), or for some information associated with this path, e.g., the size of the connecting path, and queries that ask structural information about the graph, e.g., queries about the connectivity between two nodes. Such data structures have applications to the authentication of network management systems.

Given the path hash accumulator authentication scheme described in the previous section, we follow a bottom-up approach in presenting our new authenticated data structures. We first develop a generic authenticated data structure for a forest, that is, a collection of trees. The forest is dynamic, evolving through update operations that create, destroy, merge or separate trees. Trees store data items and querying information about these data items can be expressed by path property queries for paths in the trees of the forest and some path property that satisfies the concatenation criterion. Path hash accumulators are used as the primitive building blocks of this authenticated data structure. This new authentication structure for answering path properties queries on forests is then used to support more sophisticated queries on forests by appropriately defining the path property in use. Finally, we consider general graphs and use our forest related structures to authenticate searching queries on them.

2.4.1 Hierarchy of Paths

We start the construction of our authentication schemes by extending the use of path hash accumulators in *collections* of paths. In particular, we use the path hash accumulator authentication scheme over a dynamic collection Π of paths, that is maintained through the update operations on paths split and concatenate. At a high-level point of view, Π is organized by means of a rooted tree $\mathcal{T}$ of paths, meaning that each node of $\mathcal{T}$ corresponds to a path in Π. Neighboring paths in $\mathcal{T}$ are generally interconnected and share information. This is achieved by the definition of suitable node attributes and properties.

A tree of paths $\mathcal{T}$ is considered to be directed; the direction of an edge is from a child to a parent. Let μ be a node of $\mathcal{T}$, let $\mu_1, \ldots, \mu_k$ be its children in $\mathcal{T}$ and let p be the path that corresponds to node μ. A node attribute $N(v)$ of a node v of p is extended so that it depends not only on v but possibly also on some path properties of the paths $p_1, \ldots, p_k$ that correspond to nodes $\mu_1, \ldots, \mu_k$ of $\mathcal{T}$. We say that path p is the *parent path* of paths $p_1, \ldots, p_k$ and these paths are the *children paths* of p. This extension of the semantics of a

node attribute, i.e., allowing node attributes of a node of a path p to be related to the path properties of p's children paths, eventually allows the path property $\mathcal{P}(p)$ of path p that correspond to node μ to include information about paths in the subtree of $\mathcal{T}$ having as root node μ. As always, we consider path properties that satisfy the concatenation criterion.

In general, the idea above can be further extended by using a directed acyclic graph (instead a tree) as the high level graph for the organization of a path collection Π. In fact, using such a graph, we introduce a *hierarchy* over paths in Π, where, accordingly, path properties are extended to include information (expressed by path properties) about other paths subject to the hierarchy induced by the graph.

2.4.2 Path Properties in a Forest

We now develop an efficient and fully dynamic authenticated data structure that supports path property queries in a forest, where the forest is realized as a hierarchy of paths. The data structure has fast, update, query, and validation times.

Let $\mathcal{N}$ be any node property and $\mathcal{P}$ be any path property. We assume that $\mathcal{P}$ satisfies the concatenation criterion. Let F be a forest, a collection of trees. F is associated with a data set by storing at each tree node u some information as node attributes, or equivalently as node property $\mathcal{N}(u)$. Additionally, using the framework presented in Section 2.3, any path p in a tree of F is associated with some path property $\mathcal{P}(p)$. We study the implementation of the authenticated query operation $\mathsf{property}(u, v)$– return the path property $\mathcal{P}$ of the path from u to v in F, if such a path exists– while the following update operations over trees in F are performed:

- $\mathsf{destroyTree}(w)$– destroys the tree with root w;
- $\mathsf{newTree}()$– creates a new tree in F that consists of a new, single, node;
- $\mathsf{link}(u, v)$– merges two trees into one by adding an edge between the root u of some tree to a leaf v of another tree;
- $\mathsf{cut}(u)$– separates a tree to two new trees by removing the edge between non-root node u and its parent.

Note that any tree can be assembled or disassembled using these operations.

Our data structure is based on dynamic trees [134] introduced by Sleator and Tarjan. Conceptually, a dynamic tree T is a rooted tree whose edges are classified (according to some criteria) as being either *solid* or *dashed*, with the property that any internal node of

T has at most one child connected by a solid edge. This edge classification partitions the tree into *solid paths*, i.e., consecutive nodes connected with each other through solid edges, whereas these solid paths are connected with each other by dashed edges (see Figure 2.2(a)). Using the framework of Section 2.3, we view every solid path of a dynamic tree as a path, i.e., a sequence of nodes, directed towards the root of T, that is, the predecessor of any node that is not the tail of the path is its parent node.

Moreover, by definition, every non-leaf node v of a dynamic tree T has at most one child u_0 such that a solid edge connects them. Assume that v has more children and consider all these children, say nodes $u_1, \ldots, u_k$ in T (connected with v through dashed edges). Using again the framework of Section 2.3, we define the *dashed path* $d(v)$ of node v to be a path, i.e., a sequence of nodes of length k, such that there is a one-to-one correspondence between edges (u_i, v) in T and path nodes of $d(v)$. The ordering of the nodes in $d(v)$ is thus in accordance with the ordering of nodes $u_1, \ldots, u_k$, which in turn can be arbitrary.

Having at hand the solid and dash paths defined in a dynamic tree, we now consider the trees of the forest F to be dynamic trees, which allows us to perform a transformation of trees into solid and dash paths. In particular, let $T_1, \ldots, T_m$ be the trees in F. We view all these trees as dynamic trees. Let $\Pi(T_i)$ be the collection of all solid and dashed paths defined for tree T_i of F as explained above. Using the concept of hierarchies of paths discussed in Section 2.4.1, we can associate $\Pi(T_i)$ with a directed tree $\mathcal{T}_i$ of paths. This is performed as follows:

- each path p (solid or dashed) in $\Pi(T_i)$ corresponds to a vertex μ_p of $\mathcal{T}_i$;
- if p is solid, for each node v of p that has only one child u in T_i such that u is node of path p' in $\Pi(T_i)$ and $p \neq p'$ (and v is connected with u through a dashed edge), the directed edge $(\mu_{p'},\mu_p)$ is an edge of $\mathcal{T}_i$;
- if $p' = d(v)$ is dashed with length k, that is, p' corresponds to the dashed edges of a node v in T_i, let p be the solid path that v belongs to, let $u_1, \ldots, u_k$ be the corresponding children of v in T_i, and let $p_1, \ldots, p_k$ be the solid paths containing these children; then, directed edges $(\mu_{p'},\mu_p)$ and $(\mu_{p_i},\mu_{p'})$, $1 \leq i \leq k$, are edges of $\mathcal{T}_i$.

Finally, given the directed trees of paths $\mathcal{T}_i$, $i = 1, ..., m$, we add a new *root vertex* ω which is the parent of all the roots of trees $\mathcal{T}_i$, thus, obtaining a new tree $\mathcal{F}$[4]. All the newly added edges are directed towards ω. We consider one last *root path* $\pi(\omega)$ that corresponds

[4]Root vertex ω is a fictitious node, used only as an intermediate tool to define a special root path on top of the trees $\mathcal{T}_i$.

to the root vertex ω. The nodes of this path correspond to trees T_is of F, where any node ordering in $\pi(\omega)$ can be used. Our final graph is a tree of solid and dash paths rooted at a special root path.

Consider the collection $\Pi(F)$ of paths (solid, dash, root) associated with the nodes of tree $\mathcal{F}$. The children of the root path $\pi(\omega)$ are solid paths. The children of a solid path are either solid of dashed paths. The children of a dashed path are solid paths. Figure 2.2(b) shows such a tree $\mathcal{F}$.

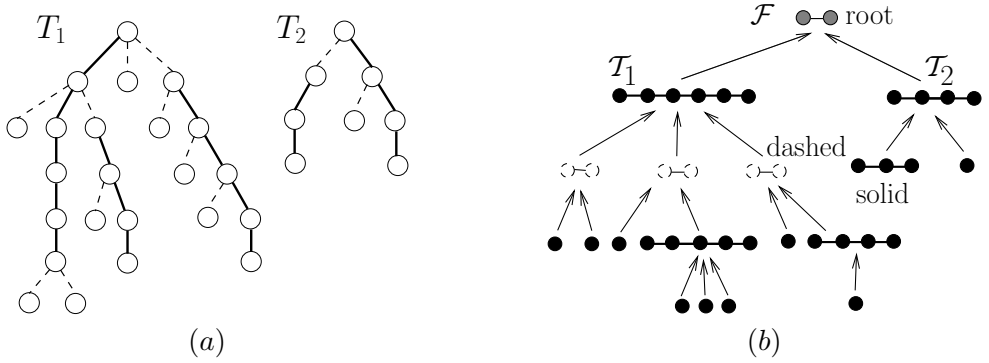

Figure 2.2: (a) The partition of trees into solid paths. (b) Trees of paths and final tree $\mathcal{F}$.

Using this tree of paths, we implement our data structure as follows. Each path (root, solid or dashed) is implemented through the path hash accumulator authentication scheme (Section 2.3), where the individual data structure that implements each path hash accumulator is chosen to be a biased binary tree [6].

A path property $\mathcal{P}$ of our interest, i.e., a collection of path attributes, that satisfies the concatenation criterion is defined. By the implicit path interconnection, through the idea of setting a path property of a path to be a node attribute of another, neighboring, path, $\mathcal{P}(p)$ includes information about the children paths of p and, in general, about all of its descendant paths. We include path attributes in node properties, as follows. If v is a node of path p and $L(p)$ denotes the path hash accumulation of path p, then:

1. if p is root path or dashed path, then $L(p')$ and $tail(p')$ are included in $\mathcal{N}(v)$, where p' is the child solid path of p corresponding to node v;

2. if p is solid path, then $L(p')$ and $tail(p')$ are included in $\mathcal{N}(v)$, if v corresponds to a solid child path p', or $L(p')$ is included in $\mathcal{N}(v)$, if v corresponds to a dashed child path p'.

The above scheme of inclusions of path attributes and path properties of a path as a

node attribute in the node property of a node of the parent path corresponds to connecting the individual hashing schemes of the path hash accumulators implementing paths in $\mathcal{F}$ and composing them into one hashing scheme G for the entire data structure. This hashing scheme G yields a digest for data stored in the forest F, namely, the path hash accumulation of the root path $\pi(\omega)$ of $\mathcal{F}$. Next, we present our first theorem, which fills in the details of the entire authenticated data structure that supports update and query operations on forest F, analyze its performance and prove its efficiency.

Theorem 2.4.1. *Let F be a forest of trees with n nodes. There exists a fully dynamic authenticated data structure that supports query operations* property *on paths in dynamic forest F that evolves through update operations* destroyTree, newTree, link *and* cut *having the following performance:*

1. *query operation* property *takes $O(\log n)$ time;*
2. *for query operation* property *the answer authentication information has size $O(\log n)$;*
3. *for query operation* property *the answer verification time is $O(\log n)$;*
4. *the total space used is $O(n)$;*
5. *for every update operation the update authentication information has size $O(1)$;*
6. *update operations* destroyTree *and* newTree *take each $O(1)$ time; update operations* link *and* cut *take each $O(\log n)$ time.*

Proof. (*Data Structure*) We first complete the description of the data structure. As we already have seen, the entire forest F is represented as a collection of paths $\Pi(F)$, which is organized using the hierarchy induced by the tree $\mathcal{F}$ of paths in $\Pi(F)$. Recall that the solid paths in $\Pi(F)$ are defined by considering each tree T_i of F to be a dynamic tree and by using a partition of edges into solid and dashed. In any tree T_i, let $\mathsf{size}(v)$ denote the number of nodes in the subtree defined by v and let u the parent node of v. Edge $e = (u, v)$ is called *heavy* if $\mathsf{size}(u) > \mathsf{size}(v)/2$. The edge labeling of dynamic tree T_i of m_i nodes with root w, such that an edge is labeled solid only if it is heavy, has the following important property [134]: for any node u of T_i there are at most $\log m_i$ dashed edges on the path from u to w. We use this edge labeling to partition each tree T_i into solid paths.

Consider all the paths that correspond to the final tree $\mathcal{F}$ (after the dashed paths and the root path have been added). Each path p of $\mathcal{F}$ is represented using the path hash

accumulator authentication scheme[5] of Section 2.3, with only one exception. Path p is represented by a binary tree $T(p)$, but $T(p)$ is implemented as a biased binary tree $T(p)$ (see [6]), thus, it is not necessarily balanced[6].

In a biased binary tree, each leaf node is associated with a weight, each non-leaf node is associated with the sum of the weights of its children, (consequently) tree's root r carries the sum W of the weights of the leaves and any node v lies at depth $O(\log \frac{W}{w(v)})$. For us, node weights are defined using function $\mathsf{size}()$, and we consider weight $w(v)$ of node v to be an additional node or path attribute (depending on whether v is a leaf node in $T(p)$ or not). If p is a path having no child path (μ_p is a leaf in $\mathcal{F}$), then $w(v) = \mathsf{size}(v)$. Otherwise (μ_p is not a leaf in $\mathcal{F}$), $w(v) = w(u_1) + w(u_2)$, if v is internal node of $T(p)$ with children u_1, u_2. Otherwise, v is a node of path p. If $v = head(p)$ and p is solid, then $w(v) = w(u_1) + w(u_2)$, where u_1,u_2 are the children of v in T (through dashed edges). If $v \neq head(p)$ and p is solid, then $w(v) = w(u) + 1$, where u is the unique dashed child of v and $w(u) = 0$, if no such child exists. If p is dashed, $w(v) = w(u) + 1$, where u is the node connected with v with the corresponding dashed edge. If p is root path, again $w(v) = w(u) + 1$, where u is the root of the corresponding tree root.

To complete the description of the authenticated data structure, we note that the hashing scheme of the entire data structure is defined through the individual path hash accumulator hashing schemes of the paths in $\mathcal{F}$. All these hashing schemes compose a hashing scheme for the entire forest $\mathcal{F}$ and the date structure as a whole, where the path hash accumulation of a path p is included in the computation of the path hash accumulation of p's parent path in $\mathcal{F}$.

(*Efficiency*) Consider any tree T_i in F, any two nodes u and v of T_i and the path p_{uv} in T_i that connects u and v. Our data structure represents T_i implicitly through the path collection $\Pi(T_i)$, where each path in $\Pi(T_i)$ is implemented as a (biased) binary tree. For answering queries regarding properties of path p_{uv}, we first consider a *multipath* π_{uv}, that is, a path of paths in $\mathcal{T}_i$, that connects the paths in $\Pi(T_i)$ where u and v belong, and using this multipath, we then consider a (different of p_{uv}) path P_{uv}: the path that virtually connects u and v in $\mathcal{T}_i$ *through* the binary trees that implement the paths of $\Pi(T_i)$. Path P_{uv} is a connecting path of nodes u and v in the hashing scheme of the entire data structure, when no edge directions are taken into consideration. These three types of paths p_{uv}, π_{uv} and

[5]Recall that using this term we refer in an indistinguishable way to both the data structure implementing the tree and the corresponding hashing scheme.

[6]Note that this fact does not affect the corresponding hashing scheme; the path hash accumulation $L(T(p))$ of p is still well-defined.

P_{uv} are described in Figure 2.3.

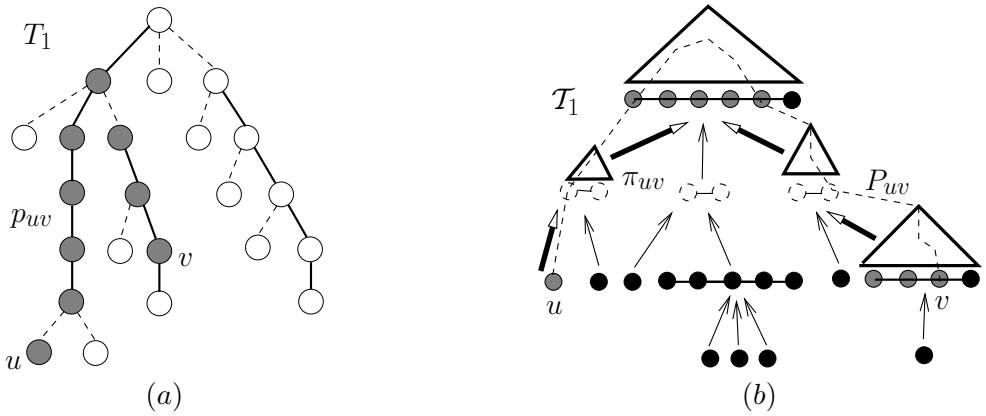

Figure 2.3: (*a*) Path p_{uv} in tree T_1 (grey nodes), connecting nodes u and v. (*b*) Multipath (path of paths) π_{uv} in $\mathcal{T}_i$ (indicated by dark arrows), connecting the paths containing u and v and path P_{uv} (indicated by dashed line) in the data structure, connecting nodes u and v *through* the binary trees implementing paths in $\Pi(\mathcal{T}_i)$. Triangles denote biased binary trees, not necessary balanced. Path P_{uv} is also a path in the hashing scheme G of the entire data structure. Observe that P_{uv} defines (visits) allocation nodes in binary trees implementing solid paths in $\mathcal{T}_i$, which allocation nodes correspond to subpaths of p_{uv} and store path properties that constitute path property $\mathcal{P}(p_{uv})$. P_{uv} has logarithmic on the size of T_1 length.

Observe that path P_{uv} passes through nodes of the binary trees implementing solid paths in $\mathcal{T}_i$ that have as children, tree nodes defining subtrees whose leaves are subpaths of p_{uv}, that is, nodes that are allocation nodes of subpaths of path p_{uv}. This is exactly what is needed: in authenticating a path property of p_{uv}, we will use the path properties stored at nodes (of the biased binary trees) related to path P_{uv} that, given the fact that the path property in study satisfies the concatenation criterion, completely describe the path property of p_{uv}. Accordingly, the above considerations are also valid for paths in the data structure that connect nodes u and v of different trees in F: in this case, a multipath in $\mathcal{F}$ that passes through the root path $\pi(\omega)$ connecting the paths in $\Pi(F)$ of nodes u and v exists and the corresponding path P_{uv}, connecting u and v in the hashing scheme of the data structure, passes through the binary tree implementing the root path $\pi(\omega)$.

Using of the previously described biasing in our data structure, it can be shown that, when considered *through* the individual biased binary trees that implement paths in $\mathcal{F}$, any path P_{uv} in the data structure in tree T_i of size m_i has length $\log m_i$ and any leaf-to-root path in the data structure representing $\mathcal{F}$ has length $O(\log n)$. The proof is based on the analysis in [134] . Consequently, for any nodes u, v in F, path P_{uv} in our data structure

has length $O(\log n)$.

(1) – (3) Consider query property(u, v). Although this query is defined for nodes in forest F, to cover the most general case, we do not require that query nodes u and v necessarily exist in F. So, we first determine whether nodes u and v are in forest F using any authenticated data structure that supports containment queries (e.g., [52]) in $O(\log n)$ time with authenticated responses of size $O(\log n)$. If one of the two nodes are not in F, a negative answer is given, along with a proof that verifies the negative containment. If u and v are in F, the path property query is performed by accessing three multipaths in $\mathcal{F}$ (see Figure 2.4).

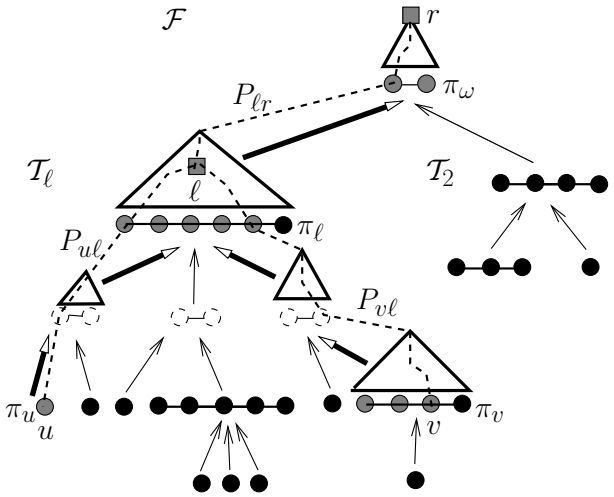

Figure 2.4: The answer and proof for property(u, v) query are computed by visiting paths $\pi_{u\ell}$, $\pi_{v\ell}$ and $\pi_{\ell r}$ in the data structure and by accessing information stored at nodes related to these paths.

In particular, assume first that u and v belong to the same tree T_i of size m_i in F, i.e., there exists a connecting path p_{uv} in F. Let π_u and π_v be the paths in $\mathcal{F}$ that contain u and v and let π_ℓ be the least common ancestor π_u and π_v in $\mathcal{F}$ (π_l may overlap with π_u and/or π_v). Let $\pi_{u\ell}$, $\pi_{v\ell}$ be the multipaths from π_u, π_v to π_ℓ and $\pi_{\ell r}$ be the multipath from π_l to the root path $\pi(\omega)$ of $\mathcal{F}$. Multipaths $\pi_{u\ell}$, $\pi_{v\ell}$ and $\pi_{\ell r}$, when considered *through* the binary trees used to implement paths in $\mathcal{F}$, define paths $P_{u\ell}$, $P_{v\ell}$ and $P_{\ell r}$, respectively, in the data structure. Specifically, paths $P_{u\ell}$ and $P_{v\ell}$ are connected at node ℓ of the binary tree implementing path π_ℓ (node ℓ is the least common ancestor of u and v in the data structure). The answer $A(u, v)$ given to the user is computed by following paths $\pi_{u\ell}$ and $\pi_{v\ell}$, finding, at each traversed path hash accumulators that implement a solid path in $\mathcal{F}$, the allocation nodes whose subtrees correspond to subpaths of p_{uv} and, finally, reporting the path properties stored

at these allocation nodes. Similarly, the proof given to the user is collected by providing path property proofs (i.e., collection of appropriate hash values and path properties as in Section 2.3 and in the proof of Lemma 2.3.1) $proof(\pi_{u\ell})$, $proof(\pi_{v\ell})$, each corresponding to a traversed multipath to π_{ell} and consisting of subproofs: a subproof for each path hash accumulator that is visited. To compute the proof, following path $P_{\ell r}$ up to the root r of the binary tree implementing path $\pi(\omega)$, we compute the *multipath verification sequence* $\mathcal{V}(\ell)$, a collection of hash values and path properties (similar to the verification sequence in the proof of Lemma 2.3.1) that allows the user to recalculate the signed hash value (hash path accumulation of $\pi(\omega)$) given $A(u, v)$, $proof(\pi_{u\ell})$ and $proof(\pi_{v\ell})$. By the biased scheme used over $\mathcal{F}$, the set of allocation nodes of path p_{uv} has size $O(\log m_i)$ and also paths $\pi_{u\ell}$, $\pi_{v\ell}$ and $\pi_{\ell r}$ have each $O(\log n)$ size. Since path properties have constant size and for each node of these paths visited in the data structure a constant amount of information is included in the proof and a constant amount of work is performed , the answer $A(u, v)$ has $O(\log n)$ size, the proof ($\mathcal{V}(\ell)$, $proof(\pi(u))$, $proof(\pi(v))$) has also size $O(\log n)$ and path properties queries are answered in $O(\log n)$ time. Accordingly, clearly the verification time is also $O(\log n)$.

Similarly, in the special case[7] where no path connecting u and v exists in F (u and v belong in different trees), a negative answer is given to the user, indicating that no path connecting the query nodes exists in F) and the previous approach is used to provide proof of this fact: proofs corresponding to paths $P_{u\ell}$, $P_{v\ell}$ and $P_{\ell r}$ verify the nonexistence of path p_{uv}, since the least common ancestor ℓ of u and v in the data structure is node of the binary tree implementing root path $\pi(\omega)$.

(4) Since a path property has constant size, the hash path accumulator occupies $O(n)$ space.

(5) Since the signed digest of the entire data structure is simply a hash value, the path hash accumulation of the root path in $\mathcal{F}$, the update authentication information has constant size.

(6) All update operations correspond to accessing and modifying multipaths through the primitive path operations split and concatenate. In particular, operations link and cut can be implemented in $O(\log n)$ time by modifying only $O(\log n)$ path hash accumulators and by examining, modifying and restructuring only $O(\log n)$ nodes in total. Restructuring

[7]This case is included in the proof only for completeness, since, by definition, a path property query returns the path property of existing in F paths. Our used approach is similar with the one used in Section 2.4.3 for the authentication of higher-level path and connectivity queries: the nonexistence of a path in F is authenticated by the existence of a path in $\mathcal{F}$ passing through the root path $\pi(\omega)$.

means connecting a node to new children. Our scheme works by, every time a node v is restructured, recalculating $L(v)$, which can be done in $O(1)$ time, since the values of the children, parent or neighbors of v are known. Consequently, our update operations can be performed in $O(\log n)$ time.

(*Security*) Hashing scheme G is based on the path hash accumulator. By allowing neighboring (in $\mathcal{F}$) paths to share information (properties) we achieve the desired security results based on the security of path hash accumulator authentication scheme (proof of Lemma 2.3.1): any attack to our data structure can be reduced to an attack on the security of the path hash accumulator, thus, in turn to a collision on the cryptographic hash function h. □

2.4.3 Path, Connectivity and Type Queries on Forests

Theorem 2.4.1 supports the basis for an authenticated data structure that efficiently answers and authenticate the following queries on a dynamic forest F:

- path(u, v): reports the path, if any, between nodes u and v in F;
- pathLength(u, v): reports the length of the path, if any, between nodes u and v in F;
- areConnected(u, v): reports whether there is a path between nodes u and v in F (i.e., whether u and v are nodes of the same tree);
- type(u, v): reports whether there is a node of a given type[8] in a path, if any, between nodes u and v in F.

Our results are obtained by appropriately defining a path property that expresses each one of the above queries. In other words, each new query is answered as a path property query, as in Section 2.4.2, for a specially defined path property, which, of course, satisfies the concatenation criterion. Each specific query corresponds to a specially chosen individual path attribute that is included in the path property in use. The next theorem presents the details of these results.

Theorem 2.4.2. *Let F be a forest of trees with n nodes. There exists a fully dynamic authenticated data structure that supports query operations* path, pathLength, areConnected *and* type *on paths in dynamic forest F that evolves through update operations* destroyTree,

[8] We assume that the type of a node is a well-defined notion and that the type of a node can be checked in constant time.

newTree, link and cut having the following performance, where k is the length of the path returned by operation path:

1. *query operations pathLength, type and areConnected*
   - *each take $O(\log n)$ time;*
   - *each have answer authentication information of size $O(\log n)$;*
   - *each have $O(\log n)$ answer verification time;*
2. *query operation path*
   - *takes $O(\log n + k)$ time;*
   - *has responder-to-user communication cost[9] of size $O(\log n + k)$, where the answer has size $O(k)$ and the answer authentication information has size $O(\log n)$;*
   - *has $O(\log n + k)$ answer verification time;*
3. *the total space used is $O(n)$;*
4. *for every update operation the update authentication information is $O(1)$;*
5. *update operations destroyTree and newTree take each $O(1)$ time; update operations link and cut take each $O(\log n)$ time.*

Proof. (*Data Structure*) It is essentially the same data structure as the one of Theorem 2.4.1, except one main difference with respect to the support of query path and the exact hashing scheme of the data structure that is explained in detail in what follows[10]. The path property $\mathcal{P}$ in use – except, by definition, from (the *id*s of) the head and the tail nodes of the corresponding path – includes four specially chosen path attributes a_1, a_2, a_3 and a_4 one for each of the supporting query operations, respectively, path, pathLength, areConnected and type. In this way, path property $\mathcal{P}$ contains information for all supported queries and, more importantly, it has size that is *linear* on the size of the corresponding path.

Because of this, we use the implementation of the hashing operation of Equation 2.4 in defining the hashing scheme of our data structure (see Remark 2.3.3). In our implementation, the hashing operation over a path property is accordingly designed (see Section 2.2.1

[9] Since the answer size in this case is not constant, we analyze the responder-to-user communication cost and accordingly distinguish the costs of the answer and the answer authentication information.

[10] However, an alternative implementation of our data structure could be as follows. The data structure simply consists of the aggregation of four different hashing schemes, one for each supported query operation. That is, each query operation corresponds to a property query for a specially defined path property.

and Remark 2.3.3) so that more efficiency is achieved. In particular, the path property $\mathcal{P}$ is viewed as a sequence of path attributes a_1, a_2, a_3, a_4 and, accordingly, its hash value is computed as

$$h(\mathcal{P}) = h(h(a_1), h(a_2), h(a_3), h(a_4)). \tag{2.6}$$

That is, the hash of a path property is the hash of the concatenation of the hash values of the attributes that it consists of. The time complexity in computing this hash value is linear on the size of the path.

Overall, this choice for implementing the hashing operation used in the definition of the hashing scheme affects has two important consequences. First, the answer authentication information is still logarithmic in the size of the forest (regardless the fact that the path property has size proportional to the path). Second, query operations can be answered separately, where path attributes are treated not as a whole but as individual pieces of information, and not a single path property query of Theorem 2.4.1. That is, attribute a_1 of a path can be authenticated without the authenticating or revealing any of the other attributes. In essence, using this hashing scheme, a path property query can be answered as in Theorem 2.4.1 but *only with respect to* a specific path attribute.

Moreover, regarding the concatenation function $\mathcal{F}$ of property $\mathcal{P}$, it is simply defined as a per attribute application of function $\mathcal{F}$. That is, if $p = p' \| p''$, $\mathcal{P}(p') = (a'_1, a'_2, a'_3, a'_4)$ and $\mathcal{P}(p'') = (a''_1, a''_2, a''_3, a''_4)$, then

$$\mathcal{P}(p) = \mathcal{F}((a'_1, a'_2, a'_3, a'_4), (a''_1, a''_2, a''_3, a''_4)) = (\mathcal{F}(a'_1, a''_1), \mathcal{F}(a'_2, a''_2), \mathcal{F}(a'_3, a''_3), \mathcal{F}(a'_4, a''_4)).$$

Additionally, since the path property is not of constant size, the concatenation function $\mathcal{F}$ operated in time proportional to the path property.

In what follows, we define the path attributes, discuss their concatenation function $\mathcal{F}$ and any relevant to our authenticated data structure complexity issues.

(1) Query operation areConnected corresponds to the existence of a node of the root path $\pi(\omega)$ in the path P_{uv} connecting nodes u and v in the data structure representing forest F (or tree $\mathcal{F}$). Note that path P_{uv} always exists. That is, the answer to the query is negative if such a node exists in P_{uv} and positive if no node of the root path $\pi(\omega)$ exists in P_{uv}. This property is expressed by assigning a unique *id* value to every path in the tree of paths $\mathcal{F}$. Thus, attribute a_3 of a path p in the tree of paths $\mathcal{F}$ and accordingly in our data structure is defined to take on one of the following two values: root-path-true, if p is a subpath of $\pi(\omega)$, or root-path-false. Concatenation function $\mathcal{F}$ operates in constant time as the OR boolean function on these two values of a_3.

A similar idea is applied for query operation type. A node attribute corresponding to path attribute a_1 (or path attribute a_4 of path of size 1) takes on two values: either type-true or type-false, depending on whether on not the corresponding node is of the type of interest (that the query operation asks about). Again, the concatenation function operates as the boolean function OR.

For query operation pathLength, we define the path attribute a_2 of a path to be simply the size of the path and the concatenation function is the addition function.

The complexity for all these three query operations is similar to the complexity of the path property query operations of Theorem 2.4.1. Although, path attribute a_1 has size that is proportional to the size of the corresponding path, because of the hashing scheme, which hashes individually the path properties in producing the hash of a path property (Equation 2.6), no path attribute a_1 is included in the answer or in the answer authentication information (see Remark 2.3.3, Equation 2.5).

(2) Query operation path is answered by first performing a query areConnected. If there is a path between nodes u and v, it can be found by answering a path property query with respect to path attribute a_1, where attribute a_1 of path p includes all the (ids of the) nodes of path p, that is, the path itself! To this end, we need a slightly different definition for the path attribute, namely, a path attribute can be of any size (not necessarily constant). Since $|a_1| = O(|p|)$, the introduced complexity is $O(\log n + k)$, where k is the length of the path from u to v. In particular, both the query time and the answer verification time are $O(\log n + k)$, since the answer itself is of size $O(k)$ and both computations need to spend time proportional to the answer (to compute and process the answer respectively). The logarithmic term is due to the complexity carried from Theorem 2.4.1. On the other hand, the answer authentication information is still logarithmic, since our hashing scheme applies an extra hashing of the path property, before the hashing of the sibling hash labels (see Equation 2.4).

(3) Although, according to the construction in Section 2.3 the path hash accumulator for a path property of linear size has $O(n \log n)$ storage, using *threads* in our data structure, we can reduce the storage needs to $O(n)$. The idea is to store path attribute a_1 of path p, the one corresponding to the subtree defined by internal node u, at the leaves of this subtree (that is, at the path p itself), rather than storing it at node u. Then, we add a special pointer from node u to $head(p)$ and a special pointer from $tail(p)$ up to node u and pointers for every node in solid path towards its successor node in the path (if any). These pointers, called threads, can be used to traverse path p in $O(|p|)$ time and compute (retrieve) the path attribute a_1. Thus, storage can still be kept linear, even with the presence of a path

property of non linear size.

(4), (5) & (*Security*) They follow directly from Theorem 2.4.1. □

In the rest of the section, we show how this result can be extended to give us authenticated schemes for more advanced graph queries. All these results have applications to the authentication of network management systems.

2.4.4 Path and Connectivity Queries on Graphs

We now move our attention to graphs rather than forests. Suppose we want to authenticate path and connectivity queries on a general graph G. That is, as before, we want to authenticate the answers to the following queries on G:

- path(u, v): report the path, if any, between nodes u and v in G;
- areConnected(u, v): report whether there is a path between nodes u and v in G (i.e., whether u and v are nodes of the connected component in G).

We can immediately apply Theorem 2.4.2 to design an authenticated data structure for path and connectivity queries in a graph G that evolves through vertex and edge insertions. In particular, graph G is maintained through update operations:

- makeVertex(v): create a new vertex v in G;
- insertEdge(u, v, e): add edge e between vertices u and v in G.

The new data structure has similar performance bounds with the one in Theorem 2.4.2. The main idea is our data structure to maintain a spanning forest F of the graph G. We note that for embedded planar graphs our data structure can actually be extended to also support deletions of vertices and edges, through new update operations:

- destroyEdge(e): destroy edge e in G; and
- destroyVertex(u): destroy isolated vertex u in G.

The idea is to use techniques similar to the data structure described in [39].

Theorem 2.4.3. *Let G be a general graph with n nodes. There exists a semi-dynamic authenticated data structure that supports query operations* path, *and* areConnected *on pairs of nodes in graph G that evolves through update operations* makeVertex *and* insertEdge *having the following performance, where k is the length of the path returned by operation* path*:*

1. *query operation* areConnected *takes* $O(\log n)$ *time and query operation* path *takes* $O(\log n + k)$ *time;*

2. *for query operation* areConnected *the answer authentication information is of* $O(\log n)$ *size, and for query operation* path *the responder-to-user communication cost is of* $O(\log n + k)$ *size, where the answer has size* $O(k)$ *and the answer authentication information has size* $O(\log n)$*;*

3. *for query operation* areConnected *the answer verification time is* $O(\log n)$*; for query operation* path *the answer verification time is* $O(\log n + k)$*;*

4. *the total space used is* $O(n)$*;*

5. *for every update operation the update authentication information is* $O(1)$*;*

6. *update operations* makeVertex *and* insertEdge *take* $O(1)$ *and* $O(\log n)$ *time, respectively;*

7. *if* G *is an embedded planar graph, additional update operations* destroyEdge *and* destroyVertex *are supported in* $O(\log n)$ *and* $O(1)$ *time, respectively.*

Proof. (*Data Structure*) The idea is to use the data structure of Theorem 2.4.2 to maintain a *spanning forest* of graph G. That is, we maintain a forest F that spans through the entire graph, meaning that all nodes in G are nodes of F as well and that each connected component of G corresponds to a tree of F. Data structure of Theorem 2.4.2 allows us to authenticate answers to path and connectivity queries. Note the correctness of the data structure: if two nodes are connected in G, they are connected in F as well, for they belong to the same connected component of G, thus to the same tree in F, and in this case, the path connecting them in this tree is obviously a path connecting them in G as well.

Update operations are handled as follows. For general graphs, where only vertex and edge insertions are supported, each new vertex corresponds to a new connected component of G, thus, to simply a new tree in F and a newTree update operation, whereas each new edge either corresponds to no action, when it connects nodes of the same tree (connected component in G) in F, or it corresponds to a link operation, when it connects nodes of different trees (connected components in G) in F. Testing whether or not an edge connects nodes of the same tree in F can be performed by assigning unique *id*s to all trees in F and checking whether or not the two nodes belong in trees with the same *id*. The last operation can be done by simply storing at each node of F the corresponding tree *id* (or even by accordingly defining a path attribute).

Update operations for embedded planar graphs, which include not only edge and vertex insertions but also deletions, are a bit trickier to handle. First, any insertion of an edge connecting nodes of the same tree needs to be stored. After any deletion of an edge or a vertex, the set of stored edges that are not edges in F and, thus, not explicitly stored in the data structure representing F, is processed to decide whether or not this deletion results in a connected component destruction in G. Using the data structure described in [39], one can decide on whether a new connected component is created or, instead, the connected component stays the same but with a different spanning tree this time. Both this decision and the update of the component can be done in logarithmic on the size of the graph time, using the fact that edges admit efficient representation because of the planarity property of the graph G. The use the data structure in [39] is an orthogonal issue in our data structure, meaning that it is not connected with the operation of the authenticated data structure, but rather, it supports the maintenance of the forest F. Once forest F is updated – always through the update operations that the data structure supports – the hashing scheme is accordingly updated and the new digest is computed.

(1) – (7) & (*Security*) They follow immediately from Theorem 2.4.2. □

In the next two subsections we use the results of Theorem 2.4.1 for connectivity queries for general graphs that evolve through edge and vertex insertions. We use known techniques (data structures) that support these type of queries for regular (non-authenticated) data structures and apply our authentication framework of Section 2.3, which is based on paths and their properties, on these data structures by appropriately authenticating path properties that are related to the connectivity queries that we study. In other words, here, we have the first applications of our authentication framework of the path hash accumulator to the authentication of queries that are not directly related to paths.

2.4.5 Biconnectivity Queries on Graphs

As before, let G be a general graph that is maintained through update operations makeVertex and insertEdge. We are interested in authenticating the query operation

- areBiconnected(u, v): determines whether u and v are in the same biconnected component of G,

which we call a *biconnectivity* query. Theorem 2.4.1 can be used to support an authenticated data structure that answers biconnectivity queries.

Theorem 2.4.4. *Let G be a general graph with n nodes. There exists a semi-dynamic authenticated data structure that supports query operation* areBiconnected *on pairs of nodes in graph G that evolves through update operations* makeVertex *and* insertEdge *having the following performance:*

1. *query operation* areBiconnected *takes $O(\log n)$ time; for this query operation, the answer authentication information has size $O(\log n)$ and the answer verification time is $O(\log n)$;*
2. *the total space used is $O(n)$;*
3. *for every update operation the update authentication information is $O(1)$;*
4. *update operation* makeVertex *takes $O(1)$ time, and update operation* insertEdge *takes $O(\log n)$ amortized time.*

Proof. (*Data Structure*) We extend the data structure of [146]. We maintain the *block-cut-vertex forest* $\mathcal{B}$ of G. Each tree T in $\mathcal{B}$ corresponds to a connected component of G. There are two types of nodes in T: *block nodes* that correspond to blocks (biconnected components) of G and *vertex nodes* that correspond to vertices of G. Each edge of T connects a vertex node to a block node. The block node associated with a block B is adjacent to the vertex nodes associated with the vertices of B. We have that two vertices u and v of G are in the same biconnected component if and only if there is a path between the vertex nodes of $\mathcal{B}$ associated with u and v and this path has length 2. Thus, operation areBiconnected in G is reduced to performing operation pathLength in $\mathcal{B}$ and certifying that the returned path length equals 2.

(1)−(4) & (*Security*) They follow immediately from Theorem 2.4.2. The time complexity for edge insertions is amortized, because these are the guarantees for the data structure in [146]. □

2.4.6 Triconnectivity Queries on Graphs

Finally, we show how to authenticate the following query operation:

- areTriconnected(u, v): determine whether u and v are in the same triconnected component G,

which we call a *triconnectivity* query, in a general graph G maintained through edge and vertex insertions as before. Again, we use the results of Theorem 2.4.1 to construct an authenticated data structure that answers triconnectivity queries.

Theorem 2.4.5. *Let G be a general graph with n nodes. There exists a semi-dynamic authenticated data structure that supports query operation* areTriconnected *on pairs of nodes in graph G that evolves through update operations* makeVertex *and* insertEdge *having the following performance:*

1. *query operation* areTriconnected *takes $O(\log n)$ time; for this query operation, the answer authentication information has size $O(\log n)$ and the answer verification time is $O(\log n)$;*
2. *the total space used is $O(n)$;*
3. *for every update operation the update authentication information is $O(1)$;*
4. *update operation* makeVertex *takes $O(\log n)$ time; update operation* insertEdge *takes $O(\log n)$ amortized time.*

Proof. (*Data Structure*) We extend the data structure of [37], where a biconnected graph (or component) G is associated with an *SPQR tree* T that represents a recursive decomposition of G by means of separation pairs of vertices. Each S-, P-, and R-node of T is associated with a triconnected component C of G and stores a separation pair (s, t), where vertices s and t are called the *poles* of C. A Q-node of T is associated with an edge of G. Each vertex v of G is allocated at several nodes of T and has a unique *proper allocation node* in T.

Our authenticated data structure augments tree T with V-nodes associated with the vertices of G and connects the V-node of a vertex v to the proper allocation node of v in T. Also, it uses node attributes to store the type (S, P, Q, R, or V) of a node of T and its poles. In this setting, operation areTriconnected can be reduced to a small number of pathLength and type queries on the augmented SPQR tree.

(1) – (4) & (*Security*) They follow immediately from Theorem 2.4.2 and the complexity bounds of the data structure in [37]. □

2.5 Authenticated Geometric Searching

In this section, we consider authenticated data structures for geometric searching problems. Such data structures have applications to the authentication of geographic information systems.

2.5.1 Fractional Cascading

Fractional cascading, originally presented in [23], is a general algorithmic technique used in a broad class of geometric search problems. In fact, fractional cascading is an efficient strategy for solving the *iterative search* problem which is described in the sequel.

Let U be an ordered universe and $\mathcal{C}$= $\{C_1, C_2, ..., C_k\}$ a collection of k catalogs, where each catalog C_i is an ordered collection of n_i elements chosen from U. For any element $x \in U$, the *successor* of x in C_i is defined to be the smallest element in C_i that is equal or greater than x. We say that we *locate* x in C_i when we find the successor of x in C_i. In the iterative search problem, given an element $x \in U$, we want to locate x in each catalog in $\mathcal{C}$.

Let $n = \sum_{i=1}^{k} n_i$ be the total number of stored elements. The straightforward solution is to perform k separate searches: the search in catalog C_i can be performed in time $O(\log n_i)$ by binary search. The total time needed is $O(k \log n)$. An alternative approach is to merge the k catalogs into a master catalog M and keep a correspondence dictionary between positions in M and positions in each C_i. Using binary search on this merged catalog, we solve the problem in $O(k + \log n)$ time, but we pay the overhead of increasing the storage from $O(n)$ to $O(kn)$.

Fractional cascading succeeds in achieving an $O(k + \log n)$ time complexity for iterative search, while keeping the storage linear. In comparison with the straightforward solution, we can see that if, for instance, all catalogs have the same size and $k = O(\sqrt{n})$ or, even $k = O(\log n)$, we have a time improvement of $O(\log n)$. The key point is how the catalog correlation—that guides the search between incident catalogs—is designed and still the storage is kept linear.

We now present the fractional cascading framework as it usually appears in applications. The original work in [23] covers a more general model which is mostly interesting from a theoretical point of view.

Given a collection $\mathcal{C}$ of catalogs, we consider a one-to-one correspondence between the catalogs in $\mathcal{C}$ and the nodes of a graph G. Let G be a *single source* directed acyclic graph, without multiple edges, that has bounded degree, i.e., each node of G has both in-degree and out-degree bounded by a constant d. Each node v of G is associated with a catalog C_v. G is called a *catalog* graph. Given a catalog graph G, we define $\mathcal{Q}(G)$ to be the family of all connected subgraphs $Q = (V, E)$ of G that contain s but do not contain any other node (except s) having zero in-degree in Q. The iterative search problem for the catalog graph G can then be restated as: given an element $x \in U$ and a member $Q = (V, E)$ of $\mathcal{Q}(G)$, locate x in C_v for all $v \in V$. We refer to Q as the *query* graph.

Let k be the number of vertices of G and let n be the total number of elements in the catalogs of G. In [23], it is showed that, using the fractional cascading technique, we can build a data structure over G in $O(n)$ time, using $O(n)$ space, and solve the iterative search problem in time $O(k+\log n)$. We briefly describe the data structure used and the fractional cascading technique.

The Fractional Cascading Data Structure. Each catalog C_v is augmented to a catalog A_v by storing some extra elements. In A_v, elements in C_v are called *proper* and the other (extra) elements are called *non-proper*. Augmented catalogs that correspond to adjacent nodes of G are connected via *bridges*. Let $e = (u, v)$ be an edge of G. A bridge connecting A_u and A_v is a pair (y, z) associating two non-proper elements y and z, where $y \in A_u$, $z \in A_v$ and $y = z$. Elements y and z have references to each other. Each non-proper element y belongs to exactly one bridge. Two neighboring catalogs A_u and A_v are connected through at least two extreme bridges that correspond to non-proper elements $+\infty$ and $-\infty$ respectively.

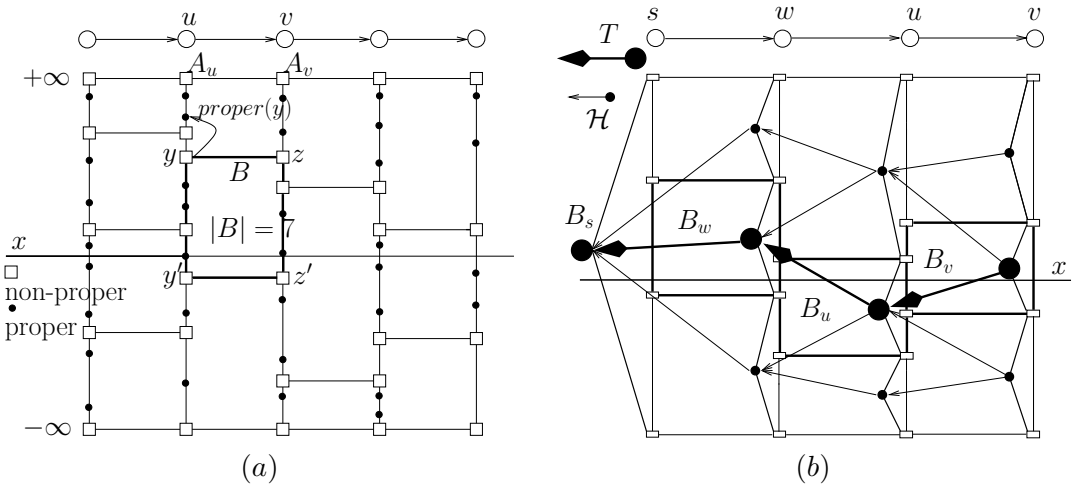

Figure 2.5: (a) The fractional cascading data structure over a path. Squares and dots represent non-proper and proper elements respectively. Edge (u, v) has three blocks. (b) Inter-block hashing: DAG $\mathcal{H}$ defines the second level hashing. For any query element x, any query graph Q and any traversal of Q, the target blocks define a tree T.

Each pair of neighboring bridges (y, z), (y', z') of edge (u, v) defines a *block* B which contains all elements of A_u and A_v lying between the two bridges. If $y' \leq y$, then bridges (y, z) and (y', z') are respectively called the *higher* and the *lower* bridge of B. The size $|B|$ of a block B is the number of the elements (both proper and non-proper) that it contains.

Block sizes constitute a crucial parameter for the performance of the data structure. If n is the total size of the original catalogs, i.e., the total number of proper elements, then, as shown in [23], the total number of non-proper elements is $O(n)$ only if block sizes are proportional to the bounded degree d of G. Thus, in fractional cascading blocks are chosen to have bounded size both from above and below: for each block B, $\alpha \leq |B| \leq \beta$, where α and β are some constants proportional to d. Each non-proper element $z \in A_v$ is associated (by storing a reference) to its next proper element $proper(z)$ in A_v, i.e., its successor in the original catalog C_v. Given z, $proper(z)$ can be retrieved in constant time (e.g., z points to $proper(z)$). If z is a proper element, then we define $proper(z) = z$. Figure 2.5(a) describes the data structure built over a path G. We, finally, describe how this data structure is used to locate query elements.

The Location Process. Suppose that we know the successor, say l, of query element x in an augmented catalog A_u. In $O(1)$ time, we can locate x in the corresponding original catalog C_u, by just retrieving $proper(l)$. Moreover, if $e = (u, v) \in E$, we can locate x in A_v as follows: starting from l, we traverse A_u moving to higher elements until a bridge, say (y, z), is reached that connects to A_v, we then follow that bridge and finally traverse A_v moving to smaller elements until x has been located. Bridge (y, z) is called the *entrance* bridge of catalog A_u. In this way, the search is propagated to a neighboring new node by means of a block B (whose higher bridge is (y, z)) in time $O(|B|) = O(d) = O(1)$. Thus, having located x in an original catalog, we can locate x in an adjacent original catalog in constant time.

Given the query value x and a query graph Q, we initially perform a binary search to locate x at the augmented catalog A_s, where s is the root of G, in time $O(\log n)$. Recall that the query graph Q always contains the root s. Starting from node s, we traverse Q and visit all of its nodes once. Given Q, such a traversal of Q can be performed by considering any topological order of Q. A move from a node u to an adjacent one v, corresponds to the procedure described above: having located x in C_u, we locate x in C_v in constant time. That is, we traverse edge (u, v) by moving through the entrance bridge of A_v. By traversing the query graph Q in this a way, we solve the iterative search problem in $O(kd + \log n)$, where k is the number of vertices of Q, n is the total number of proper elements in the catalogs of G and d is the bounded degree of G (a constant).

2.5.2 Authentication Scheme for Fractional Cascading

We now move to the authentication of the iterative search problem that is solved using the fractional cascading data structure. The idea is to construct a hashing scheme over the data structure, so that a digest is computed and signed by the source, and short answer authentication information is provided for any iterative search query for the catalog graph G. We next describe our authenticated data structure $\mathcal{D}$ for fractional cascading.

Let h be a cryptographic collision-resistant hash function. We assume that a set of rules have been defined, so that h can operate on elements of catalogs, nodes of graph G and previously computed hash values. The hashing scheme can be viewed as a two level hashing structure, built using the path hash accumulator scheme: *intra-block* hashing is performed within each block defined in the data structure, and *inter-block* hashing is performed through all blocks of the data structure. We next describe these hashing structures.

Intra-block Hashing. Consider any edge (u, v) of G, i.e., u is one of the parents of v. Also, consider any two neighboring bridges (y', z') and (y, z) that define block B, where $y, y' \in A_u$ and $z, z' \in A_v$. We define P to be the sequence of elements of B that exist in A_v plus the non-proper elements of the corresponding bridges that lie in A_v. That is, $P = \{p_1, p_2, ..., p_t\}$ is an increasing sequence, where, if $z' \leq z$, $p_1 = z'$ and $p_t = z$. We refer to P as the *hash side* of B. Using the path hash accumulator scheme, we compute the digest $D(P)$ of sequence P. For each element p_i, we set $\mathcal{N}(p_i) = \{p_i, proper(p_i), v\}$, and, in this way, the path hash accumulator can support authenticated membership queries (and authenticated path property queries: here, one property of P is the corresponding node v).

We iterate the process for all blocks defined in the data structure: for each block B having a hash side P in A_v, H_B is the hash path accumulation $D(P)$ of sequence P. We also define B_s to be a fictitious block, the augmented catalog A_s. The hash side of B_s is the whole block itself, so H_{B_s} is well defined. All the used path hash accumulators define the first level hashing structure.

Inter-block Hashing. The second level hashing structure is defined through a directed acyclic graph $\mathcal{H}$ that is defined over blocks. That is, nodes of $\mathcal{H}$ are blocks of the data structure. Suppose that w is a parent of u and u is a parent of v in G. If B is a block of edge (u, v), then we add to the set of edges of $\mathcal{H}$ all the directed edges (B, B'), where B' is a block of edge (w, u) that shares elements from A_u with B. Additionally, if v is a child of the root s in G, then for all blocks B that correspond to edge (s, v), we add to the set of

edges of $\mathcal{H}$ the directed edge (B, B_s). This completes the construction of graph $\mathcal{H}$. B_s is the unique root of $\mathcal{H}$. Figure 2.5(b) shows the graph $\mathcal{H}$ that corresponds to a catalog path.

Each block (node) B of $\mathcal{H}$ is associated with a label $L(B)$. If B is a source node (leaf) in $\mathcal{H}$, then $L(B) = H_B$. If B is the parent of blocks $B_1, \ldots, B_t$ in $\mathcal{H}$, listed in some fixed order, then $L(B)$ equals the path hash accumulation over sequence $B_1, \ldots, B_t$ using $\mathcal{N}(B_i) = \{L(B_i), H_B\}$. Here, the path hash accumulator is used simply for authenticating membership in a set (like a Merkle tree; thus, no path property is used). This hashing scheme over $\mathcal{H}$ corresponds to the second level hashing structure. Finally, we set $D(\mathcal{D})$= $L(B_s)$ to be the digest of the entire data structure $\mathcal{D}$ (which is signed by the data source).

2.5.3 Answer Authentication Information

Given a query x and a query graph Q, we describe now what is the authentication information given to the user. If v is a node of Q, let s_v be the successor of x in C_v. In the location process, to locate x in A_v, we find two consecutive elements y and z of A_v such that $y \leq x < z$, where each of y and z may be either proper or non-proper. They are both elements of a block B such that the entrance bridge of A_v is the higher bridge of B. Observe that z is the successor of x in A_v and that $s_v = \mathit{proper}(y)$ when $y = x$, or $s_v = \mathit{proper}(z)$ when $y < x$. We call z and B, the *target element* and the *target block* of A_v, respectively.

Two useful observations are that: (1) in the location process, the traversal of the query graph Q is chosen so that each node of Q is visited once, and (2) any two target blocks visited by the location process that correspond to incident edges in Q share elements of the common augmented catalog, and, thus, are adjacent in graph $\mathcal{H}$. It follows that all the target blocks define a subgraph T of $\mathcal{H}$. T consists of the all target blocks and the edges of $\mathcal{H}$ that connect neighboring target blocks (Figure 2.5(b)).

Lemma 2.5.1. *For any query graph Q, graph T is a tree.*

Proof. Consider the topological order used to define the traversal of the query graph Q. This topological order defines a directed subtree T_Q of Q. There is an one-to-one correspondence between edges of T_Q and target blocks, i.e., between edges of T_Q and nodes of T. □

For any node v of Q, let z_v be the target element of A_v and B_v the target block of A_v. The answer authentication information consists of:

1. *Intra-block:* for each node v of Q, the target element z_v of A_v and a verification sequence p_v from z_v up to the path hash accumulation of the hash side of B_v, and

2. *Inter-block:* for every node (or target block) B_v of T that is not a leaf, the verification sequences from every child of B_v in T up to the path hash accumulation $L(B_v)$.

Lemma 2.5.2. *If n is the total number of proper elements in the catalogs of $\mathcal{C}$ and d is the bounded degree of G, then for any query graph Q of k nodes, the size of the answer authentication information is $O(\log n + k \log d) = O(\log n + k)$.*

Proof. The hash side of B_s has size $|A_s| = O(n)$ and the hash side of any other target block has size $O(d)$. Thus, the intra-block answer authentication information consists of k verification sequences, $k-1$ of length $O(\log d)$ and one of length $O(\log n)$, and, thus, has size $O(\log n + k \log d) = O(\log n + k)$. For the inter-block answer authentication information, recall that G and, thus, both Q and T_Q, have out-degree bounded by d and that every target block can share elements with at most $O(d)$ other target blocks. Thus, $\mathcal{H}$ and T have in-degree bounded by $O(d)$. Now, all, but $L(B_s)$, the second level path hash accumulations are built over sequences of length $O(d)$. $L(B_s)$ is built over at most dn blocks that share elements with A_s. Observe that there is a one-to-one correspondence between inter-block verification sequences and edges in T. It follows that the inter-block answer authentication information consists of $k-2$ verification sequences of size $O(\log d)$ and one of size $O(\log n)$, thus, has $O(\log n + k \log d)$ size. In total, since d is a constant, the answer authentication information is of size $O(\log n + k)$. □

2.5.4 Verification of an Answer

For a given query element x and query graph $Q = (V, E)$, we assume that the answer given to the user is a set $A = \{(a_v, v) : v \in V\}$, where a_v is claimed to be the successor of x in C_v. The answer authentication information consists of two verification sequences for each node (target block) of tree T: one intra-block and one inter-block. These sequences form a hash tree in our two-level hashing scheme. The verification process is defined by this hash tree. Intuitively, an intra-block verification sequence of a target block B_v provides a *local proof* that a_v is the successor of x in C_v, and then, all these local proofs are accumulated through inter-block verification sequences into the signed digest.

Given elements x, y, y', z, z' and a node v of Q, consider the predicates: (1) $y \leq x < z$, (2) y and z are consecutive elements in A_v, (3) $y = x$ and $y' = \mathit{proper}(y)$ in A_v and (4) $y < x$ and $z' = \mathit{proper}(z)$ in A_v. If (1), (2) and (3) hold simultaneously, then they constitute a proof that the successor of x in C_v is y', whereas if (1), (2) and (4) hold simultaneously, they constitute a proof that the successor of x in C_v is z'. Such a proof must be provided for every v of Q.

Given A, x and the answer authentication information, the user first checks if there is any inconsistency between values a_v and z_v for every v of Q with respect to the two possible proofs above. Observe that, by the answer authentication information, the user knows for each node v of Q the target element z_v and the corresponding element y_v, such that $y_v < z_v$ and y_v and z_v are consecutive elements in A_v. If there is at least one inconsistency, the user rejects the answer. Otherwise, all that is needed is to verify the signed digest $D(\mathcal{D})$ of the data structure. Observe, that the user possesses all the data needed for the computation of the signed digest. If the digest is verified, based on the collision-resistance property of the hash function used in the scheme, the user has a proof that the answer is correct.

Lemma 2.5.3. *If n is the total number of proper elements in the catalogs of $\mathcal{C}$, then for any query graph Q of k nodes, the answer verification time is $O(\log n + k \log d) = O(\log n + k)$, where d is the bounded degree of G.*

Proof. Recall that the verification time of a path hash accumulator is proportional to the size of the verification sequence. □

If the digest is verified, then based on the collision-resistance property of the hash function h, and, in particular, the security of the path hash accumulator, the user has a proof that the answer is correct: for each v of Q, the user can verify all the three conditions previously discussed. A faulty answer can lead to a forged proof only if some collisions of h have been found: the responder needs to break the security of the path hash accumulator in authenticating membership queries, which is further reduced to finding collisions of the cryptographic hash function h in use.

Lemma 2.5.4. *For any catalog graph G of k nodes and of total size n, both intra-block and inter-block hashing schemes can be computed in $O(n)$ time using $O(n)$ storage.*

Proof. G has bounded in-degree by $d = O(1)$ and every target block can share elements with at most $\beta = O(1)$ other blocks. Moreover, the path hash accumulation of a sequence of length m can be computed in $O(m)$ time and space. □

We have thus proved the following theorem.

Theorem 2.5.1. *Let G be the catalog graph for a collection $\mathcal{C}$ of t catalogs and n be the total number of elements stored in $\mathcal{C}$, where $t \leq n$. If G is of bounded degree, then the authenticated fractional cascading data structure $\mathcal{D}$ for G solves the authenticated iterative search problem for G, achieving the following performance:*

1. *$\mathcal{D}$ can be constructed in $O(n)$ time and uses $O(n)$ storage;*

2. *given a query element x and a query graph Q with $k \leq t$ vertices, x can be located in every catalog of Q in $O(\log n + k)$ time;*

3. *the answer authentication information has size $O(\log n+k)$ and the answer verification time is $O(\log n + k)$.*

2.5.5 Applications

Our authenticated fractional cascading scheme can be used to design authenticated data structures for various fundamental two-dimensional geometric search problems, where iterative search is implicitly performed (see [24]). In all of these problems, the underlying catalog graph has degree bounded by a small constant, and, typically, the graph itself has a tree structure. We next describe how the authentication of the iterative search problem can be extended to provide authentication of this broad class of queries that involves searching in multi-catalogs that are organized in a tree structure.

Authentication scheme. The idea, here, is to extend the hashing scheme of the fractional cascading data structure over the graph structure in which catalogs are organized. In essence, what we need to additionally authenticate is that the correct subdigraph of the catalog graph is accessed by the responder and used to generate the answer. That is, so far (for the iterative search only) we assumed that this catalog subgraph is part of the query (query graph Q). In the applications of the fractional cascading data structure, however, this subgraph is not known in advance, but it is rather generated on-the-fly as the answer is being produced. Accordingly, in order to verify the answer, the user needs first to authenticate that the correct (authentic) subgraph is generated by the responder and that the final answer corresponds to this correct subgraph of catalogs. Given that in our authentication schemes verification is performed in a bottom-up fashion, the user essentially will first authenticate the iterative search (as described in the previous subsections), and then he will authenticate that the subgraph that corresponds to the iterative search is authentic (all nodes that should have been included in the graph have been included, but no extra nodes are included). This second verification step (subgraph authentication) can be easily performed as follows.

The catalog graph is accessed by some search algorithm. The key property that we wish our authentication scheme to satisfy is to authenticate this search procedure. This is done by hashing over the graph structure (bottom-up hashing over the graph) as follows. Let v

be a node of the catalog graph G—recall, G is a directed acyclic graph—that has $u_1, \ldots, u_\ell$ successor nodes, where ℓ is a constant (because the catalog graph G is of bounded degree). Let d_v be the data that is used by the search algorithm to advance the search from node v to one or more successor nodes. Typically (essentially for catalogs organized as trees) search is performed over an ordered data set, and, in this case, d is a sequence of ℓ (or $\ell - 1$) keys (members of the set) that are used along with the search items (defined by the query) to decide in which node(s) to advance the search. Node v is then associated with a hash value h_v, defined as

$$h_v = h(h_{u_1}, \ldots, h_{u_\ell}, h(d_v)).$$

Thus, we have a recursive definition of the digest of the hashing scheme at the source of G. Depending on exact format of data d_v, additional improvements (up to some constant) may be considered in producing the hash value (digest) $h(d_v)$. Given, a traversed subgraph of G, now, the answer authentication information additionally contains the hash values and search information (i.e., keys) that are needed to recompute the final data digest. Given this authentication scheme, the verification of the answer first involves the verification of iterative search problem, as described earlier in the chapter, and then the verification of the search subgraph of catalogs (depending on the exact query problem, the order may be swapped). The connection of the two individual hashing schemes can be easily performed at the root of the entire hashing structure: simply by hashing together the two individual digests to generate the digest of the entire hashing scheme.

Given this extra layer of hashing and this extension of our authentication scheme, we obtain authenticated versions of any data structure that uses iterative search over set of catalogs organized as nodes of a DAG. The following results are obtained by using the results in [24] and our the authentication schemes of this section, where n denotes the problem size.

Corollary 2.5.1. *There is an authenticated data structure for answering line-intersection queries on a polygon with n vertices that can be constructed in $O(n \log n)$ time and uses $O(n \log n)$ storage. Denoting with k the output size, queries are answered in $O(\log n + k)$ time; the answer authentication information has size $O((k+1) \log \frac{n}{k+1})$; and the answer verification time is $O((k+1) \log \frac{n}{k+1})$.*

Corollary 2.5.2. *There are authenticated data structures for answering ray shooting and point location queries on a planar subdivision with n vertices that can be constructed in $O(\log n)$ time and use $O(n \log n)$ storage. Queries are answered in $O(\log n)$ time; the answer authentication information has size $O(\log n)$; and the answer verification time is $O(\log n)$.*

Corollary 2.5.3. *There are authenticated data structures for answering orthogonal range search, orthogonal point enclosure and orthogonal intersection queries that can be constructed in $O(n \log n)$ time and use $O(n \log n)$ storage, where n is the problem size. Denoting with k the output size, queries are answered in $O(\log n + k)$ time; the answer authentication information has size $O(\log n + k)$; and the answer verification time is $O(\log n + k)$.*

2.6 Conclusions

In this chapter, we have examined the problem of designing efficient authenticated data structures for broad classes of queries. We have developed the path hash accumulator, a new authentication scheme for general decomposable queries over sequences of data elements and, in particular, queries about properties of subsequences that involve any associate operation applied over their elements. Using this authentication scheme, we then design new authenticated data structures for graph queries (e.g., path and connectivity queries) or search problems over two-dimensional geometric objects (e.g., point location and range search). Authentication of graph queries is performed by authenticating certain path properties in some tree graphs that are specially designed for the graph in question. Authentication of geometric search problems is performed by authenticating the general fractional cascading framework that solves the iterative search problem. Our authentication techniques are efficient and introduce asymptotically no extra overhead to the underlying search structure.

An interesting open problem is the design of dynamic versions of our authenticated data structures based on fractional cascading.

Much of the technical content of this chapter appears in publication [59].

Chapter 3

Authentication of Set-Membership Queries

3.1 Introduction, Contributions and Previous Work

Deriving lower bounds on the complexity of computational problems is an important, often difficult, task. On one hand, lower bounds put limits on the efficiency that one can hope for, on the other hand, when lower bounds asymptotically match upper bounds derived by known algorithmic constructions, one proves the optimality of these algorithms. In this case, exact analysis and the study of constant factors often replace asymptotic analysis for the need to explore the best possible algorithmic efficiency for the problem in study. Finally, lower bounds proofs can often provide a characterization of the problem and give an explanation of the computational difficulty that is intrinsic in it.

In this chapter, we walk along these lines. We present a unified analysis and design of algorithms and data structures for two important, and seemingly unrelated, algorithmic problems in the area of information security: (*i*) *data authentication through cryptographic hashing*, i.e., the authentication of membership queries in the presence of data replication at untrusted directories, and (*ii*) *multicast key distribution using key-graphs*, i.e., the distribution of cryptographic keys by the controller of a dynamic multicast group. We provide logarithmic lower bounds on various time and space cost measures for these problems. In view of these lower bounds, we develop new efficient data structures for these problems and give an accurate analysis of their performance, taking into account constant factors in the leading asymptotic term.

Our unified approach is based on the definition of an abstract and generic class of computational problems, where a directed acyclic graph (DAG) describes the computation of a collection of output values from an input set of n elements. In particular, we introduce *hierarchical data processing (HDP)* problems and study their computational bounds. An HDP problem is related to maintaining, in a systematic way, a dynamic collection of elements along with an associated set of values. Updates of elements generally result in updates of the associated values and queries on elements return subsets of the associated values. In either case, for a problem of this type, the computations that are carried out after an update or a query are all performed sequentially, according to an associated hierarchy, which in turn is expressed by means of a DAG. Accordingly, various cost measures related with the time or space complexity of these computations depend on certain structural properties of the underlying DAG. We define several cost measures for subgraphs of a DAG that characterize the time and space complexity of queries and update operations in an HDP problem. We prove $\Omega(\log n)$ lower bounds on these cost measures using a reduction from the problem of searching by comparisons in an ordered set. Furthermore, in view of these logarithmic lower bounds, we design a new randomized DAG scheme for HDP problems that is based on a variation of the skip-list. Our new DAG scheme achieves computational efficiency which is very close to the optimal with respect to the leading logarithmic terms.

We motivate the study of the class of HDP problems by two seemingly different algorithmic problems related with applications in the area of information security. Both problems share the property that the involved computations are performed according to the hierarchy induced by a DAG, that is, they involve some type of hierarchy-based data processing. The first application is the model of *hash-based authenticated data structures* that has been recently proposed for data authentication in distributed and untrusted environments. In this model, queries on a data set are answered by untrusted entities and not by the source and owner of the data set, in a way where each answer contains information that can been used to provide an cryptographic proof about the validity of the answer. We focus on *authenticated dictionaries*, where membership queries are asked about a set, and consider the case where authentication is achieved by hierarchically applying cryptographic hashing over the data set. The second application is the problem of *multicast key distribution using key-graphs*. Here, a dynamic group of users share a set of keys so that they can securely implement multicast transmissions by means of private-key encryption. Using key-graphs, any update in the group memberships results in changing and securely redistributing to the users some subset of the keys.

We obtain new results for these two information security problems by first showing that

they can be modeled by a HDP problem and by appropriately applying to their domain the general lower bounds and our new DAG scheme. For data authentication through cryptographic hashing, we model (the design of) any authenticated dictionary as an HDP problem where only various costs related to authentication, the *authentication overhead*, are considered. Using our framework, we prove that any hash-based authenticated dictionary incurs logarithmic on its size authentication cost in the worst case and we present a new authenticated dictionary with authentication cost closer to the theoretical optimal. Similarly, we show that a broad class of multicast key distribution protocols can be viewed as HDP problems and we thus prove that any such protocol has a logarithmic on the size of the group time or communication complexity in the worst case.

Interestingly, the proof of our lower bounds is based on a reduction from the problem of searching by comparison in an ordered set of size n to an HDP problem of the same size, which establishes a relationship between these two types of problems. This relationship further characterize the computational difficulty of the security problems that we study. Additionally, through this reduction, our new DAG scheme provides us with a new skip-list version where searches have expected cost that is closer to the theoretically optimal. Our study of HDP not only offers a unified treatment of the two information security problems in consideration, but also provides an new interesting theoretical framework in problem analysis and algorithm and data structure design. Indeed, the class of HDP problems can in principle be used to model any hierarchy-based data processing problem in terms of lower bounds and efficient constructions.

In the rest of the section we briefly describe the class of problems that we study, we summarize our contributions in the analysis of the complexity of HDP problems and in data structure design and we present an overview of the new results that we get by applying our framework in the two algorithmic problems in the domain of information security. We also review previous work.

3.1.1 Hierarchical Data Processing

We introduce an abstract and generic class of problems, the *hierarchical data processing* (HDP) problems. This class of problems models computations over a dynamic set of n elements, where all computations are lead by the hierarchy induced by a directed acyclic graph (DAG), so that various costs depend on certain structural properties of this underlying DAG. These problems share the following characteristics. Associated with the elements is a structured collection of values, organized according to the DAG. As elements change over

time, the values are accordingly updated. Additionally, queries on elements are issued, where typically the answer of a query is a subset of the associated values.

More specifically, in an HDP problem a dynamic collection of elements, maintained by update operations, is associated with a set of values, which in turn is organized by means of a DAG. Data processing involves the computation and update of the set of values after every update in the elements. In addition, queries on elements are answered, again, by accessing and processing data values through traversals of the DAG. All operations on elements and associated values involve processing of data according to the hierarchy that the DAG induces. Additionally, the complexity of all computations and all costs parameters related to the problem depend on specific properties and the structure of the DAG in use.

Previous Work. We are not aware of any previous systematic study of the class of HDP problems.

Our Results. We define several cost measures for subgraphs of a DAG that characterize the space and time complexity of queries and update operations in an HDP problem. For a problem of size n, we relate each of these cost measures to the number of comparisons performed in the search of an element in an ordered set of size n. Through this reduction, we prove an $\Omega(\log n)$ lower bound on the space and time complexity of query and update operations in any HDP problem. We also show that for this class of problems trees are optimal DAG structures compared with general DAGs. In view of the logarithmic lower bound and the optimality of tree DAGs for HDP problems, we also design a new randomized DAG, called *multi-way skip-list DAG scheme*, which is based on a variation the skip-list data structure. We give a detailed analysis of the cost measures of our DAG scheme taking into account the constant factor on the leading asymptotic term and we show that it achieves structural properties that are close to optimal. Accordingly, we get that multi-way skip-lists, when used to lead the computations of any HDP problem, achieve efficiency in terms of time and space complexities.

Hierarchical data processing problems and their computational bounds are presented in Section 3.2. In Section 3.3, we present and analyze our multi-way skip-list DAG scheme.

3.1.2 Data Authentication and Authenticated Data Structures

Data structures are designed to organize a collection of data, so that searching in the collection and answering queries about the data are performed efficiently. Implicitly, the owner and the user of the data structure are assumed to be the same entity. An important security problem arises when this assumption is abandoned. For instance, with the advent of Web

services and pervasive computing, a data structure can be controlled by an entity different than the owner or the user of the data. Additionally, data replication applications achieve computational efficiency by caching data at servers near users, but they present a major security challenge. Namely, how can a user verify that the data items replicated at a server (e.g., the answer to a query) are the same as the original ones generated by the data source? The naive approach of directly applying traditional message authentication techniques (like digital signatures or message authentication codes) in this data authentication problem either fails or lacks efficiency and scalability. For instance, digitally signing the answer to any query of a data structure is not viable for the set of possible answers can be unbounded, especially for dynamic data sets that evolve over time.

Authenticated data structures (ADSs) capture exactly this, non-conventional (and closer to today Internet's reality) new data structuring paradigm. A data structure is controlled by an entity that is *not* the creator of the data. That is, not the data source but rather an entity *not trusted* to a user issuing queries answers these queries. ADSs solve the security problem of data authentication in these untrusted distributed environments and receive more and more attention. They support *authenticated queries*: they allow the user to verify the validity of the answer, i.e., either accept the answer as authentic or reject it.

In particular, an ADS is a distributed model of computation where a *directory* answers queries on a data structure on behalf of a trusted *source* and provides to the *user* a cryptographic proof of the validity of the answer. The source signs a *digest* (i.e., a cryptographic summary) of the content of the data structure and sends it to the directory. Ideally, the digest has $O(1)$ size. This signed digest is forwarded by the directory to the user together with the proof of the answer to a query. To verify the validity of answer, the user computes the digest of the data from the answer and the proof, and compares this computed digest against the original digest signed by the source.

Cost parameters for an ADS include the space used (for the source and directory), the update time (for the source and directory), the query time (for the directory), the digest size, the proof size and the verification time (for the user). In the important class of *hash-based authenticated data structures*, the digest of the data set is computed by *hierarchical hashing*, i.e., by hierarchically applying a cryptographic hash function over the data set.

Previous Work. Early work on ADSs was motivated by the *certificate revocation* problem in public key infrastructure and focused on *authenticated dictionaries*, on which membership queries are issued. The *hash tree* scheme introduced by Merkle [95] can be used to implement a static authenticated dictionary. A hash tree T for a set S stores cryptographic hashes of the elements of S at the leaves of T and a value at each internal node, which is the

result of computing a cryptographic hash function on the values of its children. The hash tree uses linear space and has $O(\log n)$ proof size, query time and verification time. A dynamic authenticated dictionary based on hash trees that achieves $O(\log n)$ update time is described in [103]. A dynamic authenticated dictionary that uses a hierarchical hashing technique over skip-lists is presented in [52]. This data structure also achieves $O(\log n)$ proof size, query time, update time and verification time. Other schemes based on variations of hash trees have been proposed in [17, 46, 72]. The software architecture and implementation of an authenticated dictionary based on skip-lists is presented in [58]. A distributed system realizing an authenticated dictionary and an empirical analysis of its performance in various deployment scenarios are described in [54]. The authentication of distributed data using web services and XML signatures is investigated in [118] and *prooflets*, a scalable architecture for authenticating web content based on authenticated dictionaries, are introduced in [21].

An alternative approach to the design of authenticated dictionary, based on the *RSA accumulator*, is presented in [57]. This technique achieves constant proof size and verification time and provides a tradeoff between the query and update times. For example, one can achieve $O(\sqrt{n})$ query time and update time. In [2], the notion of a *persistent authenticated dictionary* is introduced, where the user can issue historical queries of the type "was element e in set S at time t". A first step towards the design of more general ADSs (beyond dictionaries) is made in [34] with the authentication of relational database operations and multidimensional orthogonal range queries. In [86], a general method for designing ADSs using hierarchical hashing over a search graph is presented. This technique is applied to the design of static ADSs for pattern matching in tries and for orthogonal range searching in a multidimensional set of points. Efficient ADSs supporting a variety of fundamental search problems on graphs (e.g., path queries and biconnectivity queries) and geometric objects (e.g., point location queries and segment intersection queries) are presented in [59]. This paper also provides a general technique for the design of ADSs that follow the *fractional cascading* paradigm. Work related to ADSs includes [18, 32, 55, 83, 84]. Related to ADSs is also recent work on zero knowledge sets and consistency proofs [98, 108] that model data authentication in a more adversarial environment, where the data source is not considered trusted *per se*. These schemes use significantly more computational resources than ADSs.

No previous study on the cost of ADSs has been made. As we will see later, from our study we get the following. The computational overhead incurred by an ADS over a non-authenticated data structure consists of: (1) the additional space used to store authentication information (e.g., signatures and hash values) in the data structure and in the proof of the answer, and (2) the additional time spent performing authentication computations

(e.g., computing signatures and cryptographic hashes) in query, update and verification operations. Since cryptographic operations such as signatures and hashes are orders of magnitude slower than comparisons and a single hash value is relatively long, the authentication overhead dominates the performance of an ADS. All the existing hash-based authenticated dictionaries have logarithmic query, update and verification cost and logarithmic proof size.

We thus address the following question: can the authenticated version of a data structure (i.e., authenticating membership queries) be more efficient then the non-authenticated one (i.e., answering membership queries)[1]? Considering only dictionaries, the existence of tree-based schemes, schemes with logarithmic authentication costs (proof size, verification time, update/query time) where hashing is performed according to the search tree (data structure) (e.g., [103, 52]), shows that answering authenticated membership queries is at most as expensive as answering non-authenticated membership queries using search trees. But can we do better? Already in the introduction of authenticated dictionaries, Naor and Nissim [103] posed as an open problem the question of whether one can achieve sub-logarithmic authentication overhead for dictionaries. We answer this question negatively for hash-based ADSs.

Our Results. We present the first study on the cost of ADSs, focusing on dictionaries. We model a hash-based dictionary ADS as an HDP problem. We consider a very general authentication technique where hashing is performed over the data set in *any possible* way and where *more than one* digests of the data structure are digitally signed by the source. Applying our results from Section 3.2 in this domain, we prove the first nontrivial lower bound on the authentication cost for dictionaries. In particular, we show that in any hash-based authenticated dictionary of size n where the source signs k digests of the data set, any of the authentication costs (update/query time, proof size or verification time) is $\Omega(\log \frac{n}{k})$ in the worst case. Thus, the optimal trade-off between signature cost and hashing cost is achieved with $O(1)$ signature cost and $\Omega(\log n)$ hashing cost. In this case, we show that hash-based authenticated dictionaries of size n incur $\Theta(\log n)$ complexity.[2] We also show that among all DAGs, trees are optimal when used for hashing. Also, our skip-list structure from Section 3.3 can be used to implement an efficient authenticated dictionary, where certain cost parameters are reduced with respect to previous constructions. Our results on

[1] Clearly, for the question to make sense, we completely separate the notions of authenticating and answering membership queries.

[2] Interestingly, the use of a different cryptographic technique, namely the use of one-way accumulators in [57], achieves constant proof size and verification time, but involves more expensive update costs and more expensive primitive operations (exponentiations) with respect to the efficient cryptographic hashing.

authenticated dictionaries are described in Section 3.4.

3.1.3 Multicast Key Distribution

Multicast key distribution (or multicast encryption) is a model for realizing secrecy in multicast communications among a dynamic group of n users. To achieve secrecy, one needs to extend the conventional point-to-point encryption schemes to the multicast transmission setting. Namely, the users share a common secret key, called *group-key*, and encrypt multicast messages with this key, using a secret-key (symmetric) encryption scheme. When changes in the multicast group occur (through additions/deletions of users), in order to preserve (forward and backward) security, the group-key needs to be securely updated.

In general, a *group controller* (physical or logical entity) is responsible for distributing an initial set of keys to the users. Each user possesses his own secret-key (known only to the controller), the group-key and a subset of other keys. Upon the insertion/removal of a user into/from the group, a subset of the keys of the users are updated. Namely, new keys are encrypted by some of the existing keys so that only legitimate users in the updated group can decrypt them. The main cost associated with this problem is the number of messages that need to be transmitted after an update. Additional costs are related to the computational time spent for encrypting and decrypting the keys.

Previous Work. Many schemes have been developed for multicast key distribution. We focus on the widely studied *key-graph* scheme, introduced in [143, 147], where constructions are presented for key-graphs realized by balanced binary trees such that $O(\log n)$ messages are transmitted after an update, where n is the current number of users. Further work has been done on key-graphs based on specific classes of binary trees, such as AVL trees, 2-3 trees and dynamic trees. See, e.g., [60, 56, 127]. In [19], the first lower bounds are given for a restricted class of key distribution protocols, where group members have limited memory or the key distribution scheme has a certain structure-preserving property. In [135], an amortized logarithmic lower bound is presented on the number of messages needed after an update. The authors prove the existence of a series of $2n$ update operations that cause the transmission of $\Omega(n \log n)$ messages. Recently, a similar amortized logarithmic lower bound has been shown in [99] for a more general class of key distribution protocols, where one can employ a pseudorandom generator to extract (in a one-way fashion) two new keys from one key and one can perform multiple nested key encryptions. Pseudorandom generators for this problem were first described in [20], where the number of messages are decreased from $2 \log n$ to $\log n$.

Our Results. We show that the multicast key distribution problem using key-graphs is an HDP problem. Applying our results from Sections 3.2 and 3.3 to this domain: (*i*) we perform the first study of general key-graphs (other than trees) and show that trees are optimal structures; (*ii*) we prove an exact worst-case logarithmic lower bound on both the communication cost (number of messages) and the computational cost (cost due to encryption/decryption) of any update operation, the first of this type; and (*iii*) we present a new scheme (tree DAG) that achieves costs closer to the theoretical optimal. Note that we give the first lower bounds on the encryption/decryption costs and that our lower bound proof is more generic since it depends on *no certain series* of update operations. In essence, we present the first *exact*, *worst case* logarithmic lower bound for the communication cost of the multicast key distribution problem. All of the previously known lower bounds are *amortized*, i.e., they prove the existence of a sequence of updates that include an expensive (of at least logarithmic cost) one. In contrast, we prove the existence of a single update of at least $\lfloor \log n \rfloor$ communication cost for any instance of the problem. Our lower bound holds also for protocols that use pseudorandom generators or multiple encryption, as in the model studied in [99]. These results are described in Section 3.5.

3.1.4 Skip-Lists

The skip-list, introduced in [120, 121], is an efficient randomized data structure for dictionaries. It is well known that skip-lists correspond to some tree (search) structure, so they constitute an optimal DAG structure for HDP problems.

Previous Work. In [120, 121] it is shown that the expected number of comparisons for a search in a skip-list is $(\log_2 n)/(p \log_2 \frac{1}{p}) + O(1)$, where p is a probability parameter. In the same work, an improved – in terms of number of comparisons – skip-list version gives $\frac{1-p^2}{p \log_2 \frac{1}{p}} \log_2 n + O(1)$ expected comparisons for a search. We are not aware of any improved skip-list based scheme with better logarithmic constant.

Our Results. Through the relation between HDP and searching by comparison we obtain a new version of skip-lists, where the expected number of comparisons in a search is $1.25 \log_2 n + O(1)$, which is closer to the theoretically optimal up to an additive constant term. In particular, for p being the probability parameter of the skip-list, our skip-list version reduces the expected number of comparisons down to $\frac{(1-p)(1+p^2)}{p \log_2 \frac{1}{p}} \log_2 n + O(1)$ [3]. Details for the new multi-way skip-list DAG and the corresponding new skip-list version

[3] For $p = \frac{1}{2}$, the logarithmic constant of the expected number of comparisons for a search drops from 1.5 to 1.25.

are presented in Section 3.6.

3.1.5 Chapter Structure

The chapter is organized as follows. In Section 3.2, we introduce the problems of hierarchical data processing, study their complexity, prove lower bounds on various associated costs and show that tree DAGs are optimal when used for HDP problems. In view of these results, we focus on tree DAGs and in Section 3.3 we design and analyze our new multi-way skip-list DAG scheme for HDP problems that is based on skip-lists and achieves performance close to optimal. In Section 3.4 we apply our results to data authentication problem through hashing and in Section 3.5 to the problem of multicast key distribution using key-graphs. Finally, in Section 3.6 we present the new improved skip-list version. We conclude and discuss open problems in Section 3.7.

3.2 Hierarchical Data Processing and its Theoretical Limits

In this section, we define several structural cost measures for subgraphs of a DAG and prove lower bounds them. Such cost measures are related to the computational complexity of operations in a class of problems that we call *hierarchical data processing* problems. Our lower bounds are naturally translated to complexity results of this type of problems.

3.2.1 DAG Scheme

Before we introduce our new concepts, we define some graph notation. Let $G = (V, E)$ be a directed acyclic graph. For each node v of G, $indeg(v)$ denotes the in-degree of v, i.e., the number of incoming edges of v, and similarly, $outdeg(v)$ denotes the out-degree of v, i.e., the number of outgoing edges of v. A *source* of G is a node v such that $indeg(v) = 0$. A *sink* of G is a node v such that $outdeg(v) = 0$. We denote with $V_{source} \subset V$ the set of *source* nodes of G and with $V_{sink} \subset V$ the set of sink nodes of G. For any edge $e = (u, v)$ in E, node u is a *predecessor* of v and node v is a *successor* of u. A directed path in G connecting node u to some other node is called *trivial*, if every node in the path other than u has in-degree 1. A subgraph H of G is said to be *weakly connected*, if it is connected when one ignores edge directions, *non-trivial*, if it contains no trivial paths, and *complete*[4], if all edges in G connecting nodes of H are edges of H. An edge in G connecting nodes of H that is not an edge of H is called a *missing* edge of H and by $\bar{H}$ we denote the complete

[4]The used term is not confusing: cliques are not defined for DAGs.

graph that we take if we add in H all of its missing edges. For any node v in a DAG G, G_v denotes the subgraph in G whose nodes are connected with v through directed paths that start at v, i.e., they are successor nodes of v in the transitive closure of G and whose edges belong in these directed paths. We say that subgraph G_v is *reachable* from node v. Note that for any node v, graph G_v is a complete graph (having no missing edges).

Definition 3.2.1 (DAG scheme)**.** *A* DAG scheme Γ *is a quadruple* (G, S, n, k)*, where* $G = (V, E)$ *is a directed acyclic graph without parallel edges,* $S \subset V$ *is a set of* special *nodes and* n *and* k *are integers such that:* (i) $|V_{source}| = n$*;* (ii) $|V|$ *is bounded by a polynomial in* n*; and* (iii) $|S| = k$*,* $S \supset V_{sink}$ *and* $S \cap V_{source} = \emptyset$*.*

That is, G has n source nodes and $poly(n)$ nodes in total; G contains a subset of k non-source nodes, called special nodes, that includes all the sink nodes of G.

3.2.2 Cost Measures of a DAG Scheme

We first define three *structural cost measures* for any weakly connected subgraph of a DAG.

Definition 3.2.2 (Structural cost measures for subgraphs)**.** *Let* $H = (V_H, E_H)$ *be a weakly connected subgraph of a DAG* G*. We define the following with respect to* G*:*

1. *The* node size $size(H)$ *of* H *is the number of nodes in* H*, that is,* $size(H) = |V_H|$*;*
2. *the* degree size $indeg(H)$ *of* H *is the sum of the in-degrees (with respect to* G*) of the nodes of* H*, that is,* $indeg(H) = \sum_{v \in H} indeg(v)$*;*
3. *the* combined size $comb(H)$ *of* H *is the sum of its node and degree sizes, that is,* $comb(H) = size(H) + indeg(H)$*;*
4. *the* boundary size $bnd(H)$ *of* H *is the number of edges of* G *that enter nodes of* H *but are not in* H*.*

Whenever it is not clear from the context with respect to which DAG G a structural cost measure is defined, we use subscripts; e.g., $indeg_H(\cdot)$ denotes the degree size with respect to graph H.

Using the above structural cost measures of subgraphs of DAGs, we define three *cost measures* for a DAG scheme Γ.

Definition 3.2.3 (Cost measures of DAG scheme)**.** *Let* $\Gamma = (G, S, n, k)$ *be a DAG scheme, let* s *be a source node of* G*. Let* P_s^t *denote the set of directed paths connecting node* s *to*

node t in G. The associated path π_s *of s is a directed path in G_s that starts at s, ends at a node of S and has the* minimum *combined size among all such paths, i.e., $comb(\pi_s) = \min_{u \in S, p \in P_s^u} comb(p)$. We define the following cost measures for Γ:*

1. update cost $\mathcal{U}(\Gamma)$ *of Γ: $\mathcal{U}(\Gamma) = \max_{s \in V_{source}} comb(G_s)$, i.e., the maximum, over all source nodes in V_{source}, combined size of the subgraph G_s reachable from s;*

2. query cost $\mathcal{Q}(\Gamma)$ *of Γ: $\mathcal{Q}(\Gamma) = \max_{s \in V_{source}} comb(\pi_s) = \max_s \min_{u \in S, p \in P_s^u} comb(p)$, i.e., the maximum, over all source nodes in V_{source}, combined size of the associated path π_s of s;*

3. sibling cost $\mathcal{S}(\Gamma)$ *of Γ: $\mathcal{S}(\Gamma) = \max_{s \in V_{source}} bnd(\pi_s)$, i.e., the maximum, over all source nodes in V_{source}, boundary size of the associated path π_s of s.*

Note that the associated path of a source node s (which is not necessarily unique) is generally not the minimum boundary size path from s to a node in S, because for general graphs minimum combined size does not imply minimum boundary size. In our study, sibling costs defined as above are thus linked to query costs; we justify this choice at the end of the section. For trees, this asymmetry disappears.

The following lemma states some useful facts about the structural cost measures of subgraphs of a DAG and the cost measures of a DAG scheme.

Lemma 3.2.1. *Let $\Gamma = (G, S, n, k)$ be any DAG scheme with update cost $\mathcal{U}(\Gamma)$, query cost $\mathcal{Q}(\Gamma)$ and sibling cost $\mathcal{S}(\Gamma)$, H be any weakly connected subgraph of G, $\bar{H}$ be the complete subgraph that corresponds to H and p be any directed path. We have:*

1. *$comb(H) = \sum_{v \in H}(1 + indeg(v))$ and $bnd(p) = 1 + indeg(p) - size(p)$;*

2. *$bnd(\bar{H}) \leq bnd(H)$ and $bnd(\bar{H}) \leq indeg(H) - size(H) + k$;*

3. *$comb(H) > indeg(H) \geq size(H) - 1$ and $indeg(H) \geq bnd(H)$;*

4. *$\mathcal{U}(\Gamma) \geq \mathcal{Q}(\Gamma) > \mathcal{S}(\Gamma)$.*

Proof. (1) Both expressions are derived by inspecting the definition of degree and boundary sizes.

(2) Clearly $bnd(\bar{H}) \leq bnd(H)$, because the missing edges of H contribute to its boundary size. Let now $pred(H, v)$ denote the number of predecessors of node v in G that are not nodes of H and $\overline{pred}(H, v)$ denote the number of predecessors of v in H (i.e., the in-degree of v in H). Since $\bar{H}$ is a complete subgraph and $V_{\bar{H}} = V_H$, we have that $bnd(\bar{H}) =$

$\sum_{v \in V_H} pred(H, v) = \sum_{v \in V_H} [indeg(v) - \overline{pred}(H, v)] = indeg(\bar{H}) - \sum_{v \in V_H} \overline{pred}(H, v)]$. But $indeg(\bar{H}) = indeg(H)$ and, if V^H_{Sink} denotes the set of sink nodes in subgraph H with respect to their out-degree in H, then $\sum_{v \in V_H} \geq size(\bar{H}) + V^{\bar{H}}_{Sink}$, since each non-sink node in $\bar{H}$ contributes one in the sum: the edge that connects it to a successor node of $\bar{H}$. Since $size(\bar{H}) = size(H)$ and $V^{\bar{H}}_{Sink} = V^H_{Sink} \leq k$, we finally have that $bnd(\bar{H}) \leq indeg(H) - size(H) + k$.

(3) By definition and since $size(H) > 0$ we have that $comb(H) > indeg(H)$. Also, consider any spanning tree of the weakly connected subgraph H; all of its $size(H) - 1$ edges belong in H when they are appropriately given a direction. Thus, $indeg(H) \geq size(H) - 1$. (This implies that $comb(H) \geq size(H)$, although by definition for weakly connected graphs we take that $comb(H) > size(H)$.) Also note that $indeg(H) = bnd(H) + |E_H|$ for any subgraph H in G and $|E_H| \geq 0$.

(4) Note that the associated path π_s of any source node of G is a subgraph of the subgraph G_s of G that is reachable from s. Thus, $comb(G_s) \geq comb(\pi_s)$ for any source node s of G and accordingly, $\mathcal{U}(\Gamma) \geq \mathcal{Q}(\Gamma)$. Similarly, since for any subgraph H of G we have from (3) that $comb(H) > bnd(H)$, we take that for any $s \in V_{source}$ it holds that $comb(\pi_s) > bnd(\pi_s)$. Thus, if s^* is the source node in G with the associated path π_s of maximum boundary size, we have that $\mathcal{Q}(\Gamma) \geq comb(\pi_{s^*}) > bnd(\pi_{s^*}) = \mathcal{S}(\Gamma)$. □

Note that, by the above lemma, for any DAG scheme, the update cost is no less than the query cost and the query cost is no less than the sibling cost. This fact will be used for the lower bound derivation: it suffices to focus only on the smallest of the costs of a DAG scheme, i.e., its sibling cost. Note that from Lemma 3.2.1 and in the *worst case*, the combined, degree and boundary sizes of the associated paths of source nodes of DAG scheme $\Gamma = (G, S, n, k)$, but also of the subgraphs reachable from the source nodes of G, are each lower bounded by cost measure $\mathcal{S}(\Gamma)$ of Γ.

3.2.3 Hierarchical Data Processing Problems

Our motivation for introducing DAG schemes is that they model an abstract class of computational problem where a DAG G holds a collection of n input *elements* (stored at source nodes) and a collection of output *values* of size that is bounded by a polynomial on n (stored at non-source nodes) that are computed using the DAG. *Query operations* on elements return a collection of values. *Update operations* modify the DAG G and the input elements, causing corresponding changes to the set of values. Computations are performed sequentially and hierarchically, according to the hierarchy induced by the underlying DAG G. The

computational cost (time, space, or communication complexity) of query and update operations can be expressed as the combined, degree or boundary size of a subgraph (usually G_s or π_s, for a source node s of G), where every node v in the subgraph contributes to the cost an amount proportional to $indeg(v)$. Generally, any computational cost measure for a problem in this class is completely characterized by structural cost measures of subgraphs of DAG G. We refer to such problems as *hierarchical data processing (HDP) problems*. More formally:

Definition 3.2.4 (Hierarchical Data Processing problems)**.** *The class of* hierarchical data processing *problems contain computational problems* Π *with the following characteristics.*

1. *The* input *of* Π *is a set of* elements $\mathcal{E} = \{el_1, ..., el_n\}$ *of size* n*.*
2. *The* output *of* Π *is a collection of* values $\mathcal{V} = \{val_1, ..., val_t\}$ *where* t *is bounded by a polynomial on* n*.*
3. *Associated with* Π *is a DAG scheme* (G, S, n, k)*, such that elements in* $\mathcal{E}$ *and values in* $\mathcal{V}$ *are stored at source and respectively non-source nodes of* G*.*
4. *Given a subset* $\mathcal{E}' \subseteq \mathcal{E}$ *of elements, problem* Π *involves a* computation $\mathcal{C}_{\mathcal{E}'}$ *that performs some type of* data processing*. Computations in* Π *are triggered by some* operation*: either an* update *of an element in* $\mathcal{E}$ *or a* query *about a subset* $\mathcal{E}' \subseteq \mathcal{E}$ *of elements.*
5. *Any computation* $\mathcal{C}$ *in* Π *is fully characterized by an* associated *subgraph* H *of* G*. In particular:*
   - *The associated subgraph* H *of computation* $\mathcal{C}_{\mathcal{E}'}$ *depends on the elements in* $\mathcal{E}'$*, graph* G *and the set of special nodes of* G*. In particular,* H *is a weakly connected and complete subgraph of* G *that includes the source nodes hosting elements in* $\mathcal{E}'$ *and* at least one *of the nodes in* S *(special nodes of* G*).*
   - *Computation* $\mathcal{C}$ *is performed* sequentially *at* steps*, where each node* v *of* H *corresponds to a step of* $\mathcal{C}$*, and* hierarchically *according to the hierarchy induced by* H*, where a step corresponding to node* v *can be executed only after all steps corresponding to the predecessor nodes of* v *have been executed.*
   - *The execution of any step of computation* $\mathcal{C}$ *corresponding to node* v *of* H *contributes to its computational cost an amount* proportional *to* $indeg(v)$*. That is, a step at node* v *has time, space or communication complexity cost of* $\Theta(indeg(v))$*.*

- *Computations in* Π *involve one (or more) of the following types of data processing of the collection* $\mathcal{V}' \subseteq \mathcal{V}$ *of values stored at the nodes of the associated subgraph* H*, where data processing is performed sequentially and hierarchically according to DAG* H*:* (i) *the update of values in* $\mathcal{V}'$*,* (ii) *the output of values in* $\mathcal{V}'$ *or of values in a subset* $\mathcal{V}'' \subset \mathcal{V}'$ *of them or* (iii) *the evaluation of a function on values in* $\mathcal{V}'$ *or evaluations of functions on the values in* $\mathcal{V}'$ *or on the values in* $\mathcal{V}'' \subset \mathcal{V}'$*.*

We note that the DAG scheme that is associated with an HDP problem is *not* part of the input of the problem. That is, it is not given in advance. The DAG scheme is part of the algorithmic and data structuring technique that is used for the solution of the problem. Naturally, we ask which particular DAG scheme or DAG schemes of which type achieve optimal performance in terms of the complexity of the computations performed. By definition, any computation in an HDP problem is performed sequentially and hierarchically according to an associated subgraph of DAG G. Additionally, any such computation incurs computational costs that are fully described by the structural cost measures of the associated subgraph H. Thus, the cost measures of the DAG scheme associated to a HDP problem capture the intrinsic worst case computational complexity of this problem.

In the rest of the section we derive results that reveal the inherent computational limits that exist in any HDP problem and, furthermore, that characterize the optimal DAG scheme structure for these problems.

3.2.4 Sibling Cost and Search by Comparisons

We first show that the cost measures for a tree-based DAG scheme are related to the number of comparisons in a *search tree* derived from the scheme. By the above discussion and Lemma 3.2.1, focusing on the sibling cost suffices. For completeness, we first give some definitions.

Definition 3.2.5 (Directed Trees)**.** *A* directed tree *is a DAG resulting from a rooted tree when its edges are assigned directions towards the root of the tree. Then, parent-child relation is defined among neighboring nodes, leaves correspond to source nodes and internal nodes correspond to non-source nodes.*

Definition 3.2.6 (Search Trees)**.** *Let* $(X, \preceq)$ *be a totally ordered set of size* n*, drawing values from universe* $\mathscr{U}$*, where* $\preceq$ *is a binary relation, referred in this chapter as* greater or equal.

- *Given any element $y \in \mathscr{U}$, we say that we* locate *y in X if we find the* predecessor *(element) $pr(y)$ of y in X, if it exists, defined to be $pr(y) = \{\max_{\preceq} x | x \in X \wedge x \preceq y\}$, i.e., the maximum (with respect to relation $\preceq$) element $x \in X$ such that $x \preceq y$. Locating an existing element of X in X corresponds to* finding *the element itself.*

- *A* leaf-based search tree *for $(X, \preceq)$ is a rooted tree such that: (i) the tree has exactly $|X| = n$ leaves, each storing an element in X, and the internal tree nodes are assigned with $n-1$ elements from X, (ii) given any element $y \in \mathscr{U}$, $pr(y) \in X$ can be located by searching in the tree, where at each tree node the search is lead by* comparisons*, i.e., evaluations of relation $\preceq$ on pairs of elements consisting of y and an element that is assigned at this tree node.*

Lemma 3.2.2. *Let $(X, \preceq)$ be a totally ordered set with n elements drawn from universe $\mathscr{U}$ and let $\Delta = (T, S, n, 1)$ be a DAG scheme, where T is a directed tree. We can build from T a search tree T' for X by storing the elements of X at the leaves of T and assigning tuples of elements in X to internal tree nodes of T, such that using T' any element $y \in \mathscr{U}$ can be located with $bnd(\pi_s)$ comparisons, with s being the source node of T where the search ends. Accordingly, element $x \in X$ stored at source node s of T can be found in X using T' with $bnd(\pi_s)$ comparisons $\preceq$.*

Proof. Assume that tree T is non-trivial. For each internal node of tree T, we fix a *left-right* ordering of its children. Using this ordering, we consider the topological order of T that corresponds to a postorder traversal of tree T and traverse the nodes of T according to this topological order. That is, any node in T is visited after all of its children nodes have been visited and according to the left-right ordering (with respect to the visit of its siblings in T). As we encounter leaves of T we store at them elements of X, one element at each leaf, selecting elements from X in increasing order. Next, we perform the following element assignment for each internal node in T, using again the topological order. Each non-source node v with predecessor nodes $u_1, \ldots, u_\ell$, listed according to the corresponding left-right node ordering, is assigned the ordered $(\ell-1)$-tuple of elements $x_1^v, \ldots, x_{\ell-1}^v \in X$, where for $1 \leq i \leq \ell - 1$, x_i^v is the maximum element with respect to relation $\preceq$ that has been stored at the source nodes of the subtree in T having as root node u_i.

Tree T along with the elements stored at leaves and the assigned elements at internal tree nodes is a leaf-based search tree T' for set X. First, it is easy to see, using induction, that the number of assigned elements to internal nodes of T is $n-1$ (only the maximum according to $\preceq$ relation element in X is not assigned to any node). Suppose we search for element y in the universe $\mathscr{U}$ where elements of X are drawn from. Observe that, while being

at non-source node v, elements $x_1^v, \ldots, x_{\ell-1}^v$ can be used to decide to which node, among nodes $u_1, \ldots, u_\ell$, to advance our search as follows. For relation $\succ$ being the complement of relation $\preceq$, we advance the search at node u_1 if $y \preceq x_1^v$, at node u_ℓ if $y \succ x_{\ell-1}^v$, or otherwise at node u_i, where i is the unique integer value $1 \leq i \leq \ell - 2$ such that $y \succ x_i^v$ and $y \preceq x_{i+1}^v$. While visiting a source node s storing element x_s, we simply report as the predecessor of y in X element x_s. Besides, with respect to the correctness of the above searching procedure, we see that because of the way elements in X are stored at leaves and assigned to internal nodes of T, T' satisfies the following desired (search tree) property: at any node, elements stored in subtrees of children nodes that are to the right in the left-to-right ordering are larger (with respect to relation $\preceq$ on elements in X) than elements stored in subtrees of children nodes that are to the left in this left-to-right ordering. Moving to new node u_{i+1}, we correctly reduce the search space to the ordered subset of X $\{suc(x_i^v), \ldots, x_{i+1}^v\}$, where $suc(x)$ denotes the *successor* of x in X (defined to be the unique element x' in X such that $x = pr(x')$). Note that if a search to locate y ends at a source node s storing element x_s, x_s is the predecessor of y, and if $y \succ x_s$, then y and x_s are the same elements. That is, with one additional comparison at a leaf node we can test equality (whether the predecessor of the located element is the element itself).

Consider the search path p_s, when searching for an element $y \in \mathscr{U}$, that ends at source node s in T', that is the path connecting the root of T' to node s. Since for DAG scheme $\Delta = (T, S, n, 1)$, k (the number of special nodes) equals one, the root of T is the special node and thus p_s is the unique associated path π_s of s, i.e., $p_s = \pi_s$. Let v be a non-source node of this path. We have that by performing $indeg(v) - 1$ comparisons at node v we can advance our search to the correct node among the children of v. When we enter a source node, not comparison is needed. Thus, the total number of comparisons $\preceq$ performed when searching to locate y in X is equal to $\sum_{v \in \pi_s | indeg(v) > 0} (indeg(v) - 1) = 1 + \sum_{v \in \pi_s} (indeg(v) - 1) = 1 + indeg(\pi_s) - size(\pi_s) = bnd(\pi_s)$. Accordingly, since locating an existing element in X corresponds to finding the element itself, we can find element $x_s \in X$ stored at source node s of T using search tree T' by performing $bnd(\pi_s)$ comparisons.

If T contains trivial paths, the proof is as above with only one difference with respect to any nodes with in-degree 1. At every node w with in-degree 1, we assign the element assigned to its child and when searching in the tree we perform no comparison $\preceq$ at w, but rather immediately advance our search at the child node. (That is, at a node v assigned with only one search key, we perform a comparison only when $indeg(v) = 2$.) The number of comparisons is again $bnd(\pi_s)$. $\square$

Lemma 3.2.2 draws a direct analogy between the sibling cost of any tree-based DAG scheme and the number of comparisons performed in a search tree corresponding to the DAG scheme. We use this analogy as the basis for a reduction from searching by comparisons to any computational procedure of a HDP problem with cost that is expressed by the sibling cost of a tree-based DAG scheme.

Theorem 3.2.1. *Any DAG scheme $\Delta = (T, S, n, 1)$ such that T is a directed tree has $\Omega(\log n)$ update, query and sibling costs.*

Proof. It follows from Lemma 3.2.2 and the well-known $\Omega(\log n)$ lower bound on searching for an element in an ordered sequence in the comparison model (see for instance [71]). Namely, this fundamental result states that *any algorithm* for finding an element x in a list of n entries, by *comparing* x to list entries, *must perform* at least $\lfloor \log n \rfloor + 1$ comparisons for some input x. Comparisons are simply evaluations of binary relations on pair of elements. Obviously the above statement applies also to any search tree built for a totally ordered set X of size n using binary relation $\preceq$ on the elements of X. To see why, observe that a search tree can be viewed as an index structure for searching an ordered list of n elements, represented as the leaves of the search tree according to postorder tree traversal; additionally, any search path to an element of X in the tree completely describes the sequence of comparisons performed in order to locate this element. Consequently, applying the above to the search tree T' of Lemma 3.2.2, we get that for any DAG scheme $(T, S, n, 1)$, T being a directed tree, there exists a source node s such that the boundary size $bnd(\pi_s)$ of the associated path π_s of s is at least $\lfloor \log n \rfloor$, thus there exists a source node s such that $bnd(\pi_s)$ is $\Omega(\log n)$. Then by definition, we get that for any DAG scheme $\Delta = (T, S, n, 1)$ it holds that $\mathcal{S}(\Delta)$ is $\Omega(\log n)$. We finish the proof, by noting that from Lemma 3.2.1, for any DAG scheme $\Delta = (T, S, n, 1)$ we have that $\mathcal{S}(\Delta) < \mathcal{Q}(\Delta) \leq \mathcal{U}(\Delta)$ and that, since T is a directed tree its query cost $\mathcal{Q}(\Delta)$ is equal to its update cost $\mathcal{U}(\Delta)$. Thus, $\mathcal{Q}(\Delta) = \mathcal{U}(\Delta)$ and both are $\Omega(\log n)$. □

Remark 3.2.1. *In any DAG scheme $(T, S, n, 1)$, where T a directed tree, there is a unique path connecting source node s to the unique special node in S (the root of T). Thus, the associated path π_s is also the minimum boundary size path from s to the root.*

3.2.5 Optimality of Tree Structures

Next, we show that trees have optimal cost measures among all possible DAG schemes. We start by showing that for DAG schemes $(G, S, n, 1)$ having one special node, optimal costs

are achieved when G is a directed tree.

Theorem 3.2.2. *Let $\Gamma = (G, S, n, 1)$ be a DAG scheme. There exists a DAG scheme $\Delta = (T, S, n, 1)$ such that T is a directed tree and $\mathcal{U}(\Delta) \leq \mathcal{U}(\Gamma)$, $\mathcal{Q}(\Delta) \leq \mathcal{Q}(\Gamma)$, and $\mathcal{S}(\Delta) \leq \mathcal{S}(\Gamma)$.*

Proof. DAG scheme Γ has only one special node, the unique sink node of G, so associated paths of the source nodes of G are paths from a source node to the sink node of G. we fix a topological order $t(G)$ of G. We define DAG T to be the union of all minimum combined cost directed paths π_s for all source nodes in G, where ties in computing paths π_s are broken using a consistent rule according to the topological order $t(G)$. It is easy to see that the union is a directed tree: if two paths π_{s_1} and π_{s_2} from source nodes s_1 and s_2 cross at node v and meet again at node u (u may be the sink node of G), this contradicts either the fact that each path has minimum combined cost or the tie breaking rule. For instance, if subpath $p_{v,u}(s_2)$ of π_{s_2} does not coincide subpath $p_{v,u}(s_1)$ of π_{s_2}, either $comb(p_{v,u}(s_1)) \neq comb(p_{v,u}(s_2))$, in which case, one of the two paths π_{s_1} and π_{s_2} is not optimal (it is not of minimum combined cost), or $comb(p_{v,u}(s_1)) = comb(p_{v,u}(s_2))$, in which case, the tie breaking rule was violated.

By definition, it holds that $\mathcal{U}(\Delta) \leq \mathcal{U}(\Gamma)$, since for any source node s the reachable from s subgraph T_s in T (which is simply the corresponding leaf-to-root path in T) is a subgraph of the reachable from s subgraph G_s in G. With respect to query and sibling cost, it is easy to see that, since for any source node s the associated path π_s stays the same in graphs G and T but $|E_G| \geq |E_T|$, it holds that $indeg_T(\pi_s) \leq indeg_G(\pi_s)$. Thus, $\mathcal{Q}(\Delta) \leq \mathcal{Q}(\Gamma)$, and $\mathcal{S}(\Delta) \leq \mathcal{S}(\Gamma)$, as stated. □

Remark 3.2.2. *The directed tree T in the proof above, by construction, consists of minimum combined size paths from source nodes in G to the unique special node and thus T is an optimal tree with respect to the structural cost measure of combined size. However T is not necessarily optimal with respect to the structural cost measure of boundary size, because, for general graphs, minimum combined cost does not imply minimum boundary cost. Thus, although tree T is a minimum combined size tree in G and it is a tree such that $\mathcal{S}(\Delta) \leq \mathcal{S}(\Gamma)$, T may not be a minimum boundary size tree in G. In other words, the minimum combined size and minimum boundary size trees in G do not necessarily coincide. This does not affect any of our results.*

Finally, we examine how allowing more than one special nodes in a DAG scheme affects its cost measures. We have that a DAG scheme Γ with k special nodes achieve minimum

cost measures when the roots of at most k *distinct trees* are the only special nodes in Γ.

Lemma 3.2.3. *Let $\Gamma = (G, S_G, n, k)$ be a DAG scheme. There exists a DAG scheme $\Phi = (F, S_F, n, \ell)$ such that F is a forest of $\ell \leq k$ directed trees, $S_F \subseteq S_G$ and additionally $\mathcal{U}(\Phi) \leq \mathcal{U}(\Gamma)$, $\mathcal{Q}(\Phi) \leq \mathcal{Q}(\Gamma)$, and $\mathcal{S}(\Phi) \leq \mathcal{S}(\Gamma)$.*

Proof. The proof is similar to the proof of Theorem 3.2.2. Consider the union F of the minimum combined size paths $\pi_1, \ldots, \pi_n$ in G from source nodes $s_1, \ldots, s_n$ to a special node, where ties break according to a well-defined and consistent rule (e.g., using a topological order of G). The resulting subgraph F of G is a forest: two paths never cross, but they only meet to same special node, and additionally, no path connecting two distinct special nodes exists in F. As in the proof of Theorem 3.2.2, since (i) for any source node s the reachable from s subgraph F_s in F is a subgraph of the reachable from s subgraph G_s in G, (ii) $|E_F| \leq |E_G|$, and (iii) the minimum combined size paths (from source nodes to special nodes) are the same in G and F, we have that $\mathcal{U}(\Phi) \leq \mathcal{U}(\Gamma)$, $\mathcal{Q}(\Phi) \leq \mathcal{Q}(\Gamma)$, and $\mathcal{S}(\Phi) \leq \mathcal{S}(\Gamma)$, as desired. □

3.2.6 Lower Bounds for Hierarchical Data Processing

The following theorem summarizes the results of this section with respect to the cost measures of any DAG scheme.

Theorem 3.2.3. *Any DAG scheme $\Gamma = (G, S, n, k)$ has $\Omega(\log \frac{n}{k})$ update, query and sibling costs.*

Proof. It follows directly from Theorems 3.2.1 and 3.2.2 and Lemma 3.2.3. First, by Lemma 3.2.3 and Theorem 3.2.2 we get that the best (lowest) cost measures of DAG scheme Γ are achieved when G is a forest F of at most k trees. In this case, since these trees are minimum combined cost trees, the update, query and sibling cost of G are defined by the tree $T^* \in F$ having the maximum complexity in terms of the cost measure of sibling cost. Moreover, this cost measure depends of the number of source nodes $V_{source}(T^*)$ of T^*. We know that $|V_{source}(T^*)|$ is $\Omega(\frac{n}{k})$. If $\Phi = (F, S_F, n, \ell)$, $\ell \leq k$, is the DAG scheme of Lemma 3.2.3 and $\Delta = (T^*, S, |V_{source}(T^*)|, 1)$ is the DAG scheme that corresponds to tree T^*, we have that $\mathcal{S}(\Gamma) \geq \mathcal{S}(\Phi) = \mathcal{S}(\Delta)$, which from Theorem 3.2.1 is $\Omega(\log \frac{n}{k})$. By Lemma 3.2.1 we get also that $\mathcal{U}(\Gamma)$ is $\Omega(\log \frac{n}{k})$ and that $Q(\Gamma)$ is $\Omega(\log \frac{n}{k})$. □

The above results form the basis for a reduction from searching by comparisons to any computation of an HDP problem. Essentially, we draw an analogy between searching by

comparisons and the sibling cost of any DAG scheme. The above theorem can be directly combined with the definition of the class of HDP problems to give the following general result about the complexity of any problem in the class.

Theorem 3.2.4. *All computations of any hierarchical data processing problem* Π *associated with DAG scheme* $\Gamma = (G, S, n, k)$ *have* $\Omega(\log \frac{n}{k})$ worst case *time, space or communication complexity.*

Proof. By definition, an HDP problem Π involves computations that correspond to associated subgraphs of G. Recall that a computation $\mathcal{C}$ is associated to a subgraph H of G and involves some type of data processing and is performed sequentially and hierarchically according to graph H, such that: (i) computations are performed in steps, one step per node of H and steps are executed one after the other, (ii) computational step corresponding to node u of H can be executed only when the computational steps corresponding to all predecessor nodes of u in H has been executed and (iii) the computational step corresponding to node u of H has time, space or communication complexity that is proportional to $indeg(u)$, that is, the complexity is $\Theta(indeg(u))$. Also recall that computation (and graph H) depends on a subset $\mathcal{E}'$ of the input elements $\mathcal{E}$. Namely, graph H is the complete subgraph of G that contains the source nodes storing elements in $\mathcal{E}'$ and at least one special node.

Let $\mathcal{C}_{\mathcal{E}'}$ be any computation of problem Π that depends on subset $\mathcal{E}'$, with H being its associated subgraph of G. Given the above setting with respect to the computational model for HDP problems, it follows that computation $\mathcal{C}$ has time, space or communication complexity C_H proportional to $\sum_{v \in V_H} \Theta(indeg(v))$, thus $C_H = \Theta(\sum_{v \in V_H} indeg(v)) = \Theta(indeg(H)))$. Accordingly $C_H = \Omega(bnd(H))$, since $indeg(H) \geq bnd(H)$ for any H (Lemma 3.2.1). We have concluded that any computation $\mathcal{C}_{\mathcal{E}'}$ with associated graph H has time, space or communication complexity $C_H = \Omega(bnd(H))$. To complete the proof, simply consider $\mathcal{E}' = x_{s^*}$, where x_{s^*} is the input element stored at the source node s^* that defines the sibling cost $\mathcal{S}(\Gamma)$ of DAG scheme Γ, i.e., the source node for which the minimum combined cost path to a special node u_{sp} is the maximum over all other source nodes, and let H contain only this special node $u_{sp} \in S$. The associated path π_{s^*} of s^* is then a subgraph of H. Thus, $bnd(H) \geq bnd(\pi_{s^*}) = \mathcal{S}(\Gamma)$ and from Theorem 3.2.3 $\mathcal{S}(\Gamma)$ is $\Omega(\log \frac{n}{k})$, so $C_H = \Omega(\log \frac{n}{k})$. It follows that for any HDP problem Π associated with DAG scheme Γ, there exists a computation with $\Omega(\log \frac{n}{k})$ time, space or communication complexity. □

Remark 3.2.3. *Regarding the definition of the sibling cost of a DAG scheme* Γ *with respect to the minimum combined size associated paths, we note that this choice is twofold. First,*

we have seen that $\mathcal{S}(\Gamma)$ serves our purposes, since it strictly *bounds the query and update costs of Γ and provides us with worst case lower bounds for the computational costs of the update and query operations in any HDP problem. Second, in the case where the cost of a computation in an HDP problem is characterized by the boundary size of some source-to-special node path, this path typically corresponds to a minimum combined cost path. We will see such a case in Section 3.4.*

We have established logarithmic lower bounds for the complexity of computations related to HDP problems, where we have further characterized the tree-based DAG schemes as the optimal structures for these problems. The connection between problems in this class and DAG schemes is illustrated in Sections 3.4 and 3.5, where we model two information security problems as HDP problems and translate the above results to their domain.

3.3 A New DAG Scheme Based on Skip-Lists

In view of the logarithmic lower bounds and the optimality of tree structures for HDP problems, in this section, we describe a new tree-based DAG scheme $\Delta = (T, S, n, 1)$, which we call *multi-way skip-list DAG scheme*, that is based on and defined with respect to skip-lists [120, 121], which are randomized data structures, equivalent to balanced trees, upon which dictionaries can be built. We study the performance of Δ in terms of the node size $size(\cdot)$, the degree size $indeg(\cdot)$ and also the boundary size $bnd(\cdot)$, where we show that all these cost measures have low expected values.

3.3.1 Skip-Lists and Bridges

We briefly describe the notation that we will use. A *skip-list* with *probability parameter* p is a set of lists $L_1, ..., L_h$, where L_1 stores all the element of a totally ordered set $(X, \preceq)$ of size n, sorted according to $\preceq$, where elements are drawn from universe $\mathscr{U}$, and, for each i, each of the elements of list L_i is independently chosen to be contained in L_{i+1} with probability p. Lists are viewed as *levels* and we consider all elements of the same value that are stored in different levels to form a *tower*. That is, a tower consists of nodes of lists that store the same copied element. The *level* of a tower is the highest level of the tower (or the level of its top element). Each node of a tower has a forward pointer to the successor element in the corresponding list and pointer to the element one level below it. A *header* tower that stores sentinel element $-\infty$, representing the minimum with respect to $\preceq$ value in $\mathscr{U}$, is included in the skip-list as the left-most tower of level one more than the maximum level

in the skip-list. A node of the skip-list is a *plateau* node if it is the top node of its tower. Furthermore, we introduce the notion of a *bridge* and also define relative concepts.

Definition 3.3.1 (Skip-List Notation)**.** *In a skip-list:*

- *a* bridge *b is a sequence of towers of the same level, where no higher tower is interfering them and the plateau nodes of the towers are all reachable in a sequence using forward pointers;*
- *the* bridge size *$|b|$ of bridge b is the number of towers in the bridge (i.e., the size of the sequence); the* bridge size *of a tower is the size of the bridge that the tower belongs to;*
- *a* child bridge *of b is a bridge that is contained under b and to which a tower of b is connected through forward pointers;*
- *the* plateau towers *of a tower t are the towers whose plateau nodes can be reached by t using one forward pointer.*

3.3.2 Construction of New Directed Tree

We now describe the new DAG scheme $(T, S, n, 1)$, the skip-list DAG, where T is a directed tree with n leaves (source nodes) and one special node, the root r of T, thus $S = r$. For simplicity, we use the term skip-list DAG to refer to both the DAG scheme $(T, r, n, 1)$ and the directed tree T. The skip-list DAG is defined with respect to a skip-list with probability parameter p. In what follows, by *list node* we refer to a node of the skip-list and by *DAG node* to a node of skip-list DAG T. Any edge (v, u) in T is assumed to be directed from node v to node u. To facilitate our description we define an operation on DAG nodes, which, given existing DAG nodes in T, creates new nodes and edges in T: if $v, v_1, \ldots, v_l$ are nodes in T, then operation $New(v, v_1, \ldots, v_l)$ creates in T new nodes $u_1, \ldots, u_l$ and new edges $(v_1, u_2), \ldots, (v_{l-1}, u_l), (v_l, v)$ and $(u_1, u_2), (u_2, u_3), \ldots, (u_{l-1}, u_l), (u_l, v)$, where DAG node u_1 is a *source* node in T. So, operation $New(\cdot)$, in fact, creates and appropriately connects it in T a directed path from source node u_1 to the existing in T node v, where existing nodes in T $v_1, \ldots, v_l$ are sibling nodes in the path. Finally, to better understand the connection between the skip-list DAG and the skip-list and also for our analysis, we consider that each DAG node is *attached* to some list node in the skip-list.

The notion of a bridge is essential in skip-list DAG. For each bridge b in the skip-list, a corresponding node $v(b)$ is created in T. We call $v(b)$ the DAG node of b. In essence,

$v(b)$ is connected in T with the DAG nodes of all the child bridges of b. This allows us to define DAG T with respect to a skip-list in a recursive way: first all bridges in the skip-list are identified and the DAG node for the outer bridge, corresponding to the header tower of the skip list, is created, then, given that the DAG node $v(b)$ of a bridge b is created, using operation $New(\cdot)$, it is connected with paths in T to the newly created DAG nodes of the child bridges of b (see Figure 3.1). We next explain how this connection is performed.

First, suppose that the size of b is one, i.e., b is simply a tower t (see Figure 3.1(a)). Let $t_1, ..., t_l$ be the plateau towers of t in increasing order with respect to their level. Note that the level of t_l is less than the level of t. If plateau tower t_i belongs in bridge b_i, then let $v(b_1), ..., v(b_l)$ be the corresponding DAG nodes of the bridges. Then we perform operation $New(v(b), v(b_1), \ldots, v(b_l))$ where $v(b)$ is the DAG node of b. We attach the new nodes of T created from this operation as follows: source node u_1 is attached to the lowest list node of the tower, node u_i, $2 \leq i \leq l$, is attached to the list node of t at the level of bridge b_{i-1} and the DAG node $v(b)$ of b is attached to the list node of t at the level of bridge b_l. Note that DAG node u_1 is the basis in the recursion. If the size of b is more than one, say k, then, let $t_1, \ldots, t_k$ be the towers of b (see Figure 3.1(b)). First, for each such tower t_i, we create a new DAG node $v(t_i)$, $1 \leq i \leq k$. For tower t_k, we consider its, say l, plateau towers $t_{k1}, \ldots, t_{kl}$ and perform operation $New(v(t_k), v(b_{k1}), \ldots, v(b_{kl}))$, where $b_{k1}, \ldots, b_{kl}$ are the child bridges of b that plateau towers $t_{i1}, ..., t_{il}$ belong in and where the new source node u_1 created by this operation corresponds to the element that is stored in tower t_k. Moreover, for each tower t_i, $i < k$, of, say $l + 1$, plateau towers, we consider its l lowest plateau towers $t_{i1}, \ldots, t_{il}$, that is, for $i < k$, tower t_{i+1} is *omitted* from this sequence (only in this case, t_{i+1} is not considered to be a plateau tower of t_i). Let $b_{i1}, \ldots, b_{il}$ be the child bridges b that plateau towers $t_{i1}, ..., t_{il}$ belong in. Then for tower t_i, $i < k$, we perform operation $New(v(t_i), v(b_{i1}), \ldots, v(b_{il}))$, where the new source node u_1 created by this operation corresponds to the element that is stored in tower t_i. Finally, we add the following k new edges in T: edge $(v(t_i), v(b))$ for $1 \leq i \leq k$. Newly created nodes are attached to list nodes in a similar way as before. We attach the DAG nodes created when considering tower t_i, $1 \leq i \leq k$, to the list nodes of the tower at the level of the bridge they are connected to, source nodes to the corresponding lowest-level list nodes and DAG node $v(b)$ of bridge b to the top left-most list node of b.

By this recursive definition, it is easy to see that, indeed, T is a directed tree; the root and unique sink node of T corresponds to the first created DAG node r for the highest bridge of the skip-list, the one consisting of the left-most header tower. The leaves and source nodes of T correspond to the lowest-level list where n elements of set X are stored in

the skip-list. Thus, skip-list DAG is a $(T, r, n, 1)$ DAG scheme (see also Figure 3.2). Note that since our new DAG scheme is defined with respect to a skip-list, it is a *randomized* DAG scheme.

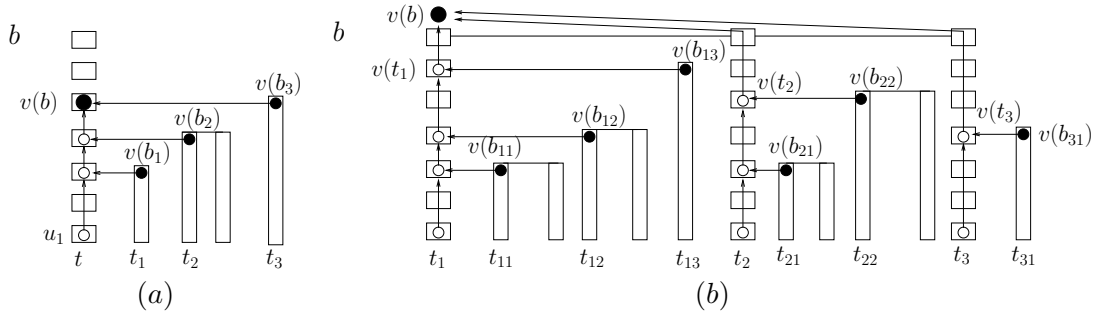

Figure 3.1: Multi-way skip-list scheme Δ, where circle nodes are DAG nodes (bridge DAG nodes are solid) and square nodes are nodes of the skip-list. DAG node $v(b)$ of bridge b is recursively connected to the DAG nodes of the child bridges depending on the bridge size: (a) $|b| = 1$ and (b) $|b| > 1$. A DAG node is attached to the list node in which it is contained.

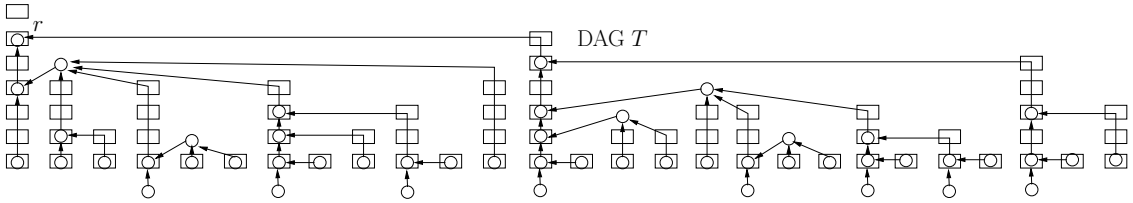

Figure 3.2: A multi-way skip-list DAG scheme $\Delta = (T, S, n, 1)$. T is a directed tree.

3.3.3 Cost Measures of Skip-List DAG

We now analyze the cost measures of the DAG scheme $\Delta = (T, r, n, 1)$ according to the definitions in Section 3.2.2. From Theorem 3.2.2 and Lemma 3.2.3, it suffices to study only the case where there is only one special node in T, the root of T. Also, we know that for the update , query and sibling cost measures of Δ it holds $\mathcal{U}(\Delta) = Q(\Delta) > \mathcal{S}(\Delta)$ and that all are lower bounded $\log n$. Accordingly, it suffices to concentrate our analysis to the associated path π_s of a source node s in T. This path is not only the minimum combined size path from s to the root but also the minimum boundary size path of s. Since Δ is a randomized DAG scheme, we study its *expected* cost measures, and since these measures can be expressed using the structural metrics node size $size(\cdot)$, degree size $indeg(\cdot)$ and boundary size $bnd(\cdot)$ (see Lemma 3.2.1), we are interested in studying the *expected node*

size $E(size(\pi_s))$, the *expected degree size* $E(indeg(\pi_s))$ of path π_s and the *expected boundary size* $E(bnd(\pi_s))$ of path π_s, i.e., the expected number of nodes in the path, the expected total number of the predecessor nodes of nodes in the path and the expected total number of sibling nodes in the path π_s.

In the next Theorem we prove that our new DAG scheme achieves cost measures that are close to the theoretical optimal value of $\lfloor \log n \rfloor + 1$. Note that this value is theoretically optimal also for randomized DAG schemes, since Lemma 3.2.2 holds true for any DAG scheme, even randomized, because only its structural properties matter for the definition of the corresponding search tree.

Theorem 3.3.1. *With respect to a skip-list with probability parameter p, the skip-list DAG scheme $\Delta = (T, r, n, 1)$, for any fixed source node s of T and the corresponding source to root path π_s, has the following expected performance:*

1. $E[size(\pi_s)] \leq 2(1-p)\log_{\frac{1}{p}} n + O(1)$;
2. $E[indeg(\pi_s)] \leq (1-p)\frac{(1+p)^2}{p}\log_{\frac{1}{p}} n + O(1)$;
3. $E[bnd(\pi_s)] \leq \frac{(1-p)(1+p^2)}{p}\log_{\frac{1}{p}} n + O(1)$ *and*
4. $E[size(T)] \leq (1 + pq^2 + pq + \frac{p}{q(2-pq^2)})n$, *where* $q = 1-p$.

Proof. In the proof, we use a worst case scenario, assuming that the skip-list has *infinite* size to the left or to the right, that is, it is unbounded to one direction. This allows us to actually compute upper bounds of expected values of random variables related to our analysis.

We first compute the expected size of a bridge b in the skip-list. Consider the left-most tower t of b and its top skip-list node u. Assuming an infinite to the right skip-list, u has a forward pointer to the skip-list node v, the next node in list of u. With probability $1-p$ node v is the top node of its tower t_v, thus t_v belongs in b and with probability p tower t_v is of higher level, thus t_v is not a tower of b. If $|b|$ denotes the size of bridge b, then $|b| = 1 + Y$, where Y is a geometrically distributed random variable with parameter p, which counts the number of failures before the success occurs, where $\Pr[\text{success}] = p$. Thus, $E[|b|] = 1 + \frac{1-p}{p} = \frac{1}{p}$ and on average $1/p$ towers are consecutive having the same height.

(1), (2) & (3) We use a backward analysis as in [120, 121]. Given a skip-list, we consider the corresponding skip-list data structure storing a totally ordered set X of size n, where elements in X can be found by traversing the list nodes of the skip-list. We consider traveling backwards on the search path π in the skip-list data structure for element x

stored at source node s. It is easy to see, that, given the way with which DAG nodes of T are assigned to list nodes of the skip-list, path π_s in T, the leaf-to-root path in T from source node s of T, is *contained* in path π. Consequently, as we travel backwards (starting from s) along the search path π, we compute the structural metrics $size(\pi_s)$, $indeg(\pi_s)$ and $bnd(\pi_s)$ of path π_s in T. Assuming a worst case analysis, where π reaches level $\log_{\frac{1}{p}} n$, we split π in two parts: subpath π_1 and subpath π_2. Path π_1 takes us to level $\log_{\frac{1}{p}} n$ and path π_2 completes the backward search up to the first skip-list node of π. Accordingly, in our analysis we partition π_s into subpaths π_{s_1} and π_{s_2} corresponding to subpaths π_1 and respectively π_2 of π. Obviously, since $\pi_s = \pi_{s_1} \cup \pi_{s_2}$, $size(\pi_s) = size(\pi_{s_1}) + size(\pi_{s_2})$, $indeg(\pi_s) = indeg(\pi_{s_1}) + indeg(\pi_{s_2})$ and $bnd(\pi_s) = bnd(\pi_{s_1}) + bnd(\pi_{s_2})$.

The node, degree and boundary sizes of π_{s_2} are all on average constant, $E[size(\pi_{s_2})] = O(1)$, $E[indeg(\pi_{s_2})] = O(1)$ and $E[bnd(\pi_{s_2})] = O(1)$, because the skip-list size above level $\log_{\frac{1}{p}} n$ is on average (and with high probability) *constant*. Indeed, subpath π_2 can be further partitioned into the set L of nodes that we reach in our backward traversal moving leftward and the set U of nodes that we reach moving upward. We have that $|L| \leq Y$, where Y is a random variable counting the number of towers in the skip list with level $\log_{\frac{1}{p}} n$ or higher and that $Y \sim Bin(n, \frac{1}{np})$, since with probability $p^{(\log_{\frac{1}{p}} n)-1} = \frac{1}{np}$ a tower has level $\log_{\frac{1}{p}} n$ or higher. So, $E[L] \leq \frac{1}{p} = O(1)$. Also, we have that $|U| \leq Z$, where Z is the size of the highest skip-list tower above level $\log_{\frac{1}{p}} n$ and that $Z \sim G(1-p)$, i.e., geometrically distributed with parameter $1-p$ (where we use the memoryless property of geometric distribution). So $E[U] \leq \frac{p}{1-p} = O(1)$. We conclude that the size of π_2 is on average at most $E[L] + E[U] = O(1)$. Since every node in π_{s_2} has constant in-degree (2 or $E[|b|] = O(1)$ for a bridge b), we conclude that the node, degree and boundary sizes of π_{s_2} are all $O(1)$.

(1) For the node size of π_{s_1}, we assume an infinite skip-list to the left, that is, no header tower is present (and thus a worst case analysis is in place). Let $C_k(t)$ be a random variable counting the node size $size(\pi)$ counted so far when k upwards moves remain to be taken in part π_1 of path π and we are performing the t-th step. Then if we move up $C_k^t = X_U + C_{k-1}^{t+1}$, otherwise $C_k^t = X_L + C_k^{t+1}$, where X_U, X_L are 0-1 random variables that count if a DAG-node is encountered when moving up or left respectively. Then we have that $\Pr[X_U = 1] = p(1-p)$, because with probability $1-p$ the node that the forward pointer points to is a plateau node and with probability p the node that we move to is not a plateau node (i.e., we count a DAG-node created by applying operation $New(\cdot)$). Also we have that $\Pr[X_L = 1] = p + p(1-p)$, because with probability p we left a bridge and with probability $(1-p)$ the bridge has size more that one (i.e., we count a node created by

operation $New(\cdot)$ and possibly a bridge DAG node). Observe that we count DAG nodes of bridges of size 1 when moving up. Since we have an infinite skip list, $C_k^t \sim C_k^{t+i} \sim C_k$ for any $i > 0$, where C_k is a random variable distributed as C_k^t. Thus, using conditional expectation

$$\begin{aligned} E[C_k] &= E[E[C_k|\text{move}]] \\ &= E[\Pr[\text{up}]C_k|\text{up} + \Pr[\text{left}]C_k|\text{left}] \\ &= E[p(X_U + C_{k-1}) + (1-p)(X_L + C_k)] \\ &= p(E[X_U] + E[C_{k-1}]) + (1-p)(E[X_L] + E[C_k]) \\ &= p(p(1-p) + E[C_{k-1}]) + (1-p)(p + p(1-p) + E[C_k]), \end{aligned}$$

which gives $E[C_k] = E[C_{k-1}] + 2(1-p)$, and finally we get that $E[C_k] = 2(1-p)k$. So, $E[size(\pi_s)] \leq E[C_k] = 2(1-p)\log_{\frac{1}{p}} n + O(1)$.

(2) Similarly, regarding the degree size of π_{s_1} and, again, assuming an infinite skip-list to the left, let $C_k(t)$ be a random variable counting the degree size $indeg(\pi)$ counted so far when k upwards moves remain to be taken and we are performing the t-th step. Then if we move up $C_k^t = X_U + C_{k-1}^{t+1}$, otherwise $C_k^t = X_L + C_k^{t+1}$. Here X_U and X_L are random variables that count the number of predecessor DAG nodes that we have to add when moving up or left respectively. We have that $E[X_U] = 2p(1-p)$ because with probability $p(1-p)$ we count two predecessors after moving up (a node created by operation $New(\cdot)$ has in-degree two). Also $E[X_L] = p(2 + \frac{1-p^2}{p})$, because with probability p we have just left a bridge moving left and, thus, we count $2 + Y$ predecessors. I.e., a node created by operation $New(\cdot)$ has two predecessors and Y is a random variable counting the in-degree of a possible bridge DAG node that must be encountered. Observe that $Y = 0$ unless the bridge has size at least 2 and we compute $E[Y] = (1-p)(2 + \frac{1-p}{p}) = \frac{1-p^2}{p}$. Using conditional expectation as above, we finally get $E[indeg(\pi_s)] \leq (1-p)(2p + 2 + \frac{1-p^2}{p})\log_{\frac{1}{p}} n + O(1) = (1-p)\frac{(1+p)^2}{p}\log_{\frac{1}{p}} n + O(1)$.

(3) For the boundary size of π_{s_1} and assuming an infinite skip-list to the left, let $C_k(t)$ be a random variable counting the degree size $bnd(\pi)$ counted so far when k upwards moves remain to be taken and we are performing the t-th step. As before, if we move up $C_k^t = X_U + C_{k-1}^{t+1}$, otherwise $C_k^t = X_L + C_k^{t+1}$, where X_U and X_L are random variables that count the number of sibling DAG nodes in π_{s_1} that we have to add when moving up or left respectively. We have that $E[X_U] = p(1-p)$ because with probability $p(1-p)$ we count one sibling DAG node after moving up (a node created by operation $New(\cdot)$ has in-degree two: one node is in p_s and one is a sibling node). Also $E[X_L] = p(1 + \frac{1-p}{p}) = 1$, because with probability p we have just left a bridge moving left and, thus, we count $1 + Y$

sibling nodes, one sibling of the node created by operation $New(\cdot)$ and Y siblings (a random variable) corresponding to the possible bridge DAG node that we just left. We have that $E[Y] = (1-p)(1+\frac{1-p}{p}) = \frac{1-p}{p}$. Using conditional expectation as before, we finally get $E[bnd(\pi_s)] \leq (1-p)(\frac{1+p^2}{p}) \log_{\frac{1}{p}} n + O(1)$.

(4) By construction, $V_T = V_{source}(T) \cup B \cup N$, where $V_{source}(T)$ is the set of the n source nodes in T, B the set of bridge DAG nodes and N the set of the non-source, non-bridge, DAG nodes (created by operation $New(\cdot)$ or being predecessors of a DAG node of a bridge). Let B_1 denote the set of DAG nodes in B that are assigned to list-nodes of level 1 and let $B_{>1} = B - B_1$. Similarly, let N_1 denote the set of DAG nodes in N that are assigned to list-nodes of level 1 and let $N_{>1} = N - N_1$. We can compute that $E[|B_1|] \leq p(1-p)^2 n$ using the union bound, since with probability $p(1-p)^2$ the lowest plateau tower of a tower is the left most tower of a bridge of level 1 and of size at least two. Similarly, we can compute that $E[|N_1|] \leq p(1-p)n$, again by applying the union bound and noticing that with probability $p(1-p)$ a non-source DAG node is created by operation $New(\cdot)$. Now, let M denote the set of list nodes in the skip-list that have not been assigned a DAG node, which we call *empty* nodes, then for the set $K_{>1}$ of list nodes in the skip-list of level 2 or more, we have that $K_{>1} = M \cup B_{>1} \cup N_{>1}$, where in this formula DAG nodes are treated like the list nodes they are assigned to. We next pair up DAG nodes in $B_{>1} \cup N_{>1}$ to *distinct* empty list nodes in M. Nodes in $B_{>1}$ are paired up with probability 1. Consider any node u in $B_{>1}$ corresponding to bridge b, with $|b| \geq 2$. Node u can be paired with any of the empty list nodes of the top level of b. Any DAG node u in $B_{>1}$ corresponding to bridge b, with $|b| = 1$, can be paired with the empty list node one level up in its tower. Any DAG node u in $N_{>1}$ can be paired up with the empty list node $l(u)$ on its left, if it not paired with a node in $B_{>1}$. The probability that u can not be paired up with $l(u)$ is $\lambda = p(1-p)^2$ ($l(u)$ is the top-right node of a bridge of size 2), an *independent* event from any other node in $N_{>1}$ not being paired up with its empty node of the left. Thus, we have that, if $M = M_1 \cup M_2 \cup M_3$, where M_1, M_2, M_3 are disjoint sets containing the empty nodes paired up with nodes in $B_{>1}$, in $N_{>1}$ and respectively with no DAG nodes, then $E[|M_1|] = E[|B_{>1}|]$ and $E[|M_2| = (1-\lambda)E[|N_{>1}|]]$. Using this, we get that $E[|K_{>1}|] = 2E[|B_{>1}|] + (2-\lambda)E[|N_{>1}|] + E[|M_3|]$ and the bound $E[|B_{>1}|] + E[|N_{>1}|] \leq \frac{E[K_{>1}]}{2-\lambda}$. Putting all together, we finally get the upper bound for $E[size(T)] = E[|V_T|]$. Note that $E[size(T)] < \frac{n}{1-p}$, the expected size of a skip-list. □

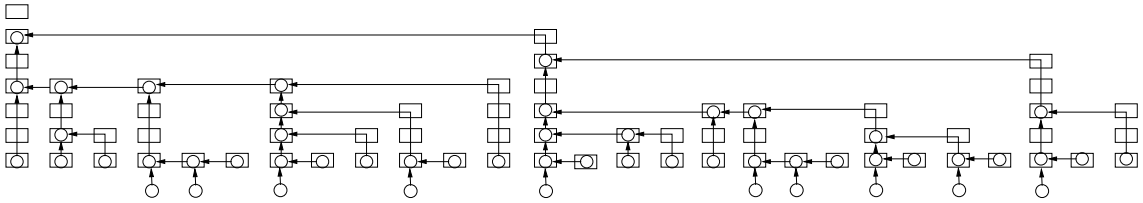

Figure 3.3: The DAG scheme Δ that corresponds to an improved version (with respect to the expected number of comparisons) of the standard skip list [121].

3.3.4 Comparison with other DAG Schemes

We now compare our multi-way skip-list with red-black trees and the standard skip-list in terms of the cost measures of the underlying search DAGs. Figure 3.3 shows that search DAG that corresponds to an improved version of the standard skip-list appeared in [121]. For these trees and for any fixed element stored at a source node s and the corresponding to path π_s in DAG T, we compare the corresponding expected values for the node size $size(\pi_s)$, degree size $indeg(\pi_s)$ and boundary size $bnd(\pi_s)$ of path π_s and the node size $size(T)$ of T. Note that by Lemma 3.2.2, $comb(\pi_s)$ corresponds to the number of comparisons performed in the search structure and that $size(T)$ corresponds to the total number of decision nodes in search tree T.

	$E[size(\pi_s)]$	$E[indeg(\pi_s)]$	$E[bnd(\pi_s)]$	$E[size(T)]$
red-black tree	$\log n$	$2\log n$	$\log n$	$2n$
standard skip-list	$1.5\log n$	$3\log n$	$1.5\log n$	$2n$
multi-way skip-list	$\log n$	$2.25\log n$	$1.25\log n$	$1.9n$

Table 3.1: Comparison of three tree DAG schemes in terms of structural metrics $size(\pi_s)$, $indeg(\pi_s)$, $comb(\pi_s)$ and $size(T)$. Numbers correspond to expected values of the corresponding cost parameters: for red-black trees, expectation corresponds to the average search paths, which has size $c\log n$ for a constant c very close to 1 (see [131]); for skip-lists, $p = 0.5$. Note that $comb(\pi_s)$ corresponds to the number of comparisons performed in the search path π_s and that $size(T)$ corresponds to the total number of decision nodes in search tree T.

Table 3.1 summarizes the comparison results. For red-black trees, expectation corresponds to the average search path, which has size $c\log n$ for a constant c which is very close to 1 (see [131]), whereas for skip-lists it corresponds to the random skip-list structure. To simplify the comparison, we choose parameter $p = \frac{1}{2}$ for the two skip-lists. The multi-way skip-list DAG has better performance (achieves lower constants) when compared with the standard skip-list. On the other hand, we observe an interesting trade-off on the

performance that red-black tree and multi-way skip-lists achieve: in multi-way skip-lists the combined and boundary sizes are larger but the node size of the tree is less.

3.4 Data Authentication Through Hashing

In this section, we apply our results of Sections 3.2 and 3.3 in *hash-based data authentication*, that is data authentication based on cryptographic hashing. We focus on *authenticated dictionaries*, where membership queries on sets are authenticated. We show that this problem is a hierarchical data processing problem. Applying our results to authenticated dictionaries, we get a logarithmic lower bound on the authentication cost for any authentication scheme that uses cryptographic hashes and a new authenticated dictionary based on skip lists with authentication cost closer to optimal.

3.4.1 Authenticated Data Structures

Authenticated data structures provide a model of computation where an untrusted *directory* can answer queries issued by a *user* on a data structure on behalf of a trusted *source* but provide a proof of the validity of the answer to the user. The data source ideally signs only a single *digest* of the data. For *data authentication through hashing* a hash function is systematically used to produce this digest. On a query, along with the answer, the signed digest and some information that relates the answer to this digest are also given to the user and these are used for the answer verification.

In particular, the model involves a structured collection X of objects, the *source*, the *directory*, and the *user*. A repertoire of *query operations* and optional *update operations* are assumed to be defined over X. The role of each party is as follows. The *source* holds the original version of X. Whenever an update is performed on X, the source produces *update authentication information*, which consists of a signed time-stamped statement about the current version of X. The *directory* maintains a copy of X. It interacts with the source by receiving from the source the updates performed on X together with the associated update authentication information. The directory also interacts with the user by answering queries on X posed by the user. In addition to the answer to a query, the directory returns *answer authentication information*, which consists of (i) the latest update authentication information issued by the source; and (ii) a *proof* of the answer. The *user* poses queries on X, but instead of contacting the source directly, it contacts the directory. However, the user trusts the source and not the directory about X. Hence, it verifies the answer from the directory using the associated answer authentication information. The data structures

used by the source and the directory to store collection X, together with the algorithms for queries, updates, and verifications executed by the various parties, form what is called an *authenticated data structure*.

Authenticated Dictionary. We focus on the dictionary problem where we want to answer membership queries about a set of objects in the previous authentication model. Let X be a data set owned by the source that evolves through update operations insert and delete. Membership queries exists are issued on X. A (multivariate extension of a) cryptographic hash function h is used to produce a *digest* of set X, which is signed by the source. In our study, we actually consider a more general model where more than one digests are produced and signed by the source. These digests are computed through a *hashing scheme* over a directed acyclic graph (DAG) that has k *signature* nodes $t_1, \ldots, t_k$ and stores the elements of X at the source nodes (see [59]). Each node u of G stores a *label* or *hash value* $L(u)$ such that if u is a source of G, then $L(u) = h(e_1, \ldots, e_p)$, where $e_1, \ldots, e_p$ are elements of X, else (u is not a source of G) $L(u) = h(L(w_1), \ldots, L(w_l), e_1, \ldots, e_q)$, where $(w_1, u), \ldots, (w_l, u)$ are edges of G and $e_1, \ldots, e_q$ are elements of X (p, q and l are some constant integers). We view the labels $L(t_i)$ of the sink nodes t_i of G as the digests of X, which are computed via the above DAG G.

The authentication technique is based on the following general approach. The source and the directory store identical copies of the data structure for X and maintain the same hashing scheme on X. The source periodically signs the digests of X together with a time-stamp and sends the signed time-stamped digests to the directory. When updates occur on X, they are sent to the directory together with the new signed time-stamped digests. In this setting, the update authentication information has $O(k)$ size. When the user poses a query, the directory returns to the user (1) a signed time-stamped digest of X, (2) the answer to the query and (3) a proof consisting of a small collection of labels from the hashing scheme (or of data elements if needed) that allows the recomputation of the digest. The user validates the answer by recomputing the digest, checking that it is equal to the signed one and verifying the signature of the digest; the total time spent for this process is called the *answer verification time*. Security (against the possibility that the user verifies a, forged by the directory, proof for a non-authentic answer), typically follows from the properties of the signature scheme and the hash function.

Authentication Overhead. Now we study the performance overhead of computations related to authentication in an authenticated dictionary based on a hashing scheme (the

analysis is valid for any ADS). This overhead, called *authentication overhead*, consists of *time overhead* for the (*i*) maintenance of the hashing scheme after updates, (*ii*) generation of the answer authentication information in queries, and (*iii*) verification of the proof of the answer; *communication overhead*, defined as the size of the answer authentication information; *storage overhead*, given by the number of hash values used by the authentication scheme; and *signature overhead*, defined as the number of signing operations performed at the source (and thus the number of signatures sent by the source). Even with the most efficient implementations, the time for computing a hash function is a few orders of magnitude larger than the time for comparing two basic numerical types (e.g., integers or floating-point numbers). Thus, the rehashing overhead dominates the update time and the practical performance of an ADS is characterized by the *authentication overhead*, which depends on the hash function h in use and the mechanism used to realize a multivariate hash function from h.

3.4.2 Cryptographic Hash Functions

The basic cryptographic primitive for an ADS is a *collision-resistant hash function* $h(x)$ that maps a bit string x of arbitrary length to a hash value of fixed length (e.g., 128 or 160 bits), such that collisions (i.e., distinct inputs that hash to same value) are hard to find. We refer to h simply as *hash function*. Generic constructions of hash functions, the *Merkle-Damgård constructions* [30, 95], are modeled by iterative computations (see, e.g., [137]) based on a *compression function* $f(z, y)$ that maps a string z of N bits and a string y of B bits to an output string of N bits. The input string x is preprocessed (using a *padding rule*) into a string y whose length is a multiple of B. Let $y = y_1 \| y_2 \| ... \| y_k$, where each y_i has length B and let z_0 be a public *initial value* of N bits. Then $h(x) = z_k$, where z_k is given by the following iterative application of function f: $z_1 = f(z_0, y_1), z_2 = f(z_1, y_2), \ldots, z_k = f(z_{k-1}, y_k)$.

Lemma 3.4.1. *There exist constants c_1 and c_2 such that, given an input string x of size ℓ, the iterative computation of a hash function $h(x)$ takes time $T(\ell) = c_1\ell + c_2$.*

Note that the constants in Lemma 3.4.1 depend on the compression function in use and, thus, they may depend on some security parameter k. Still, the dependency of the hashing time on the input length is *linear*. This is a very general assumption that holds for any collision resistant function. For instance, for hash functions based on block ciphers or on algebraic structures and modular arithmetic (e.g. based on discrete logarithm using Pedersen's commitment scheme [113]) or custom designed hash function (e.g, SHA-1).

Multivariate Hash Functions. Let $h(x)$ be a hash function. In order to realize a hashing scheme, we extend h to a multivariate function using *string concatenation*. Namely, we define $h_C(x_1, ..., x_d) = h(x_1 \| ... \| x_d)$. There exist alternative realizations of a multivariate hash function. For example, one may use h_C for $d = 2$ and use a binary hash tree for $d > 2$. The following lemma states that, without loss of generality, we can restrict our analysis to the concatenation hash function h_C.

Lemma 3.4.2. *Any realization of a d-variate function $h(x_1, ..., x_d)$ can be expressed by iterative applications of h_C expressed through a hashing scheme G.*

3.4.3 Cost of Data Authentication Through Hashing

Let G be any hashing scheme used to implement an hash-based authenticated dictionary for set X of size n, where the hash values stored in k nodes of G have been digitally signed by the source. A node with signed hash value is called a *signature* node. Hashing scheme G along with the k signature nodes can be viewed as a DAG scheme $\Gamma = (G, S, n, k)$, where S is the set of k signatures nodes in G and there are exactly n source nodes in G corresponding to elements in X. Each cost parameter of the authentication overhead, (update, query or verification time overhead and storage overhead), is expressed as some structural metric of a subgraph H of the hashing scheme G. In general, the node size $size(H)$ corresponds to the number of *hash operations* that are performed at some of the three parties (source, directory or user) and the degree size $indeg(H)$ corresponds to the total number of hash values that participate as operands in these hash operations. In particular, each cost parameter of the authentication overhead depends *linearly* on $size(H)$ and $indeg(H)$ for some subgraph H of G.

We have for any authenticated dictionary implemented in the model of data authentication through hashing.

Lemma 3.4.3. *Let $\Gamma = (G, S, n, k)$ be any hashing scheme used to implement a hash-based authenticated dictionary for set X, where special nodes are signature nodes. Let s be a source node of G storing element $x \in X$, G_s be the subgraph of G that is reachable from s, and π_s the associated path of s. We have:*

1. *an update operation on x has update time overhead that is bounded by $comb(G_s)$;*
2. *a query operation on x has communication overhead that is lower bounded by $bnd(\pi_s)$ and verification time overhead that is lower bounded by $comb(\pi_s)$;*

3. *the storage overhead is $size(G)$;*

4. *all involved computations are performed sequentially and hierarchically according to the hierarchy induced by G, where at any node v of G computations have time or space or communication complexity proportional to $indeg(v)$.*

Proof. For an update operation (insert or delete) on an element at source node s in G, the hash values stored in the nodes of the reachable from s subgraph G_s of G need to be updated. Updating the hash value of node u of G_s using the concatenation multivariate hash function (Lemma 3.4.2 justifies this choice) takes time $c_1 indeg(u) + c_2$ (Lemma 3.4.1), thus the update operation is performed hierarchically in time $c_1 indeg(G_s) + c_2 size(G_s)$, which is $\Omega(comb(G_s))$. For a query operation exists the query element is first located in X and suppose it is found in X [5]. Providing the proof to the user involves collecting a set of hash values that can be used to recompute a data digest. For this reason, a path of minimum authentication cost from source node s storing the query element is found and this path is exactly the associated path π_s of s, the minimum combined size path from s to a signature node. Regarding the size of proof that is sent to the user, we note that in order the user to compute the hash value that is stored at node u of G, exactly $indeg(u) - 1$ hash values need to be sent by the directory. Thus the proof consists of $bnd(\pi_s)$ labels and the communication overhead is $\Omega(bnd(\pi_s))$. In addition, the user verifies this proof by hierarchically hashing the hash values of the proof along path π_s in time $c_1 indeg(\pi_s) + c_2 size(\pi_s)$, which is $\Omega(comb(G_s))$. Finally, for the storage overhead, clearly $size(G)$ hash values are stored in the data structure at the source and the directory. □

Thus, hash-based authentication of membership queries is a hierarchical data processing problem, where operations insert/delete are related to the update cost and operation exists to the query cost of the hashing scheme in use. Theorems 3.2.1, 3.2.2 and 3.2.3 suggest that, for the dictionary problem, signing more than one hash values does not help and tree hashing structures are optimal. Finally, these theorems and the fact that signature nodes are signed periodically also translate to the following.

Theorem 3.4.1. *In the data authentication model through hashing, any hashing scheme with k signature nodes that implements an authenticated dictionary of size n has*

- $\Omega(\log \frac{n}{k})$ *worst-case update and verification time overheads;*

[5] Without loss of generality we consider only positive answers. Standard techniques to authenticate negative answers have similar authentication overhead.

- $\Omega(\log \frac{n}{k})$ *worst-case communication overhead; and*
- $\Omega(k)$ *signature overhead.*

Finally, Lemma 3.4.3 suggests the use of our skip-list DAG to implement an authenticated dictionary.

Theorem 3.4.2. *There exists a skip-list authenticated dictionary of size n and probability parameter p that achieves the following expected performance. For any fixed element in the skip-list and constants c_1 and c_2 that depend on the hash function h in use, the expected hashing overhead of an update or verification operation is upper bounded by $(1-p)(2c_2 + \frac{(1+p)^2}{p}c_1)\log_{\frac{1}{p}} n + O(1)$, the expected communication cost is upper bounded by $(1-p)(\frac{1+p^2}{p})\log_{\frac{1}{p}} n + O(1)$ and the expected storage overhead is upper bounded by $(1 + pq^2 + pq + \frac{p}{q(2-pq^2)})n$, where $q = 1-p$.*

In particular, by our experimental value of $1.41n$, when $p = \frac{1}{2}$, for the node size $size(G)$ of the skip-list DAG G, such an implementation benefits low storage cost: on average only $1.41n$ hash values are stored.

3.5 Multicast Key Distribution Using Key-Graphs

In this section, we apply results from Sections 3.2 and 3.3 in multicast key distribution using key-graphs. We prove that key-trees are optimal compared to general key-graphs and derive logarithmic lower bounds for involved computational and communication costs. In contrast to previous *amortized* logarithmic lower bounds on the communication cost that any protocol requires [135, 99], we proof *exact worst case* lower bounds and our proof is more general, since it does not depend on any series of update operations. Even though general graphs were initially defined for this model [147], only tree DAGs have been studied. We prove the optimality of tree DAGs.

3.5.1 Multicast Key Distribution

The problem refers to the confidentiality security problem in multicast groups. A *group* consists of a set of n *users* and a *group key controller*. Private key cryptography is used to transmit *encrypted* multicast messages to all the users of the group. These messages are encrypted using a *group key* available to all the current users of the group. The security problem arises when updates on the group are performed, i.e., when users are added or removed from the group. The goal is to achieve *non-group confidentiality*, i.e., only members

of the group can decrypt the multicast messages, and *forward (backward) secrecy*, i.e., users deleted from (added in) the group can not decrypt messages transmitted in the future (past). No collusion between users should break any of the above requirements.

3.5.2 Communication Complexity for Key-Graphs

In this model, the group controller, a trusted authority and, conventionally, not member of the group, is responsible for distributing secret keys to the users and replacing (updating) them appropriately after user updates (additions/removals) in the group. The idea is that a set of keys, known to the controller, is distributed to the users, so that a key is possessed by more than one user and a user possesses more than one keys. In particular, any user is required to have a *secret key* that no other user knows and all users possess a *group key*, which is used for secure (encrypted) transmissions. In this model, on any user update, a subset of the keys needs to be updated to preserve the security requirements. Some keys are used for securely changing the group key as needed and for updating previous keys that have to be replaced. That is, new keys are encrypted with existing valid keys or with other previously distributed new keys.

Two extreme, trivial and inefficient solutions in this model are the following: each user possesses only one secret to other users key, where on any user removal from the group $n-1$ messages must be transmitted; alternatively, one key for all possible subsets of group users is used, but now the number of keys grows exponentially. *Key-graphs* [143, 147] provide a framework to implement this idea. A key-graph models the possession of keys by users and the computations (for key encryption at the controller and key decryption at the users) and message transmissions that need to be performed after any update. A key-graph is a single-sink DAG G that the group controller and users know and that facilitates group updates. Source nodes in G correspond to users and store their individual secret keys and all non source nodes correspond to keys that can be shared among many users. A user possesses *all* and *only* the keys that correspond to the subgraph G_s of G that is reachable from its source node s. On any update of the user corresponding to s, these keys have to change (the group key at the root is always among them) to achieve forward and backward secrecy. A new keys is distributed by being sent encrypted by an old or previously distributed key.

Cost parameters. The cost parameters associated with key distribution using key-graphs after an update are: (i) the computational cost at the controller, the *encryption cost*, for encrypting all new keys and thus producing the messages for transmission, and the computational cost at a user, the *decryption cost*, for decrypting received messages and

updating her keys, (*ii*) the *communication cost*, the number of transmitted messages, and (*iii*) the total number of keys stored at the key controller or a user. We can view a key-graph G as DAG scheme $\Gamma = (G, S, n, 1)$, where S consists of the unique sink node of G, called *group* node and n source nodes correspond to the users. Each cost parameter of key distribution is expressed as some structural metric of a subgraph of G. The node size corresponds to keys stored at users and also to key generations and the degree size corresponds to the number of encryptions and decryption performed during the update. In particular, each cost parameter depends linearly on $size(H)$ and $indeg(H)$ for some subgraph H of G.

Definition 3.5.1 (Reduced Key-graphs)**.** *Let $\Gamma = (G, S, n, 1)$ be a key-graph scheme and let v be a node in G. The* support *$Sup(v)$ of v is the set of source nodes of G that can reach v through directed paths, i.e., the set of users that possess the key stored at v. Note that $s \in Sup(v)$ if and only if $v \in G_s$. Let $U = \{u_1, ..., u_k\}$ be a set of nodes of G and let $T \subseteq V_{source}$ be a set of source nodes of G. We say that set U* spans *set T if*

$$\bigcup_{1 \leq i \leq k} Sup(u_i) = T.$$

A node v is said to be safe *if v is a source node or if $indeg(v) > 1$ and any node set that spans $Sup(v)$ and does not include v has size at least $indeg(v)$. Key-graph scheme Γ is said to be* reduced*, if all nodes in G are safe.*

Lemma 3.5.1. *Let $\Gamma = (G, S, n, 1)$ be a reduced key-graph scheme used for the multicast group management problem. Then we have:*

1. *an update operation on a user that corresponds to a source node s has communication cost at least $indeg(G_s)$ and encryption cost at least $comb(G_s)$;*
2. *the key-graph stores $size(G)$ keys in total; and*
3. *all involved computations are performed sequentially and hierarchically according to the hierarchy induced by G, where at any node v of G computations have time or space or communication complexity proportional to $indeg(v)$;*

Proof. (1) By the definition of a key-graph scheme, all the keys stored at nodes of G_s (i.e., $size(G_s)$ keys) need to be updated. Consider the key stored at a node $v \in G_s$. We now show that at least $\ell = indeg(v)$ messages must be broadcasted by the controller in order for this key to be updated. Either $\ell = 0$ or $\ell \geq 2$. If $\ell = 0$, then v is a source node and

we need no broadcast message to update its key. If $\ell \geq 2$, then all and only the users that correspond to nodes of $Sup(v)$ need to be able to decrypt the broadcasted messages of the encrypted new key of v. Suppose that fewer than ℓ messages suffice in updating the key of v. This implies that there exists a set of nodes $U = \{u_1, ..., u_k\}$ of G such that:

- U spans $Sup(v)$;
- U does not contain v; and
- $k < \ell$.

This is a contradiction since Γ is reduced and thus v is safe, i.e., $Sup(v)$ cannot be spanned by a set not containing v of size less than ℓ. In particular, we have that $Sup(u)$ is spanned by the set of predecessor nodes $w_1, ..., w_\ell$ of v. Thus, in order to update all the nodes of G_s, the controller needs to broadcast at least $indeg(G_s)$ messages. Additionally, when sending t messages to update the key of a node v, the encryption cost is proportional to $1+t$, since, one (new) key generation and t encryptions of this key are performed in order to create the messages. It follows that the encryption cost is lower bounded by $comb(G_s)$.

(2) By the definition of a key-graph scheme, $size(G)$ keys are stored. □

Thus, multicast key distribution using reduced key-graphs is a HDP problem, where the overhead of an update in the group is related to the update cost of the underlying DAG scheme. By studying more carefully reduced key-graph schemes and using Theorems 3.2.1 and 3.2.2, we now prove the main result of the section.

Theorem 3.5.1. *For a multicast key distribution problem of size n using key-graphs, in the worst case, an update operation in the group requires at least $\lfloor \log n \rfloor + 1$ communication cost and $\Omega(\log n)$ encryption and decryption costs. Also, key-tree structures are optimal over general key-graphs.*

Proof. We describe a transformation $R_v(\cdot)$ on key-graphs that given a non-safe node v in G performs changes on G (edge deletions and possibly node deletions occur). Let $\Gamma = (G, S, n, 1)$ be a key-graph scheme. Transformation R_v can be applied to G only if v is a non-safe node of G such that all nodes in G_v other than v are safe in G. (Recall that G_v is the subgraph of G that is reachable from node v through directed paths.) Let v be a non-safe node in G satisfying the above property. Let $W = \{w_1, ..., w_d\}$ be the set of predecessor nodes of v, where $d = indeg_G(v)$, $d > 0$. Transformation R_v is defined as a series of steps; we consider the following two cases, depending on the in-degree d of v.

Case I (replacement) $d \geq 2$

1. since v is not safe, we can find a minimum-size set of nodes $U = \{u_1, ..., u_\ell\}$ such that U spans $Sup(v)$, U does not contain v, and $\ell < d$; note that set U may contain one or more predecessor nodes of v;
2. edges $(w_1, v), ..., (w_d, v)$ are removed from G (this step may cause some predecessors of v to become sink nodes);
3. edges $(u_1, v), ..., (u_\ell, v)$ are added to G (note that edges removed by the previous step may be reinserted by this step);
4. while G has a sink node z distinct from the special node (the unique node of set S), remove z and its incoming edges.

Case II (contraction) $d = 1$

1. let w be the predecessor of v and let $z_1, ..., z_m$ be the successor nodes of v;
2. if $m > 0$, then edges $(v, z_1), ..., (v, z_m)$ are removed from G and, additionally, edges $(w, z_1), ..., (w, z_m)$ are added to G;
3. edge (w, v) and node v are removed from G.

Note that, in case I, R_v possibly makes node v safe (if $\ell > 1$) and that, in case II, R_v deletes node v. Also no cycle is introduced in step 3 of case I since node v cannot reach any node u_i through a directed path, or otherwise, u_i, a node of G_v, is not safe, a contradiction regarding the precondition in applying R_v on G. Obviously, no cycle is introduced in case II.

Let $G' = R_v(G)$ be the graph that results from applying transformation R_v to graph G. We have the following:

- the set of nodes of G' is included in the set of nodes of G, and thus the number of nodes of G is greater than or equal to the number of node of G';
- the set of source nodes of G' is the same as the set of source nodes of G, and thus both G and G' have n source nodes;
- R_v deletes at least one edge, thus G' has fewer edges than G;
- in case I, v is not deleted and $Sup_G(v) = Sup_{G'}(v)$; in case II, v is deleted and $Sup_G(v) = Sup_{G'}(w)$.

Let $\Gamma' = (G', S, n, 1)$ be the corresponding transformed scheme after transformation R_v has taken place. Given any key-update algorithm A for scheme Γ, we now derive a corresponding transformed key-update algorithm A' for scheme Γ'. Transformed algorithm A' is defined as follows.

- To update the key of a node z of G' that is distinct from v, algorithm A' mimics algorithm A. Namely, it performs exactly what A performs when it updates the key of node z in G.
- Regarding node v, algorithm A' either broadcasts $\ell = indeg_{G'}(v) > 0$ messages (case I), encrypting the new key for node v using the keys stored at the predecessor nodes $u_1, ..., u_\ell$ of v in G', or it broadcasts no message if v is not a node of G' (case II). Note that v is not a source node, otherwise, since any source node is safe, transformation R_v would not have been applied.

We now show that for any source node s, the communication cost of algorithm A for updating s in Γ is greater than or equal to the communication cost of algorithm A' for updating s in Γ', i.e., the number of messages sent by A is greater than or equal to the number of messages sent by A'. We consider the following two cases.

- If $s \notin Sup_G(v)$, then $G_s = G'_s$, so algorithms A and A' have exactly the same executions and send the same number of messages.
- If $s \in Sup_G(v)$, then,
  - for case I, G'_s has fewer edges than G_s since transformation R_v reduces the indegree of v, i.e., $indeg_{G'}(v) < indeg_G(v)$. Also, we have $indeg(G'_s) < indeg(G_s)$ and $size(G'_s) \leq size(G_s)$, i.e., G'_s has no more nodes than G_s. Algorithm A sends at least as many messages as algorithm A' does. Let k be the number of messages that algorithm A sends to update the key at v and k' be the number of messages that algorithm A' sends for the same task. We have $k' = indeg_{G'}(u) = \ell$ and $k \geq \ell$, or otherwise, if $k < \ell$, then there would be a set of nodes of size smaller than ℓ in G that spans $Sup_G(v) = Sup_{G'}(v)$ and thus, transformation R_v was not performed correctly, a contradiction.
  - for case II, G'_s has fewer edges than G_s since transformation R_v deletes one edge. We also have $indeg(G'_s) < indeg(G_s)$ and $size(G'_s) < size(G_s)$, i.e., G'_s has fewer nodes than G_s, since v is deleted. Algorithm A sends more messages than algorithm A' does: A sends at least one message to update the key of v, but A' sends no message.

Starting with key-graph scheme $\Gamma = (G, S, n, 1)$, while the current graph has non-safe vertices, we apply the above transformation repeatedly, yielding a series of graphs $G = G_0, G_1, G_2, \ldots$ and corresponding schemes $\Gamma = \Gamma_0, \Gamma_1, \Gamma_2, \ldots$, where $G_{i+1} = R_{v_i}(G_i)$, for some vertex v_i of G_i for which transformation R_{v_i} is applicable on G_i. That is, v_i is a non-safe node of G_i such that all nodes in $(G_i)_{v_i}$ other than v_i are safe in G_i. Recall that $(G_i)_{v_i}$ is the subgraph of G_i that is reachable from node v_i through directed paths. Since each transformation reduces the number of edges and does not add any new nodes and never makes a safe node non-safe, there exists a finite sequence of q transformations, indexed by nodes $v_0, ..., v_{q-1}$, resulting in a scheme Γ_q whose graph G_q is reduced (i.e., it has no non-safe nodes).

Given a key management algorithm A for Γ, let $A_0 = A$ and, for $i = 0, \ldots, q-1$, let A_{i+1} be the algorithm for Γ_{i+1} obtained by transforming algorithm A_i for Γ_i. By repeating the argument above, we have that, given a source node s, for $i = 0, \ldots, q-1$, for the update of s, algorithm A_i in Γ_i has communication cost greater than or equal to the communication cost of algorithm A_{i+1} in Γ_{i+1}.

Let s^* be a source node whose update has largest communication cost for algorithm A_q in Γ_q. For $i = 0, \ldots, q$, we define c_i to be the communication cost of algorithm A_i for Γ_i in the update of node s^*. We have that $c_i \geq c_{i+1}$ for $i = 0, \ldots, q-1$. Also, since graph G_q is reduced, by Lemma 3.5.1 and Theorem 3.2.1, we have that $c_q \geq \lfloor \log n \rfloor + 1$. Thus, we obtain that $c_0 \geq \lfloor \log n \rfloor + 1$. We conclude that for any key-management algorithm for a scheme $\Gamma = (G, S, n, 1)$, there is an update whose communication cost is at least $\lfloor \log n \rfloor + 1$.

Recall that the encryption and decryption costs are each lower bounded by the communication cost, thus, we get that for any instance of the multicast key distribution problem of size n using key graphs, there exists an update that has $\Omega(\log n)$ encryption and decryption costs. Finally, Theorem 3.2.2 implies that the best reduced key-graphs are the key-trees. □

We observe that by the proof of Theorem 3.5.1, we can view the multicast key distribution problem as a HDP problem.

3.6 A New Skip-List Version

From Lemma 3.2.2 and Theorem 3.3.1, we have a version of the skip-list data structure for searching in a totally ordered set $(X, \preceq)$ with expected number of comparisons $\preceq$ close to the theoretical optimal $\lfloor \log n \rfloor + 1$, up to an additive constant term. Our DAG scheme and the new skip-list version can be viewed as a *multi-way* extension of the skip-list data

structure, in the same way multi-way trees (e.g., B-trees, 2-4 trees) are extension to binary search trees. We call the new skip-list version *multi-way skip-list*.

Theorem 3.6.1. *There is a* multi-way *version of a skip-list for set X of size n and probability parameter p, where the expected number of comparisons performed while searching in X for any fixed element is at most $\frac{(1-p)(1+p^2)}{p\log_2\frac{1}{p}}\log_2 n + O(1)$, or $1.25\log_2 n + O(1)$ for $p = \frac{1}{2}$.*

Proof. It follows from Lemma 3.2.2 and Theorem 3.3.1. Since for DAG scheme $\Delta = (T, r, n, 1)$, T is a directed tree, from Lemma 3.2.2 we get a transformation of T into a search tree T' for set X, where interior nodes of T' are assigned with elements from X. It follows that every element x_s in X stored at source node s of T' can be located with $bnd(\pi_s)$ comparisons, which completes the proof. □

Regarding the above new skip-list version, the idea is to use our skip-list DAG scheme to create a *search tree structure* in the skip-list for searching elements of X. The multi-way skip-list data structure is a appropriately modified version of the regular skip-list data structure, such that the search tree T' of Theorem 3.6.1 is implicitly represented in the skip-list and searches are performed according to search tree T'. That is, we can keep the simplicity in creating and updating a skip-list, but we can save element comparisons by using our skip-list DAG. Note that for bridges of size $k \geq 2$, by keeping the appropriate elements that advance the search in this bridge, $k-1$ (instead of k) comparisons are needed. Skipping most of the details, to implement this skip-list version, elements that are used in a bridge b to advance the search in a child bridge of b must be stored in the *entrance* (top-left) list node of the bridge. Elements are inserted and deleted as in the regular skip-list data structure. The update operations should only maintain the appropriate elements at the entrance list nodes of the bridges traversed by the update; this can be easily achieved but appropriately modifying the search procedures.

We note that also in the regular skip-list any search corresponds to a path in a search tree. This tree-like interpretation of skip-lists is known in the literature (e.g., see [101]). Our result provides a new tree-like interpretation of skip-list with closer to optimal search performance. In particular, with respect to the expected number of comparison in a search, the multi-way skip-list version achieves the best known performance for a skip-list. Indeed, in [120, 121] it is shown that the expected number of comparisons for a search in a skip-list with probability parameter p is $(\log_2 n)/(p\log_2\frac{1}{p}) + O(1)$. In the same work, an improved – in terms of number of comparisons – skip-list version gives $\frac{1-p^2}{p\log_2\frac{1}{p}}\log_2 n + O(1)$ expected comparisons for a search. We are not aware of any improved skip-list based scheme with

better (smaller) logarithmic constant. Our multi-way skip-list version reduces the expected number of comparisons down to $\frac{(1-p)(1+p^2)}{p \log_2 \frac{1}{p}} \log_2 n + O(1)$. For instance, when $p = \frac{1}{2}$, the expected number of comparisons when searching in the (improved) standard skip-list achieves is $1.5 \log n + O(1)$, whereas, using the new multi-way skip-list the expected number of comparisons for searching an element is $1.25 \log n + O(1)$, that is, the logarithmic constant of the expected number of comparisons for a search drops from 1.5 to 1.25. Interestingly, multi-way skip-lists have a more explicit tree structure than standard skip-lists, thus in fact, they constitute a new *randomized* search tree (see, e.g.,[3, 87]).

3.7 Conclusions

In this chapter, we introduced the concept of hierarchical data processing. This concept defines a new class of problems which models computations on elements and associated values that share certain properties. Computations are carried out hierarchically, according to the structure of an associated with the problem directed acyclic graph. Moreover, all costs that are related to the computations are fully characterized by the associated DAG, where in principle these costs depend on certain structural properties of the graph. Hierarchical data processing constitutes an interesting new theoretical framework for analyzing computational problems and designing efficient data structuring techniques.

We have proved logarithmic lower bounds for several cost measures related to hierarchical data processing, by studying structural measures of any DAG and by relating these measures to the number of comparison performed in a search structure. Overall, we have drawn an analogy between hierarchical data processing problems and searching by comparison. This connection not only serves as the basis for our theoretical results on the complexity of hierarchical data processing problems, but also provides us with an interesting and useful explanation of the computational difficulty that is intrinsic in them. We also proved the optimality of tree structures for any hierarchical data processing problem.

In view of the logarithmic lower bounds and the optimality of trees, we have also designed and analyzed a new randomized tree-like DAG. This DAG enjoys certain nice properties with respect to the cost measures related to the complexity of hierarchical data processing problems. Through the computational equivalence between hierarchical data processing and searching by comparison, we also get a new version of skip-list, where the expected number of comparisons of a search if closer to the theoretical optimal. This skip-list version improves the performance of previous versions.

We further apply our framework of hierarchical data processing to model two information security problems. We prove logarithmic lower bounds and efficient constructions for authenticated dictionaries in the model of data authentication through cryptographic hashing as well as for multicast key distribution using key-graphs. We show how these problems involve hierarchical data processing and accordingly get new results. We believe that more applications and more problems can be modeled as hierarchical data processing problems. Our framework not only provides a unified treatment of two interesting and seemingly unrelated problems, but also provides a general tool in studying other computational problems.

An interesting open problem is the derivation of non-trivial bounds on new cost measures of DAG schemes that characterize the complexity of different class of problems and, in particular, the derivation of lower bounds on the complexity of other problems related to data authentication. Also, it is interesting to further explore hierarchical data processing in modeling and analyzing different classes of problems. Finally, the trade-off between two important computational measures related to hierarchical data processing which was observed for two basic data structures, suggest that it is worth to further investigate the possibility of designing authentication structures that have performance closer to optimal.

An extended abstract of the results discussed in this chapter appears in [142].

Chapter 4

Authentication of Data Streams

4.1 Introduction

The authentication of multicast transmissions of data streams is a central problem in information and network security. Data transmission in a multicast setting involves a sender—the source of the data—sending data to a large set of receivers, where data is transmitted as a stream of packets over an underlying network. The authentication problem that arises is the verification of the received stream being authentic, that is, being (part of) the original stream sent by the data source. A large and still growing set of Internet's applications that are based on multicast data transmissions justify the importance of this problem. Distributed information and data management systems, systems built on top of peer-to-peer networks, digital broadcasts and public subscription systems typically include extensive multicast of information. Additionally, numerous data-driven and multicast-group applications involve large-scale stream-based dissemination of high volumes of data from one source to many users; examples include content-based networks, online data-processing financial applications, data-monitoring systems over sensor networks and various data-flow-oriented systems for Web services. Of course, the problem also includes, as a simpler case, the point-to-point transmission of data streams and, thus, captures numerous stream-based client-server applications over the Internet.

Technically, the problem of multicast authentication is a challenging one and has attracted a great amount of interest during the last years. One of the distinguishing properties of multicast data-stream transmissions is certainly the fact that, since data is unstructured and flat, for the majority of the applications packet losses are tolerated and, consequently, data transmission does not need to be reliable. For instance, IP multicast is implemented

with a best-effort delivery mechanism over the UDP transport protocol and packets can be lost due to failures. Thus, in principle, the stream that reaches a receiver may differ from the transmitted one. As a result, any authentication scheme for multicast streams should verify as many as possible of the received packets without assuming the availability of the entire original stream. In addition, it should resist against any type of attack by an adversary, even when the adversary completely controls the underlying network. Although what happens more often in practice is that packets get lost because of errors, what authentication dictates is protection against an adversarial network behavior. Indeed, the main characteristic of the problem is by definition the existence of an entity acting maliciously in between the sender and an honest receiver. In practice, the role of the adversary may play any of the parts of the underlying network, such as ISPs, routers or malicious users. It is essential that the adversary is modeled as an entity of great power and that no assumptions about the network that limit adversary's behavior exist. Finally, any authentication scheme should be efficient, scalable and lightweight with respect to the final application.

Therefore, in the *multicast authentication problem*, we wish to authenticate a packet stream transmitted over a network that may adversarially drop packets, arbitrarily rearrange the order of the packets, and inject new packets into the stream. The authentication mechanism should be as general as possible and should not depend on any specific assumptions about the underlying network. Although this problem has been extensively studied, no formal definition has been given for it for this general version. Prior work on the subject has focused on a network model where either all the received packets are valid (authentic) or packets are lost according to some predefined random patterns (e.g., [51, 100, 116, 136]) or no packet injections occur (e.g., [110, 111]), or relatively strong and hard to meet conditions are assumed about the network behavior (e.g., [115, 116]). Thus, most of the previously proposed schemes rely on less general network models, tolerate only *erroneous* network behavior and are not resilient against an *adversarial* behavior of the network.

Of course, if each packet were signed by the sender, then the only damage the adversarial network could inflict is packet loss, as the receiver would simply reject packets whose signature is not verified. However, this simple "sign-all" solution is undesirable because of the repeated use by the sender of the critical and computationally expensive sign primitive for each transmitted packet and the heavy communication overhead caused by the addition of a signature to each packet. Additionally, this solution suffers by a simple denial-of-service attack at the receiver; one signature verification must be performed for each received packet, valid or not.

In this chapter, we formally define a general model for multicast authentication where

an adversary can perform various attacks on the transmitted streams. In this model, two parameters of the network, the *survival rate* and the *flood rate*, characterize the power of the adversary. Using this network model, we formally define the authentication problem for multicast transmissions and describe the notions of correctness and security that any authentication scheme should respect. We describe an efficient authentication scheme for this model that gives almost the same security guarantees as if each packet were individually signed, but requires only one signature operation for the entire stream and adds only a constant size authentication overhead per packet. Our technique uses a novel combination of Reed-Solomon error-correcting codes with standard cryptographic primitives, such as collision-resistant hashing and digital signatures. The use of error-correcting codes for multicast authentication was inspired by the previous use of erasure codes, as in [110, 111], for this problem. We essentially design authenticated error-correcting codes that constitute a new general-purpose authentication tool. Moreover, enhanced with cryptographic primitives, our error-correcting encoding derives an interesting connection between cryptography and coding theory.

In the rest of this section, we introduce our model, summarize our contributions and present previous work on multicast authentication.

4.1.1 Model and Contributions

We consider the problem of authenticating a stream of packets transmitted over a *fully adversarial* network. Namely, the network is controlled by an adversary who can destroy packets of her choice, arbitrarily rearrange the order of the packets, and inject new, arbitrarily constructed, packets. We limit the power of the adversary to modify a stream of n packets transmitted by the sender by introducing two parameters of the network, the *survival rate* α, $0 < \alpha \leq 1$, and the *flood rate* β, $\beta \geq 1$, which are assumed to be constants. A network with these two parameters, which we call an (α, β)-*network*, guarantees that despite the presence of the adversary, at least αn packets in the received stream are valid and the received stream contains at most βn packets.

The model is formally described in Section 4.3. For now, we note that the survival and flood rates are only used to model the adversary's behavior; not to limit it. An authentication scheme should operate correctly and securely for any values of α and β, even non-constants. We briefly justify the introduction of these network parameters—and, in essence, our choice to be constants—with the following observations. If too many packets are dropped or corrupted by the adversary, then the main problem is the *loss of data*, as

the small number of valid packets received may be useless even if authenticated. On the other hand, if the adversary can inject a very large number of packets, then we have a *denial-of-service attack*. In both cases, as α gets smaller or β gets larger, the authentication problem degenerates to data loss and denial-of-service attack. No need for authentication really exists in both of these extreme cases.

The contributions of our work can be summarized as follows:

- We provide a formal definition of multicast authentication over an (α, β)-network, where arbitrary packets are lost, injected, and rearranged, subject to a given survival rate α and flood rate β. We also give the requirements for an authentication scheme to be *correct* and *secure*.

- We present the first efficient and scalable multicast authentication scheme for an (α, β)-network. Our scheme is based on digital signatures, cryptographic hash functions and Reed-Solomon error-correcting codes. In essence, we design an authenticated Reed-Solomon error-correcting code that constitutes a new, powerful and general-purpose authentication tool. This last feature of our scheme provides a new interesting connection between coding theory and security. In particular, we show how, in the public-key model and the bounded computational model for communication channels, list-decoding can be transformed into unambiguous decoding.

- We prove the correctness and security of our scheme, analyze its performance in terms of various cost parameters, discuss design and implementation choices, and compare it with previous approaches. In particular, we show that our scheme adds to each transmitted packet only a small amount of authentication information, proportional to β/α^2, and that all the valid packets received are recognized, while all the invalid packets are rejected.

The only prior approaches that provide security in our adversarial model is (i) the inefficient "sign every packet" solution, which consists of either singing each packet individually or using a Merkle hash tree as in the scheme proposed by Wong and Lam [148] and (ii) a recently proposed scheme [68] by Karlof *et al.* that uses signature dispersal and a Merkle hash tree. The trivial solution of signing each packet individually is not viable due to heavy computational operations at both the sender and the receiver, but also because secret-key operations are expensive in terms of the security architecture as well. Our scheme, by amortizing one signature per a stream of size n, suffers from no such problem. On the other hand, the Merkle-tree-based authentication schemes [68, 148] have the drawback that the

per-packet communication overhead grows logarithmically with the number of packets sent. Indeed, each packet of a stream of size n carries authentication information of size $O(\log n)$. In contrast, our scheme achieves per-packet communication overhead independent of n and, thus, more efficiency and scalability.

4.1.2 Prior and Related Work

Previous work on multicast authentication considers both unconditionally secure and computationally secure authentication. Approaches based on the information theoretic model (see, e.g., [31, 132]) tend to be less practical. In the rest of this section, we overview approaches that use computationally secure authentication. We do so, by appropriately categorizing previous work according to the underlying authentication technique in use.

MAC-Based Approaches. Various approaches use secret-key cryptography and message authentication codes (MACs). The trivial solution here is having the multicast group members (i.e., all the receivers) sharing a secret key and including a MAC into every packet sent, but this scheme is not secure, as any user can spoof packets. In another MAC-based trivial solution, each receiver has her own secret key and the sender possesses all such keys. To authenticate a stream, the sender adds to each packet a MAC for every receiver. This approach is not scalable because of the high communication cost.

Canetti *et al.* [20] described a MAC-based scheme that is secure with high probability against any coalition of w corrupted users, where $O(w)$ MACs are appended to each packet. This scheme is not fully scalable due to its communication overhead. Perrig *et al.* [115, 116] presented another MAC-based scheme, TESLA, where a MAC is appended to every packet and the key of the MAC is provided in some subsequent packet. To tolerate packet losses, the keys are generated by means of a hash chain. This approach has low communication overhead. However, it requires time synchronization between the parties. Two MAC-based schemes that make explicit use of the topology of a multicast tree were proposed in [149] by Xu and Sandhu. Both schemes are similar in concept to [115] and take denial-of-service and access control into consideration (namely, a corrupted packet is filtered out as soon as possible in the multicast tree and only legitimate group subscribers can authenticate the multicast packets). Both schemes assume the existence of secure and trusted routers at the nodes of the tree. In addition, the first scheme uses clock synchronization, whereas the second scheme relies on the existence of secure channels between the source and each of the receivers.

Boneh et al. [11] generalized MACs to a multicast setting by defining a new primitive

for multicast authentication called Multicast MAC (MMAC). A MMAC is a triplet of algorithms ($key-gen$,$mac-gen$,$mac-ver$) where: $key-gen$ produces secret key sk for the sender and secret keys $rs_1, \ldots, rs_n$ for the receivers; $mac-gen(M, sk)$ computes an authentication tag τ for message M and $mac-ver(M, \tau, rk_i)$ returns a "yes" or "no", checking whether τ is an authentication tag of message M. A MMAC must satisfy certain correctness and security constraints. In their work, Boneh *et al.* studied the existence of efficient MMACs, i.e., MMACs with short tag τ. They showed that any MMAC scheme can be transformed into a digital signature scheme of almost the same efficiency. Thus, any multicast authentication scheme not relying on additional assumptions on the network (such as synchronization, trusted routers, or secure channels) may as well use a signature scheme, which brings us to *signature amortization*. The result was also extended to the case where the adversary possesses a limited number of the receivers' keys and a lower bound on the length of the authentication tag was derived. The construction by Canetti *et al.* in [20] meets this bound, whereas TESLA by Perrig *et al.* in [115, 116], using time synchronization and one digital signature for bootstrapping, is not an exact MMAC scheme.

In view of the previous result, research efforts also focused on building faster signature schemes for signing every packet separately. Work on this direction includes (*i*) the use of one-time digital signatures by Gennaro and Rohatgi in [47] for on-line data stream multicast transmission but only in reliable communication channels, (*ii*) the use of k-time digital signatures by Rohatgi in [128] to speed up the signing rate with relatively short signature sizes and (*iii*) Perrig's BiBa one-time broadcast protocol [114].

Signature Amortization. Many approaches use the technique of signature amortization, where a single digital signature is used for the authentication of multiple packets. A first scheme that uses signature amortization over a hash chain appeared by Gennaro and Rohatgi in [47]. Each packet p_i is augmented with authentication information a_i, which is recursively defined as the hash of $p_{i+1} \circ a_{i+1}$ ($\circ$ denotes concatenation). Also, the augmented first packet $p_1 \circ a_1$ is digitally signed. This scheme has constant authentication overhead per packet but does not tolerate packets losses. In [148], a Merkle hash tree was used by Wong and Lam to amortize a signature over n packets. Namely, a hash tree is built on top of the hashes of the packets and the root hash value is digitally signed. Each packet is augmented with authentication information that consists of the signed root hash and the hashes of the siblings of the nodes on the path between the root and the leaf associated with the packet. The scheme tolerates packet losses but has logarithmic communication overhead per packet. In contrast, our approach, which also uses signature amortization, has *constant* per-packet

communication overhead.

Graph-Based Authentication. Graph-based authentication [51, 100, 116, 136] generalizes the idea of amortizing a signature over a hash chain in such a way as to tolerate packet losses. A single-sink directed acyclic graph (DAG) G is defined, where each vertex corresponds to a packet. A directed edge from packet p_i to packet p_j indicates that the authentication information a_j of packet p_j includes the hash of $p_i \circ a_i$. Also, the augmented packet $p_1 \circ a_1$ of the sink of the DAG is digitally signed. The validation of packets proceeds backward along the edges of the graph. Namely, if packet p_j has been validated and edge (p_i, p_j) exists in G, then the validity of packet p_i can be determined using the authentication information a_j of p_j. Graph-based authentication schemes offer probabilistic security guarantees provided packet losses occur randomly (i.e., they are not adversarially selected). In particular, they require that the signature packet will reach the receiver intact. Two packet loss patterns have been studied: the uniform model, where each packet is lost with a fixed probability and independently of other packets being lost, and the bursty model, where a packet is lost with a fixed probability and then a given number of successive packets are also lost.

In [116], Perrig *et al.* selected G to be an augmented-chain graph, consisting of a path plus additional edges that connect vertices at various distances. Golle and Modadugu in [51] proposed the use of another augmented-chain graph which is designed specifically to tolerate bursty packet losses. Random graphs and a new scheme that is resilient to multiple bursty losses were studied by Minner and Staddon in [100]. Finally, in [136], expander graphs were used by Song *et al.* The efficiency of graph-based authentication schemes was analyzed in [22] by Chan and experimentally studied in [28] by Cucinotta *et al.*

Erasure Codes. Park *et al.* [111] and Pannetrat and Molva [110] employed the use of erasure codes (e.g., [80, 81, 122]) for multicast authentication to tolerate adversarially-chosen packet losses and disperse one signature over a group of packets. In particular, information sufficient—if reconstructed at the receiver—to authenticate the received packets is encoded using an erasure code so that delivery of a constant fraction of packets guarantee the successful decoding of this information. The constructions are efficient in terms of communication cost and similar in principle. The two schemes only differ in that in [110], encoding is performed twice to reduce the size of the authentication information. The idea here is that a significant portion of the encoded information reaches the receiver for free through the valid packets. Both schemes are, however, vulnerable to a very simple attack:

a single injected packet can compromise the correctness of the decoding procedure at the receiver.

In [112], Park *et al.* identified a special case of this problem, where packets are altered, as a denial-of-service attack. They suggest the use of distributed fingerprints [76] (which, in turn, are based on error-correcting encoding) to tolerate a small number of symbol modifications (through packet alterations) of the erasure-encoded information. However, the proposed scheme lacks efficiency, since distributed fingerprints are used on top of the erasure encoding, and, more importantly, collapses for a specific packet-injection type of attack, where more than one packets (symbols) claim to be a specific packet (symbol). We note that our adversarial model includes this type of attack: many injected packets may pretend to be a specific original packet.

Recently, Karlof *et al.* [68] proposed a solution to the packet-injection problem that the erasure-based schemes have. They referred to this problem as *packet pollution* and introduced the notion of *distillation codes*, codes that tolerate both erasure and pollution of symbols. They realized a distillation code by using an erasure code and a one-way accumulator. Symbols are erasure encoded as usual, but also appended by a witness of set inclusion in the set of pre-encoded symbols. At the receiver, the idea is to partition the received symbols into groups of symbols so that received symbols of the same group belong in the same group of transmitted (not necessarily authentic) pre-encoded packets. Partitioning is feasible relying on the set inclusion properties of the one-way accumulator and, consequently, each group can be decoded and then examined to be authentic. Built on the scheme [112] by Park *et al.* and using Merkle's hash tree as a (weaker version of) one-way accumulator, this approach leads to a multicast authentication scheme, where both packet injections and packet erasure is tolerated. However, this scheme achieves *not constant* per-packet communication overhead, since each packet carries information of size $O(\log n)$ as witness (because of the use of Merkle tree), and suffers from the cumbersome decoding procedure with respect to the processing of packets at the receiver: the partition operation adds a considerable amount of computation and hashing and a considerable number of erasure decodings.

Recently Gunter *et al.* [61] have used erasure codes to tolerate packet injections but only random packet losses. In particular, in a less general network model, the adversary shares the transmission channel with the sender and is allowed to inject packets up to a certain transmission rate. The scheme involves the use of three different streams: (*i*) the data stream where packets only contain data, (*ii*) the hash and parity stream, which consists respectively of hash packets and encoded packets and (*iii*) the signature stream.

Signatures are verified selectively and injected packets are filtered out by processing all received packets in an exhausting way. The scheme achieves only probabilistic guarantees about the authentication of received packets. Obviously, no adversarially chosen packets are tolerated.

Related Work. In a recent related work [77], Krohn *et al.* have used erasure encoding techniques in combination with *homomorphic hashing* for the on-the-fly verification of erasure-encoded blocks. Applications involve peer-to-peer content distribution but also multicast transfers. However, their scheme relies on the following strong assumption: the receiver knows in advance the cryptographic hashes of the transmitted shares.

Techniques similar to ours have been recently used by Micali *et al.* [97] to study error correction in the bounded computational model.

4.1.3 Chapter Structure

The organization of the rest of the chapter is as follows. The cryptographic primitives and error-correcting codes used in this chapter are reviewed in Section 4.2. In Section 4.3, we describe in detail our adversarial network model and multicast authentication framework. Section 4.4 describes the construction of our authenticated error-correcting code and multicast authentication scheme and gives proofs of correctness and security. In Section 4.7 we analyze the performance of our scheme and compare it with various other proposed schemes in terms of security assumptions, underlying network model, resilience to packet loss and injection, computational effort at the sender and receiver, and communication overhead. We conclude in Section 4.8.

4.2 Preliminaries

In this section, we introduce some notation and define the cryptographic and coding primitives that we use in our construction.

4.2.1 Notation

Let A be an algorithm. By $A(\cdot)$ we denote that A has one input (resp., by $A(\cdot, \ldots, \cdot)$ we denote that A has several inputs). By $y \leftarrow A(x)$, we denote that y was obtained by running A on input x. If A is deterministic, then this y is unique; if A is probabilistic, then y is a random variable. If S is a finite set, then $y \leftarrow S$ denotes that y was chosen from S uniformly at random. By $y \in A(x)$ we mean that the probability that y is output by $A(x)$ is positive.

By $A^O(\cdot)$, we denote an algorithm that makes queries to an oracle O. I.e., this algorithm (Turing machine) will have an additional (read/write-once) query tape, on which it will write its queries in binary; once it is done writing a query, it inserts a special symbol "#". By external means, once the symbol "#" appears on the query tape, oracle O is invoked and its answer appears on the query tape adjacent to the "#" symbol. By $Q = Q(A^O(x)) \leftarrow A^O(x)$ we denote the contents of the query tape once A terminates, with oracle O and input x. By $(q, a) \in Q$ we denote the event that q was a query issued by A, and a was the answer received from oracle O.

Let b be a boolean function. By $(y \leftarrow A(x) : b(y))$, we denote the event that $b(y)$ is TRUE after y was generated by running A on input x. The statement $\Pr[\{x_i \leftarrow A_i(y_i)\}_{1 \le i \le n} : b(x_n)] = \alpha$ means that the probability that $b(x_n)$ is TRUE after the value x_n was obtained by running algorithms $A_1, \ldots, A_n$ on inputs $y_1, \ldots, y_n$, is α, where the probability is over the random choices of the probabilistic algorithms involved.

4.2.2 Cryptographic Primitives

The following definition is due to Goldwasser, Micali, and Rivest [50], and has become the standard definition of security for signature schemes. Schemes that satisfy it are also known as signature schemes secure against *adaptive chosen-message attack*.

Definition 4.2.1 (Signature scheme)**.** *The triplet of probabilistic polynomial-time algorithms* $(G(\cdot), \mathsf{Sign}_{(\cdot)}(\cdot), \mathsf{Verify}_{(\cdot)}(\cdot,\cdot))$*, where* G *is the key generation algorithm producing a pair* (PK, SK) *of public and secret keys on input a security parameter* k*,* Sign *the signature algorithm, and* Verify *the verification algorithm, constitute a digital signature scheme for a family (indexed by the public key* PK*) of message spaces* $\mathcal{M}_{(\cdot)}$ *if the following two hold:*

Correctness *If a message* m *is in the message space for a given public key* PK*, and* SK *is the corresponding secret key, then the output of* $\mathsf{Sign}_{SK}(m)$ *will always be accepted by the verification algorithm* Verify_{PK}*. More formally, for all values* m *and* k*:*

$$\Pr[(PK, SK) \leftarrow G(1^k); \sigma \leftarrow \mathsf{Sign}_{SK}(m) \ : \ m \leftarrow \mathcal{M}_{PK} \wedge \neg \mathsf{Verify}_{PK}(m, \sigma)] = 0.$$

Security *Even if an adversary has oracle access to the signing algorithm that provides signatures on messages of the adversary's choice, the adversary cannot create a valid*

signature on a message not explicitly queried. More formally, for all families of probabilistic polynomial-time oracle Turing machines $\{A_k^{(\cdot)}\}$*, there exists a negligible function*[1] $\nu(k)$ *such that*

$$\Pr[(PK, SK) \leftarrow G(1^k); (Q, m, \sigma) \leftarrow A_k^{\mathsf{Sign}_{SK}(\cdot)}(1^k) : \mathsf{Verify}_{PK}(m, \sigma) = 1 \wedge \neg(\exists \sigma' \mid (m, \sigma') \in Q)] \quad = \quad \nu(k).$$

For completeness, we give a standard definition of a family of collision-resistant hash functions.

Definition 4.2.2 (Collision-resistant Hash Function)**.** *Let* $\mathcal{H}$ *be a probabilistic polynomial-time algorithm that, on input* 1^k*, outputs an algorithm* $H \ : \ \{0,1\}^* \mapsto \{0,1\}^k$*. Then* $\mathcal{H}$ *defines a family of collision-resistant hash functions if:*

Efficiency *For all* $H \in \mathcal{H}(1^k)$*, for all* $x \in \{0,1\}^*$*, it takes polynomial time in* $k + |x|$ *to compute* $H(x)$*.*

Collision-resistance *For all families of probabilistic polynomial-time Turing machines* $\{A_k\}$*, there exists a negligible function* $\nu(k)$ *such that*

$$\Pr[H \leftarrow \mathcal{H}(1^k); (x_1, x_2) \leftarrow A_k(H) \ : \ x_1 \neq x_2 \wedge H(x_1) = H(x_2)] = \nu(k).$$

4.2.3 Error-Correcting Codes

Error-correcting codes allow recovering a message that is transmitted over a noisy channel. Let $q \geq 2$ be the size of alphabet $[q] = \{1, 2, ..., q\}$. An *error-correcting code* $[n, k]_q$, $k < n$, is a function[2] $C : [q]^k \rightarrow [q]^n$ that takes as input a k-length message x over $[q]$ and outputs a longer n-length codeword $C(x)$ over the same alphabet. That is, an error-correcting code processes k characters (symbols) in $[q]$ and adds redundancy to form n characters. If only redundancy is added by a code C such that the k first symbols of $C(x)$ form word x, then the code is called *systematic*.

The processing and the added redundancy help correcting up to e errors of the codeword $C(x)$; that is, given a received word $y \in [q]^n$ such that y and $C(x)$ differ in at most e characters of the alphabet, one can unambiguously correct (decode) y to $C(x)$. The value

[1] A function $\nu : \mathbb{N} \rightarrow \mathbb{R}$ is negligible if for every positive polynomial $p(\cdot)$ and for sufficiently large k, $\nu(k) < \frac{1}{p(k)}$.

[2] There are more than one possible definitions. Here we choose to define an error-correcting code as a function.

e depends on the exact choice of the code C. In particular, for *unambiguous* decoding, e is always bounded from above by $d/2$, i.e., $e < d/2$, where d is the *diameter* of the code, defined as follows. The *distance* of two codewords of C is the number of positions where their symbols differ and the distance of code C is the minimum distance of two codewords of C, measured over all pairs of codewords.

List-decoding allows to correct a number of errors that is beyond the bound above. Handling even more errors comes at a price, though. In fact, one can *ambiguously* correct beyond this bound. That is, given that $C(x)$ is received as word $y \in [q]^n$ and $C(x)$ and y differ in at most e positions with $e \geq d/2$, list-decoding provides a list of candidate initial messages in $[q]^k$, such that x belongs in this list. In general, as e grows, the size of the output list grows as well and, typically, we put an upper bound on e so that list-decoding is efficient (i.e., is performed in polynomial time) and the list size is reasonable to work with.

Reed-Solomon codes [125] are a family of codes that are based on properties of univariate polynomials over finite fields. An $[n, k+1]_q$ Reed-Solomon code, $k < n \leq q$, has diameter $d = n - k$ and, as long as the number of errors e is at most $(d-1)/2 = (n-k-1)/2$, one can unambiguously decode in quadratic time [145]. If $e > (n-k-1)/2$, then list-decoding is considered to be feasible [49] (i.e., it can be performed in polynomial time) as long as

$$e \leq n - \sqrt{nk}.$$

We next define the $[n, k+1]_q$ Reed-Solomon code, where n, k and q are positive integers and parameters of the code, with $k < n \leq q$,[3] and where list-decoding is considered. We present here a slightly modified definition of Reed-Solomon codes than the one commonly used in the literature, so that, by definition, Reed-Solomon codes considered in this chapter are systematic. This property is not necessary for the correctness of our scheme, but offers an extra desired property in our construction, discussed in Section 4.7 and requires no extra computational effort at the encoder.

Definition 4.2.3 (Reed-Solomon code)**.** *A $[n, k+1]_q$ Reed-Solomon code consists of the following components:*

Alphabet *The alphabet is a finite field $\mathbb{F}_q$ of size $q \geq n$, with q being a prime power.*

Encoder *The code is a function $C : \mathbb{F}_q^{k+1} \to \mathbb{F}_q^n$, where $n > k$. In more detail, the encoder:*

1. *takes as input the parameters n and k, and $k+1$ points (i, y_i), $i \in \mathbb{F}_q$, $y_i \in \mathbb{F}_q$, $1 \leq i \leq k+1$,*

[3] For simplicity, in this definition we require that $k < n$ rather than $k+1 < n$, allowing thus the extreme case, where, for $k = n-1$, the code adds no redundancy at all.

2. *finds a unique univariate polynomial* $p \in \mathbb{F}_q[x]$ *over elements of* $\mathbb{F}_q$ *and of degree at most* k*, such that* $p(i) = y_i$*,* $1 \leq i \leq k+1$*, and*
3. *outputs points* $(i, p(i))$*, for* $1 \leq i \leq n$*. The ratio* $\frac{k}{n} < 1$ *is called the* rate *of the code.*[4]

We have that C *is a systematic code.*

Decoder *Let* $0 < \epsilon < 1$ *be a parameter that controls the performance of the decoder. The decoder takes as input parameters* n *and* k*, the maximum number of errors* e *that may occur, and* n *points* (x_i, y_i)*,* $1 \leq i \leq n$*, and list-decodes, i.e., it outputs a list of all univariate polynomials* $p \in \mathbb{F}_q[x]$ *of degree at most* k *such that* $y_i \neq p(x_i)$ *for less than* e *values of* i*,* $1 \leq i \leq n$*.*

For an $[n, k+1]_q$ Reed-Solomon code, we use a decoder that runs in polynomial time and is due to Guruswami and Sudan [62, 63]. The maximum number of errors that can be tolerated by the list-decoding procedure matches the theoretical bound of $n - \sqrt{kn}$. However, the decoding algorithm is prohibitively expensive (e.g., $O(n^{12})$) when the exact bound is met. By reducing this upper bound of the maximum number of errors to

$$n - \sqrt{(1+\epsilon)kn}, \quad \epsilon \in (0, 1),$$

a quadratic decoding algorithm exists [63]. We refer to this decoder as GS-Decoder. For this decoder, parameter ϵ controls how far away from the bound of $n - \sqrt{kn}$ list-decoding operates (in the worst case). We will use the following result.

Theorem 4.2.1 (Guruswami-Sudan)**.** *Consider a* $[n, k+1]_q$ *Reed-Solomon code. For any* $\epsilon \in (0, 1)$*, given* n *points with at most* $e = n - \sqrt{(1+\epsilon)kn}$ *errors, GS-Decoder outputs a list of size* $O(\epsilon^{-1}\sqrt{n/k})$ *in* $O(n^2\epsilon^{-5}\log^2 q \log^{O(1)} \log q)$ *time, performing* $O(n^2\epsilon^{-5}\log q)$ *field operations.*

We view ϵ as a parameter of the GS-Decoder, $0 < \epsilon < 1$, and we also make use of the notation $\mathsf{GSDecode}_\epsilon(n, k, e, \{(x_i, y_i) | 1 \leq i \leq n\})$ to denote that GS-Decoder runs with parameter ϵ having as input parameters the integers n, k, e and the points (x_i, y_i), $1 \leq i \leq n$.

In practice, codes with constant expansion are used, where $k = \rho n$ for some constant $\rho < 1$, with ρ being the rate of the code in use. In particular, our construction makes use of a $[n, \rho n + 1]_q$ Reed-Solomon code, with $\rho < 1$, over some large alphabet of size $q = 2^c$ for

[4]For convenience, we note that in fact the rate of an $[n, k+1]_q$ code is $\frac{k+1}{n}$, but for simplicity we adopt the ratio $\frac{k}{n}$ as being the rate.

some constant c. From Theorem 4.2.1 we have the following, where by $\tilde{O}(\cdot)$ we denote that some logarithmic factors are omitted.

Corollary 4.2.1. *For any $[n, \rho n + 1]_q$ Reed-Solomon code, for any constants $\epsilon \in (0, 1)$ and $\rho < 1$ such that $\sqrt{(1+\epsilon)\rho} \leq 1$, on an input with at most $e = (1 - \sqrt{(1+\epsilon)\rho})n$ errors, GS-Decoder outputs a list of $O(1)$ size in $\tilde{O}(n^2)$ time, performing $\tilde{O}(n^2)$ field operations.*

Proof. It follows directly from Theorem 4.2.1 for $k = \rho n$, $\rho < 1$ and using the $\tilde{O}(\cdot)$ notation to hide constants and logarithmic on q factors. We need $\sqrt{(1+\epsilon)\rho} \leq 1$ such that $e \geq 0$. □

GS-Decoder is based on an algorithm that solves the *polynomial reconstruction* problem: given k, t, and n points $\{(x_i, y_i), 1 \leq i \leq n\}$, where $x_i, y_i \in \mathbb{F}_q$, find a list that contains all univariate polynomials $p \in \mathbb{F}_q[x]$ of degree at most k such that $y_i = p(x_i)$ for at least t values of i, $1 \leq i \leq n$. Parameter t is usually referred as *agreement*. Polynomial reconstruction and Reed-Solomon list-decoding are equivalent problems [62]. In particular, the following corollary is equivalent to Corollary 4.2.1.

Corollary 4.2.2. *For any constants $\epsilon \in (0, 1)$ and $\rho < 1$ such that $\sqrt{(1+\epsilon)\rho} \leq 1$, polynomial reconstruction on input ρn, t, and n points in $\mathbb{F}_q \times \mathbb{F}_q$ can be solved in $\tilde{O}(n^2)$ time, provided $t \geq \sqrt{(1+\epsilon)\rho}n$, where $\tilde{O}(n^2)$ field operations are performed and the output list has $O(1)$ size.*

Proof. It follows immediately from Corollary 4.2.1 by considering the equivalence between the two problems. The upper bound e of the number of errors guarantees agreement t of at least $n - e = \sqrt{(1+\epsilon)\rho}n$. We need $\sqrt{(1+\epsilon)\rho} \leq 1$ in order to be consistent with the requirement that $t \leq n$. □

4.3 Network Model and Multicast Authentication Framework

Considering data transmission in a multicast setting, a *sender*, the source of the data, transmits a data stream over an underlying "best-effort" network. Data packets are received by a large set of *receivers*. Without loss of generality, we focus our attention to one such receiver, such that the two honest parties of the authentication protocol are the sender and the receiver. No guarantees about the delivery of the packets exist in general. Furthermore, the network is an *adversary* of great—yet not unlimited—power, acting in the bounded computational model. In our model, packets may be *adversarially lost, altered, delayed,*

or injected. However, this adversary is not given complete freedom—if it were, then no messages (data) would ever get delivered, and so our task would be hopeless.

Conventionally and without loss of generality, we consider data streams consisting of n packets, that is, at the sender, the data for transmission is arranged in groups of size n. Each group of n packets is identified by a—unique per distinct group—*group identification tag GID*. That is, packets of a group are marked with the corresponding *GID*. Note that the existence of tag *GID* adds no new assumption about the transmitted stream. It corresponds to a means by which the packets can be grouped together and in practice it can be provided by any network-layer transmission protocol in use. In our framework, the *GID* is used as an abstract quantity of constant size; in practice, it is a string of some small constant size, e.g., the size of a hash value (20 bytes for SHA-1).

4.3.1 The (α, β)-Network Model

We model the network as an *adversarial* entity, i.e., an entity that can *simultaneously* inflict *any possible type* of attack to the transmitted data stream. The repertoire of attacks consists of packet *losses*, *injections*, *alterations* and *rearrangements*. These modifications of the data stream are adversarially chosen so that the adversary can cause the loss of any selected packets. The ability to tolerate packet losses has been widely considered an important property of multicast authentication schemes [51, 100, 111, 116, 136, 148]. However, only a few previous schemes [68, 111, 148] tolerate *adversarial losses*, i.e., the capability by the adversary to choose which packets are dropped and which survive. The adversarial loss model is the strongest, and also the most realistic one, since it makes the least assumptions on how the traffic is routed.

Also, the adversary can inject packets of random or malicious structure into the stream. This type of network failure has not been studied as widely in the context of multicast authentication. In contrast, we develop robust techniques for dealing with it. In their recent paper [68], Karlof *et al.*, in studying the *pollution attack* by an adversary when an erasure-code-based authentication scheme is in use, they in essence consider packets injections as well.

Finally the adversary can arbitrarily modify, delay or rearrange packets. Note that changing a packet corresponds to destroying (losing) it and injecting a new packet.

An adversarial network modeled with the above capabilities in terms of how the adversary is acting is what we call a *fully adversarial network*.

Definition 4.3.1 (Fully adversarial network)**.** *A* fully adversarial network *is a network*

that is used for the transmission of a data stream and is controlled *by a computationally bounded (i.e., probabilistic polynomial-time) adversary. In particular, the adversary can:*

- *cause packets of* her choice *to be lost;*
- *inject packets (either* random *ones or with a* specific malicious structure*); and*
- arbitrarily *alter, delay or rearrange packets.*

It is realistic to assume that even if an adversary controls part of the network, there are still some honest routers and at least a fraction of the data packets goes through them. Thus, we expect some *reliability* from the network. Namely, the network will faithfully deliver at least a constant fraction, α, of all the packets of a given stream. This assumption is also justified by the fact that if fewer than a constant fraction of the packets survive, then it is unlikely that meaningful information can be extracted from the surviving packets. Also, in modeling the ability of the adversary to maliciously inject invalid packets, we take the following into consideration: if the adversary injects packets at too high a rate, this will result in a denial-of-service attack. In this case, the receiver's primary concern is unlikely to be authentication. Thus, we assume that authentication is useful when the stream is expanded by no more than a constant factor β through adversarial packet injections.

Our two assumptions about the power of the adversary to modify a stream of n packets transmitted by the sender are expressed by two *parameters* of the adversarial network: the *survival rate* α and the *flood rate* β. In this chapter, both rates are considered to be constants. In a network with survival rate α, $0 < \alpha \leq 1$, if a stream of n packets is sent, at least αn packets of this stream will arrive at the receiver intact. In a network with flood rate β, $\beta \geq 1$, if a stream of n packets is sent, then the received stream will have at most βn packets.

Definition 4.3.2 (Network parameters)**.** *Consider an adversarial network through which a stream of n packets is transmitted by the sender. With respect to any particular receiver:*

- *The* survival rate α, $0 < \alpha \leq 1$, *is the minimum fraction of the packets that are guaranteed to reach the receiver unmodified. Namely, at least αn packets in the received stream are valid.*
- *The* flood rate β, $\beta \geq 1$, *indicates the maximum factor by which the size of the stream that reaches the receiver may exceed the size of the transmitted stream. Namely, at most βn packets are in the received stream.*

We claim that the survival and flood rates of a network are reasonable parameters that better characterize the adversary's ability to modify the transmitted stream. In particular, the network's reliability, expressed by means of α and β, is not an assumption that affects the generality or the strength of our model; in contrast, it successfully emphasizes some intrinsic characteristics of the authentication problem we study. Indeed, the extreme cases where too few packets survive or too many packets are injected are both degenerate cases of the multicast authentication problem: at the receiver, no need for authentication really exists when $\alpha \to 0$ or $\beta \to \infty$. At the same time, our scheme presented in Section 4.4 is *parameterized* by the two rates α and β; that is, it operates for *any values* of these two network parameters.

A network with the above characteristics in terms of adversarial behavior and reliability is what we call an (α, β)-*network* and is the basis for our multicast authentication framework. Although our discussion focuses on one particular receiver, for generality and completeness, in the following definition we require that the network provides the same level of reliability (expressed by rates α and β) to any of the receivers[5].

Definition 4.3.3 ((α, β)-network)**.** *An* (α, β)-network *is a fully adversarial network with survival rate* α *and flood rate* β *with respect to any receiver.*

4.3.2 Authentication Framework

We describe a new multicast authentication framework that is based on the (α, β)-network model. Our definition of a *multicast authentication scheme* essentially mimics the classical definition of security for signatures [50]. This is not surprising since in [11] it is shown that the two problems are equivalent. A signature scheme consists of key generation, signature, and verification algorithms (see Definition 4.2.1). Similarly, we have key generation, authentication, and decoding algorithms, specified below. Working in the public-key model, the key generation algorithm is run in advance to produce the private and public keys used by the involved parties, the sender and the receiver. The other two algorithms, authenticator *Auth* and decoder Decode, are executed by the sender and the receiver respectively. The sender runs *Auth* to process data packets and create the authenticated packets. The receiver runs Decode to decode the received packets and recognize the valid ones.

Key generation The key generation algorithm KeyGen is a probabilistic polynomial-time algorithm that takes as input the security parameter 1^k and outputs the key pair

[5]This does not, of course, mean that the same data packets, i.e., the same stream, reaches all the receivers.

(PK, SK). We write $(PK, SK) \leftarrow \mathsf{KeyGen}(1^k)$. We assume that the sender knows both the public key PK and the secret key SK and that the receiver knows the public key PK.

Authenticator The authenticator algorithm *Auth* takes as input:

- (SK, PK): the secret key and the public key.
- *GID*: the group identification tag of the data stream.
- n: the size of the data stream, i.e., the number of packets that need to be authenticated.
- α: the survival rate that determines what fraction of the packets are guaranteed to reach the receiver intact, $0 < \alpha \leq 1$.
- β: the flood rate that determines an upper bound of the packets that reach the receiver; namely, the received stream consists of at most βn packets that claim to belong to a given *GID*, $\beta \geq 1$.
- $DP = \{p_1, \ldots, p_n\}$: the *data packets*, i.e., the data stream that needs to be authenticated.

The output of the authenticator algorithm is the set AP of *authenticated packets*, with $AP = \{a_1, \ldots, a_n\}$. We write: $AP \leftarrow Auth(SK, PK, GID, n, \alpha, \beta, DP)$.

Decoder The decoder algorithm Decode takes as input:

- PK: the public key.
- *GID*: the group identification tag of the data stream.
- n: the number of the original data packets.
- α: the survival rate.
- β: the flood rate.
- $RP = \{r_1, \ldots, r_m\}$: the received packets.

The decoder either *rejects* the input (when less than αn of the received packets are valid, or more than $(\beta - \alpha)n$ packets are injected by the adversary[6]), or produces the *output packets* $OP = \{p'_1, \ldots, p'_n\}$. Some of these packets may be empty—an empty output packet is denoted by $\varnothing$, and corresponds to the event that the decoder

[6]Note that both, of course, cannot be true for an (α, β)-network.

did not receive the corresponding authenticated packet. We write: $\{OP, \mathsf{reject}\} \leftarrow \mathsf{Decode}(PK, GID, n, \alpha, \beta, RP)$.

A signature scheme has two requirements: correctness and security. We have similar requirements for a multicast authentication scheme.

A multicast authentication scheme is (α, β)-*correct* if, whenever at least αn correct authenticated packets are received among βn total packets, all and only the valid received packets will be decoded correctly, i.e., the corresponding data packets will be among the output packets.

A multicast authentication scheme is *secure* if, even if the adversary is allowed to query the authenticator on any number of chosen inputs, the adversary cannot make the decoder output a non-authenticated set of packets.

Definition 4.3.4 (Multicast Authentication Scheme)**.** *Probabilistic polynomial-time algorithms* (KeyGen, *Auth,* Decode) *constitute an* (α, β)*-correct and secure multicast authentication scheme if no probabilistic polynomial-time adversary* $\mathcal{A}$ *can win non-negligibly often in the following game:*

1. *A key pair is generated:*
$$(PK, SK) \leftarrow \mathsf{KeyGen}(1^k).$$
2. *The adversary* $\mathcal{A}$ *is given:*
   - *The public key* PK *as input.*
   - *Oracle access to the authenticator, i.e., for* $1 \leq i \leq poly(k)$*, where* $poly(\cdot)$ *is a polynomial, the adversary can specify the values* $(GID_i, n_i, \alpha_i, \beta_i, DP_i)$ *and obtain* $AP_i \leftarrow Auth(SK, PK, GID_i, n_i, \alpha_i, \beta_i, DP_i)$*. However, the adversary cannot issue more than one query with the same group identification tag. That is to say, for all* $i \neq j$*,* $GID_i \neq GID_j$*.*
3. *At the end,* $\mathcal{A}$ *outputs a group identification tag, GID, the values* n*,* α *and* β*, and a set of packets,* RP*.*

The adversary wins *the game if one of the following violations occurs:*

Violation of the (α, β)***-correctness property:*** *The adversary did manage to construct* RP *in such a way that even though it contains* $\alpha_i n_i$ *packets of some authenticated packet set* AP_i *for group identification tag* $GID_i = GID$*, the decoder still failed at identifying all the correct packets. Namely, the adversary wins if* all *of the following hold:*

- *For some i, the adversary's query i contained $GID_i = GID$, $n_i = n$, $\alpha_i = \alpha$ and $\beta_i = \beta$. Let $DP_i = \{p_1, \ldots, p_n\} = DP$ be the data packets associated with that query, and let $AP_i = \{a_1, \ldots, a_n\} = AP$ be the response of the authenticator.*
- *At least αn of the authenticated packets $(a_1, \ldots, a_n)$ are included in the received packets RP, i.e., $|RP \cap AP| \geq \alpha n$.*
- *The number of received packets is at most βn, i.e., $|RP| \leq \beta n$.*
- *For some $1 \leq j \leq n$, p_j is the j'th packet in the original set of data packets DP, such that the corresponding authenticated packet a_j was received, i.e., $a_j \in RP \cap AP$, and yet was not decoded correctly. Namely, let $(p'_1, \ldots, p'_n) \leftarrow \mathsf{Decode}(PK, GID, n, \alpha, \beta, RP)$. For p_j it holds that $p_j \neq p'_j$.*

Violation of the security property: *The adversary did manage to construct RP in such a way that the decoder will output packets $OP = \{p'_1, \ldots, p'_n\}$ that were never authenticated by the authenticator algorithm for the group identification tag GID. More precisely, the adversary wins if* one *of the following happens:*

- *The authenticator was never queried with group identification tag GID and the size n, and yet the decoder algorithm does not reject. In particular, it holds that $\mathsf{Decode}(PK, GID, n, \alpha, \beta, RP) \neq \mathsf{reject}$ but $\mathsf{Decode}(PK, GID, n, \alpha, \beta, RP) = OP$.*
- *The authenticator was queried with the group identification tag GID, the values n, α and β, and the data packets $DP = \{p_1, \ldots, p_n\}$. However, the decoder algorithm does not reject and some output packet $p'_j \neq \varnothing$ is different from the corresponding data packet p_j, where $OP = \{p'_1, \ldots, p'_n\}$.*

4.4 Multicast Authentication Scheme AuthECC

In this section we describe a multicast authentication scheme $(\mathsf{KeyGen}, Auth, \mathsf{Decode})$ that meets the definitions of the previous section. In the sequel, we denote with ϵ, $0 < \epsilon < 1$, the *tolerance parameter* of the decoder, which controls a trade-off between the error-tolerance ability of the decoder and its performance. Both the authenticator and the decoder know and use the value of this parameter. Recall that this parameter expresses how far away from the ultimate bound for efficient list-decoding the encoding is performed. The higher the ϵ the further away from the bound the encoding is performed, thus, the higher the communication overhead is, but the faster (up to constant factors) the list decoder operates. Similarly, values of ϵ that are closer to zero reduce the communication overhead at the cost

of increasing (by constant factors) the decoding time. We will discuss in detail this trade-off when we will perform the efficiency analysis. By $\circ$, we denote concatenation and by $\varnothing$ we appropriately denote either a packet that is empty or the empty string. We also often omit the floor and ceiling notation in order to avoid notational overload.

4.4.1 Key Generation and Authenticator

On input the security parameter 1^k, a signature scheme $(G(\cdot), \mathsf{Sign}_{(\cdot)}(\cdot), \mathsf{Verify}_{(\cdot)}(\cdot,\cdot))$ and a family $\mathcal{H}$ of collision-resistant hash functions (see Definitions 4.2.1 and 4.2.2), the key-generation algorithm KeyGen operates simply by initializing a signature scheme and a collision-resistant function using the security parameter. If $(PK_s, SK_s) \leftarrow G(1^k)$ and $H \leftarrow \mathcal{H}(1^k)$, KeyGen sets $PK = (PK_s, H)$ and $SK = SK_s$, i.e., $((PK_s, H), SK_s) \leftarrow \mathsf{KeyGen}(1^k)$ (see Algorithm 1).

Algorithm 1 Key Generation KeyGen

Input: Security parameter 1^k, signature scheme $(G(\cdot), \mathsf{Sign}_{(\cdot)}(\cdot), \mathsf{Verify}_{(\cdot)}(\cdot,\cdot))$ and collision-resistant hash-function family $\mathcal{H}$.

Output: Secret key SK and public key PK.

Algorithm:

1. Let $(PK_s, SK_s) \leftarrow G(1^k)$ and $H \leftarrow \mathcal{H}(1^k)$.
2. Set $PK = (PK_s, H)$ and $SK = SK_s$.

We now describe the authenticator *Auth* that uses the tolerance parameter ϵ of the decoder (see Algorithm 2). The idea is as follows. The data packets are each hashed using the collision-resistant hash function and these n packet hashes $h_1, \ldots, h_n$ (along with the group identifier) are digitally signed to produce signature σ. String $S = h_1 \circ \ldots \circ h_n \circ \sigma$ is called the *authentication information*. We want to guarantee that, even if only an α fraction of the packets survive, and a large number of packets $(\beta - \alpha)n$ are injected, the receiver still gets (is able to reconstruct) all the authentication information. To that end, we encode S using an $[n, \rho n + 1]_q$ Reed-Solomon code in a manner that is tolerant to packet losses and insertions, subject to the network parameters. The encoded authentication information is then appropriately dispersed and appended in the data packets to form the authenticated packets.

In Algorithm 2, we note that q is in fact a function of n, α and β, thus it does not need be transmitted to the receiver (observe that $|S|$ is a function of n). Also, we assume

Algorithm 2 Authenticator *Auth*

Input: Secret key SK, public key PK, group identification tag GID, data-stream size n, parameters α and β of the network and data packets $DP = \{p_1, \ldots, p_n\}$.

Output: Authenticated packets $AP = \{a_1, \ldots, a_n\}$.

Algorithm:

1. For $1 \leq i \leq n$, compute the hash value $h_i = H(p_i)$. The concatenation of all the hash values, together with the value GID, is digitally signed:

$$\sigma \leftarrow \mathsf{Sign}_{SK}(GID \circ h_1 \circ \ldots \circ h_n).$$

2. Let $[n, \rho n + 1]_q$ be a Reed-Solomon error-correcting code and set its rate to be

$$\rho = \frac{\alpha^2}{(1+\epsilon)\beta},$$

where α and β are the survival and flood rates of the network, and ϵ is the tolerance parameter of the decoder. (Note that since $\alpha \leq 1$, $\beta \geq 1$ and $0 < \epsilon < 1$, we have $\rho < 1$.)

3. Split S into $\rho n + 1$ substrings of size $\left\lceil \frac{|S|}{\rho n+1} \right\rceil$, where each substring is viewed as a value of $\mathbb{F}_q$, with $q = 2^{\left\lceil \frac{|S|}{\rho n+1} \right\rceil}$. If S is not an exact multiple of $\rho n + 1$, pad S with ℓ 0's, such that $|S \circ 0^\ell| \mod \rho n + 1 \equiv 0$.
4. Treat the resulting set of $\rho n + 1$ field elements as an input to the Reed-Solomon encoder (see Definition 4.2.3). Compute the corresponding codeword $C(S)$ using the $[n, \rho n + 1]_q$ Reed-Solomon code of Step 2. $C(S)$ consists of n elements of $\mathbb{F}_q$, denoted as $(s_1, \ldots, s_n)$.
5. Let $AP = \{a_1, \ldots, a_n\}$, where for $1 \leq i \leq n$, we have $a_i = GID \circ i \circ p_i \circ s_i$.

that the value of ϵ is known also to the encoder; thus, in fact $C(S) = C_\epsilon(S)$. Finally, we implicitly assume that the size of the problem n along with the parameters of the network α and β are such that $\rho n + 1 < n$, or equivalently, $\frac{\alpha^2}{(1+\epsilon)\beta} + \frac{1}{n} < 1$, so that the used $[n, \rho n + 1]_q$ Reed-Solomon code does not degenerate. This last technical requirement is easily satisfied as n and β get larger and α gets smaller.

4.4.2 Decoder

Our decoder Decode uses a modification of the GS-Decoder (see Definition 4.2.3 and Theorem 4.2.1) as a subroutine. The standard GS-Decoder expects to receive, as input, n pairs (x_i, y_i), and outputs a list L of all the polynomials of degree at most k such that

every $p \in L$ has the property that for at least $\sqrt{(1+\epsilon)kn}$ of the i's, $p(x_i) = y_i$. We write $L \leftarrow \mathsf{GSDecode}_\epsilon(n, k, \sqrt{(1+\epsilon)kn}, \{(x_i, y_i) | 1 \leq i \leq n\})$. The modified decoder is specified by parameters that are slightly different: it takes as input up to βn points (x_i, y_i) and finds a list of candidate inputs (set of points) that can be encoded by polynomials of degree at most ρn (with $\rho = \frac{\alpha^2}{(1+\epsilon)\beta}$) such that each polynomial agrees with at least αn of the input points (see Corollary 4.2.2).

What is important is that the modified GS-Decoder operates even in the presence of an *adversarially chosen* set of (x_i, y_i) pairs. In other words, the modified decoder corresponds to the alphabet and encoder of the Reed-Solomon code described in Definition 4.2.3 (the encoder is used in the *Auth* algorithm above), but now, the set of points (x_i, y_i) that constitute the input of the decoder is in principle different than the set of *valid* output points of the encoder; this corresponds to the various attacks by the adversary. For instance, this set may not include some of the original points, may be larger since some new points are added and may even contain points that are *vertically aligned*, i.e., some of the x_i's are not distinct. The modified decoder is obtained by adapting the original GS-Decoder, as follows (see Algorithm 3).

Algorithm 3 Modified GS-Decoder $\mathsf{MGSDecoder}_\epsilon$

Input: n, α, β, and m points (x_i, y_i), $1 \leq i \leq m$.

Output: List of all computed candidates $\{c_1, \ldots, c_\ell\}$ or reject.

Algorithm:

1. If $m > \beta n$, reject.
2. Else, if there are fewer than αn distinct values of x_i, reject.
3. Else, run the GS-Decoder, that is, let $L \leftarrow \mathsf{GSDecode}_\epsilon(m, \rho n, \alpha n, \{(x_i, y_i) | 1 \leq i \leq m\})$, where $\rho = \frac{\alpha^2}{(1+\epsilon)\beta}$. If L is empty, reject.
4. Process $L = \{Q_1(i), \ldots, Q_\ell(i)\}$ as follows: for each $Q_j(i) \in L$, $1 \leq j \leq \ell$, evaluate $Q_j(i)$ for $1 \leq i \leq \rho n + 1$ and let the string $Q_j(1) \circ Q_j(2) \circ \cdots \circ Q_j(\rho n + 1)$ be candidate c_j.

For this decoder, operating on input points subject to constraints that are in accordance with our (α, β)-network, we have the following.

Lemma 4.4.1. *When at least αn out of at most βn input points are valid, $\mathsf{MGSDecoder}_\epsilon$ does not reject, runs in time $\tilde{O}(n^2)$, where $\tilde{O}(n^2)$ field operations are involved, and outputs the constant-size list of all candidate inputs that are consistent with αn of the received points.*

Proof. $\mathsf{MGSDecoder}_\epsilon$ does not reject in Steps 1 and 2 of the algorithm. All claims follow considering Step 3, Theorem 4.2.1, Corollary 4.2.2 and the fact that GS-Decoder operates even when the x_i's are not distinct (see Guruswami and Sudan [63]). Corollary 4.2.2 holds, since, if $m = \gamma n$, $\alpha \leq \gamma \leq \beta$, then $\rho n = \frac{\rho}{\gamma} m$, thus, we consider the polynomial reconstruction problem on inputs $\frac{\rho}{\gamma} m$, t, m points and for $\rho = \frac{\alpha^2}{(1+\epsilon)\beta}$ we have that

$$t \geq \alpha n \geq \sqrt{\frac{\gamma}{\beta}} \alpha n = \sqrt{(1+\epsilon)\rho n m} = \sqrt{(1+\epsilon)\frac{\rho}{\gamma}} m,$$

as needed. Of course, $\tilde{O}(m^2) = \tilde{O}(n^2)$ for $m = \gamma n$ and γ a constant. Finally, we have that $\sqrt{(1+\epsilon)\frac{\rho}{\gamma}} = \sqrt{\frac{\alpha^2}{\beta\gamma}} \leq 1$ as required. Note that the correct answer (candidate input) is guaranteed to be contained in the output list, since it is a polynomial of degree at most ρn that is consistent with αn points. Thus, $\mathsf{MGSDecoder}_\epsilon$ does not reject in Steps 3 either. □

Now, we are ready to describe our decoder. The idea is simple: we resist every attack by the adversary by treating injected, altered and lost packets as errors, essentially, in a *crypto-enhanced* list-decoder (see Algorithm 4). What matters is only the achieved *agreement*, i.e., the valid packets. Note that the decoder is parameterized by the tolerance parameter ϵ (that is, Decode is in fact Decode_ϵ).

For this decoder we have the following.

Lemma 4.4.2. *When operating on a stream of packets encoded by authenticator Auth and transmitted through an (α, β)-network, algorithm* Decode *does not reject.*

Proof. From the properties of the (α, β)-network, the algorithm does not reject in Steps 3 and 5. Since at least αn packets are valid, from Lemma 4.4.1 we have that the correct codeword (authentication information) is among the candidates of the output list of Step 5. Thus, the corresponding signature verification in Step 6 excludes the rejection in Step 7. □

We postpone the analysis of the running time of the algorithms KeyGen, *Auth* and Decode of our scheme until the next section.

4.4.3 Correctness and Security Proofs

Let us show that the authentication scheme (KeyGen, *Auth*, Decode) described in subsections 4.4.1 and 4.4.2 satisfies Definition 4.3.4. Suppose that we have an adversary $\mathcal{A}$ who manages to break the (α, β)-correctness or security of our scheme with (non-negligible) probability $\pi(k)$. Then one of the following is true:

Algorithm 4 Decoder Decode

Input: Public key PK, group identification tag GID, n, parameters α and β and received packets $RP = \{r_1, \ldots, r_m\}$.

Output: $OP = \{p'_1, \ldots, p'_n\}$ or reject.

Algorithm:

1. View packets in RP as $r_i = GID_i \circ j_i \circ p_i \circ s_i$.
2. Discard all non-conforming packets, i.e., all packets for which $GID_i \neq GID$ or packets with $j_i \notin [1..n]$. Let $(r_1, \ldots, r_{m'})$ be the remaining packets in RP. Each of them is viewed as $r_i = GID \circ j_i \circ p_i \circ s_i$, such that $j_i \in [1..n]$.
3. If $m' < \alpha n$ or $m' > \beta n$, then reject.
4. For $1 \leq i \leq m'$, set $(x_i, y_i) = (j_i, s_i)$.
5. Run algorithm $\mathsf{MGSDecoder}_\epsilon$ with input parameters n, α, β and the m' points (x_i, y_i), $1 \leq i \leq m'$. If $\mathsf{MGSDecoder}_\epsilon$ rejects, reject; otherwise, obtain the candidate codewords $\{c_1, \ldots, c_\ell\}$.
6. For $1 \leq i \leq n$, set $h_i = \varnothing$. Let $j = 1$. While $j \leq \ell$:
   - Parse the codeword c_j as string $h_1^j \circ \ldots \circ h_n^j \circ \sigma$.
   - If $\mathsf{Verify}_{PK_s}(GID \circ h_1^j \circ \ldots \circ h_n^j, \sigma) = 1$, then set $h_i = h_i^j$ for $1 \leq i \leq n$ and break out of the loop; otherwise, increment j.
7. If $(h_1, \ldots, h_n) = (\varnothing, \ldots, \varnothing)$, reject. Else, compute the output packets OP as follows:
   - Initialize $OP = \{p'_1, \ldots, p'_n\}$: for each $1 \leq i \leq n$, set $p'_i = \varnothing$.
   - For $1 \leq i \leq m'$:
     - view r_i as $r_i = GID \circ j \circ p_j \circ s_j$, such that $j \in [1..n]$.
     - if $H(p_j) = h_j$, set $p'_j = p_j$.
8. Let $OP = \{p'_1, \ldots, p'_n\}$.

- With probability (at least) $\pi(k)/2$, the adversary $\mathcal{A}$ violates the (α, β) correctness property.
- With probability (at least) $\pi(k)/2$, the adversary $\mathcal{A}$ violates the security property.

Let us show that a non-negligible probability of either event contradicts the security properties of the underlying signature scheme and hash function.

Claim 4.5. *If a polynomial-time adversary $\mathcal{A}$ violates the (α, β)-correctness property of our scheme, then the underlying signature scheme is not secure, or the underlying hash function is not collision-resistant.*

Proof. Let us prove the claim by exhibiting a reduction which transforms an attack that violates the correctness of our scheme, into an attack on the underlying signature scheme.

Reduction. The input to the reduction is the public key PK_s of the signature scheme. Our reduction is also given oracle access to the corresponding signer $\mathsf{Sign}_{SK_s}(= \mathsf{Sign}_{SK})$. The reduction sets up the public key $PK = (PK_s, H)$. Our reduction does not know the corresponding secret key. Our reduction invokes the adversary $\mathcal{A}$ on input PK. It now needs to be able to answer the adversary's queries to the authenticator *Auth*. In order to respond to a query $(GID_i, n_i, \alpha_i, \beta_i, DP_i)$, run the algorithm *Auth* with the following modification: in Step 1, at the beginning of the algorithm *Auth*, instead of computing the signature σ_i, obtain it by querying the signature oracle Sign_{SK}. Everything else is carried out as prescribed by the algorithm *Auth*.

It is clear that the view of the adversary in this reduction will be identical to the view that the adversary obtains in real life. Therefore, with the same probability as in real life, the adversary violates the correctness property. Namely, it outputs values *GID*, n, α, β and the set of received packets RP, such that all of the following hold:

1. $GID = GID_i$, $n = n_i$, $\alpha = \alpha_i$, and $\beta = \beta_i$ for some i. Let $DP_i = \{p_1, \ldots, p_n\}$ be the data packets associated with that query, and let AP be the response that we gave to the adversary. In particular, let σ_i be the signature associated with this query, that is, $\sigma_i \leftarrow \mathsf{Sign}_{SK}(GID \circ H(p_1) \circ \ldots \circ H(p_n))$.

2. $|RP \cap AP| \geq \alpha n$ and $|RP| \leq \beta n$.

3. For some j such that $r_j \in RP$, it is the case that $p_j \neq p'_j$, where $(p'_1, \ldots, p'_n) \leftarrow \mathsf{Decode}(PK, GID, n, \alpha, \beta, RP)$ and r_j is a packet that corresponds to packet $p_j \in DP_i$. (Note that from 2 and Lemma 4.4.2, it follows that algorithm Decode does not reject.)

Case 1. Suppose that $p'_j \neq \varnothing$. From 3, we get that either $H(p_j) \neq H(p'_j)$, or it is easy to find a collision to the hash function. By definition of Decode, if $r_j \in RP$ and $p'_j \neq \varnothing$, then, in Step 6, the algorithm Decode processes a candidate $c = h_1 \circ \ldots \circ h_n \circ \sigma$ such that $\mathsf{Verify}_{PK_s}(GID \circ h_1 \circ \ldots \circ h_n, \sigma) = 1$. We must argue that our signature oracle was never queried on input $(GID \circ h_1 \circ \ldots \circ h_n)$. Note that the only time it was queried with this *GID*, it was when we obtained σ_i on input $(GID \circ H(p_1) \circ \ldots \circ H(p_n))$. Moreover, in Step 7, Decode includes p'_j into OP if and only if $H(p'_j) = h_j$. Therefore, $h_j \neq H(p_j)$, and so our signature oracle was never queried with $(GID \circ h_1 \circ \ldots \circ h_n)$, and yet our adversary has

caused us to compute a signature σ such that $\mathsf{Verify}_{PK_s}(GID \circ h_1 \circ \ldots \circ h_n, \sigma) = 1$. Thus, the underlying signature scheme is insecure.

Case 2. So, suppose that $p'_j = \varnothing$. From 1 and 2, we know that αn of the original authenticated packets were received, among the total of βn packets. Then, by the properties of $\mathsf{MGSDecoder}_\epsilon$ (Lemma 4.4.1), Step 5 of the algorithm Decode includes the candidate value $c = H(p_1) \circ \ldots \circ H(p_n) \circ \sigma_i$. Then, by construction, it cannot be the case that in Step 7, $(h_1, \ldots, h_n) = (\varnothing, \ldots, \varnothing)$. If $(h_1, \ldots, h_n) = (H(p_1), \ldots, H(p_n))$, then by construction of Decode, if (as is the case according to 3) $r_j \in RP$, then $p'_j \neq \varnothing$, because p'_j is set to p_j when the packet r_j is considered in Step 7. Therefore, $(h_1, \ldots, h_n) \neq (H(p_1), \ldots, H(p_n))$, and yet $\mathsf{Verify}_{PK_s}(GID \circ h_1 \circ \ldots \circ h_n, \sigma) = 1$. But the only query with GID that we ever issued to the signer was for the message $(GID \circ H(p_1) \circ \ldots \circ H(p_n)) \neq (GID \circ h_1 \circ \ldots \circ h_n)$. Thus σ is a successful forgery. □

Claim 4.6. *If a polynomial-time adversary $\mathcal{A}$ violates the security property of our scheme, then the underlying signature scheme is not secure, or the underlying hash function is not collision-resistant.*

Proof. Let us set up the reduction in exactly the same way as in the proof of Claim 4.5. Again, the adversary's view in the reduction is the same as in real life. So, just as often as in real life, the adversary will violate the security property of our scheme, namely, one of the following will hold:

1. The authenticator was never queried with group identification tag GID and size n, and yet the decoder algorithm does not reject. That is, it holds that $\mathsf{reject} \neq \mathsf{Decode}(PK, GID, n, \alpha, \beta, RP) = OP$.

2. The authenticator was queried with the group identification tag GID, with the values n, α and β, and data packets $DP = \{p_1, \ldots, p_n\}$. However, the decoder algorithm does not reject and some output packet $p'_j \neq \varnothing$ is different from the corresponding data packet p_j, where $OP = \{p'_1, \ldots, p'_n\}$.

Suppose 1 holds. Then, from the description of the decoder, we know that the only way that it will produce some non-empty set of output packets is if, in Step 6, it sees a string c and a signature σ such that $\mathsf{Verify}_{PK_s}(GID \circ c, \sigma) = 1$. Since the signature oracle was never queried for this GID and n, σ is a successful forgery.

So, suppose that 2 holds. This is exactly the same situation as Case 1 of the proof of Claim 4.5, and we obtain either a successful forgery or a hash-function collision in the same manner. □

4.6.1 Authenticated Reed-Solomon Error-Correcting Code

Our authentication scheme consists of probabilistic algorithms (KeyGen, *Auth*, Decode) described in subsections 4.4.1 and 4.4.2. We note that essentially our scheme uses an *authenticated Reed-Solomon error-correcting code*. Using this term we refer to the fact that, by allowing the use of cryptographic primitives in the data that is encoded, list-decoding succeeds in operating (even) in the presence of *adversarial behavior* of the "transmission channel" (a network in our case). In general, depending on this adversarial behavior, our authenticated decoder either rejects or *correctly* and *securely* reconstructs the original codeword that was sent (authentication information in our case). For an (α, β)-network, Lemma 4.4.2 and Claim 4.5 guarantee that the correct reconstruction is produced and Claim 4.6 guarantees that the authenticated Reed-Solomon error-correcting code is secure. For this reason, we refer to our multicast authentication scheme as AuthECC.

Definition 4.6.1 (Authenticated ECC)**.** AuthECC *is the multicast authentication scheme consisting of the triplet of probabilistic algorithms* (KeyGen, *Auth*, Decode) *described above, which, in turn, realize an* authenticated Reed-Solomon error-correcting code*.*

We have, thus, proved the following result.

Theorem 4.6.1. *Multicast authentication scheme* AuthECC *is* (α, β)*-correct and secure for any* (α, β)*-network.*

We note that our authenticated error-correcting code AuthECC constitutes a general-purpose authentication tool for data streams or other unstructured data formats.

Finally, from the above result we immediately get the following corollary, which draws an interesting connection between coding theory and cryptography.

Corollary 4.6.1. *In the public-key model and the bounded computational model for communication channels, list-decoding can be transformed into unambiguous decoding.*

4.7 Analysis

We now analyze our scheme in terms of the various cost parameters. Recall that:

- α is the survival rate of the network, where $0 < \alpha \leq 1$;
- β is the flood rate of the network, where $\beta \geq 1$;
- ϵ is the tolerance parameter of the list-decoder, where $0 < \epsilon < 1$; and
- ρ is the rate of the encoder, where $\rho = \frac{\alpha^2}{(1+\epsilon)\beta}$ and $\rho < 1$.

We start by discussing the complexity of our authentication scheme AuthECC in terms of computational and communication costs and introduced delay, we then examine how our scheme can be tuned and extended and, finally, we finish the section by comparing our scheme with various previous proposed schemes. In the sequel, by h we denote the size of a hash value and by s the size of a digital signature.

Computational Cost. The sender and the receiver execute algorithms *Auth* and Decode, respectively. Both algorithms involve field operations (additions and multiplications) over finite field $\mathbb{F}_q$ of size $q = 2^{\left\lceil \frac{nh+s}{\rho n+1} \right\rceil} \simeq 2^{\frac{h}{\rho}}$. Both operations take $O\left(\frac{h}{\rho} \log^{O(1)} \frac{h}{\rho}\right)$ time [62]. Setting $N = \frac{h}{\rho}$, both operations take $O(N \log^{O(1)} N)$ time. Note that N is independent of n.

Authenticator: The cost to encode n packets is as follows. First, n hashes are computed and one signature operation is performed over the hashes. Then, a Reed-Solomon code is applied on the authentication information, which consists of a polynomial interpolation on $\rho n + 1$ positions and a polynomial evaluation in $n - \rho n - 1$ positions. These tasks require $O(n \log n)$ field operations or $O(n \log n \; N \log^{O(1)} N)$ time, since both polynomial evaluation and interpolation for polynomials of degree at most n can be solved using $O(n \log n)$ field operations (thus, Reed-Solomon encoding requires a quasi-linear $O(n \log n)$ number of field operations). Observe that the use of a systematic Reed-Solomon code adds no extra computational cost.

Decoder: From Theorem 4.2.1 and Lemma 4.4.1, we have that $O(\beta^2 n^2 N) = \tilde{O}(n^2)$ field operations are required and thus $O(\beta^2 n^2 N^2 \log^{O(1)} N) = \tilde{O}(n^2)$ time is needed for the decoder to run. Also, for each of $O(1)$ candidate polynomials, we perform a polynomial evaluation at $\rho n+1$ positions, thus $O(n \log n)$ field operations in $O(n \log n N \log^{O(1)} N)$ time, and one signature verification. In total, we have $O(n^2 N^2 \log^{O(1)} N) = \tilde{O}(n^2)$ processing time and $O(1)$ signature verifications. Finally, $O(n)$ hash values are computed.

Communication Cost. The size of the authentication information is $\frac{n}{\rho n+1}(s+hn)$. That is, we have *constant communication overhead* per packet

$$\frac{s+hn}{\rho n+1} < \frac{h}{\rho} + \frac{s}{\rho n} = \frac{h}{\rho} + o(1).$$

We see that $1/\rho$ hash values are included in each packet, with $\rho = \frac{\alpha^2}{(1+\epsilon)\beta} < 1$. The larger the value of ρ the smaller the authentication overhead.

Delay. As *delay*, we count the number of packets that the authenticator or decoder algorithm has to buffer. Of course, by definition, any authentication scheme according to our model needs to process n packets. However, delay is a cost parameter that is useful even in our model, since it captures the ability of the authenticator or the decoder to process packets in an *on-line* fashion. In our scheme the sender processes n packets and the receiver processes βn packets in the worst case. However, the receiver can invoke an decoding procedure only after $\rho n + 1$ or αn packets have been received.

In particular, the receiver can try to compute the authentication information exactly after $\rho n + 1$ packets are received: the used code is systematic and the first $\rho n + 1$ symbols of $E(S)$ equal S (where S is the authentication information). Of course, we need no packet loss to occur among these packets. If the correct polynomial is computed (and verified) from the first $\rho n + 1$ packets, the authentication information is computed without any decoding overhead. Similarly, the receiver can try to compute the authentication information after αn packets are received: this time the decoder runs completely, but computation is a less expensive, and if the correct authentication information is computed, no attack is in process and the delay is αn. Otherwise, if no polynomial can be verified, the receiver is under attack and βn delay is required in the worst case. In other words, our scheme can distinguish between the less expensive *detection* of an attack by an adversary from the more expensive *verification* of the valid received packets. We believe that this feature is desirable, for less computational effort is spent when no adversary acts.

We can summarize the performance of our scheme as follows.

Theorem 4.7.1. *For any (α, β)-network, the multicast authentication scheme* AuthECC *achieves the following performance in authenticating n packets.*

- *The sender performs one signature operation, n hash computations and $O(n \log n)$ field operations.*
- *The receiver performs $O(1)$ signature verifications, βn hash computations and $\tilde{O}(n^2)$ field operations.*

- *The communication overhead is constant per packet, proportional to* $\frac{\beta}{\alpha^2}$.

AuthECC *introduces a delay of* n *packets at the sender and a delay of at most* βn *packets at the receiver. Also, it allows the receiver to detect an attack after* $\frac{\alpha^2}{(1+\epsilon)\beta}n + 1$ *(which is less than* n*) packets have been received, where* ϵ *is the tolerance parameter of the decoder.*

4.7.1 Tuning and Extensions

Given specific values of the survival rate α and flood rate β of the network, the parameter ρ, which controls the communication overhead, can be tuned by the tolerance parameter ϵ. This gives one degree of freedom in implementing the exact encoding-decoding procedures. Namely, bandwidth consumption can be decreased at the cost of increasing by a constant factor the time complexity and vise versa. A realistic deployment of our scheme can consider α and β as an additional degree of freedom: early packet streams (groups of packets) are encoded for bigger values of α and smaller values of β. Depending on the observed network's behavior, the network parameters can be later adjusted to a new desired level of security. Table 4.1 shows the communication overhead per packet for specific values of α, β and ϵ.

α	β	ϵ	$1/\rho$	cost c (bytes)	α	β	ϵ	$1/\rho$	cost c (bytes)
0.33	1.5	0.1	15.15	303	0.5	1	0.01	4.04	81
0.5	1.5	0.1	6.6	132	0.5	2	0.01	8.08	162
0.75	1.5	0.1	2.93	59	0.5	3	0.01	12.12	243
0.33	1.5	0.5	20.66	414	0.5	1	0.1	4.4	88
0.5	1.5	0.5	9	180	0.5	2	0.1	8.8	176
0.75	1.5	0.5	4	80	0.5	3	0.1	13.2	264

Table 4.1: Communication cost c per packet for various values of the survival rate α, flood rate β and tolerance parameter ϵ. We assume the use of the SHA-1 hashing algorithm, that is, $h = 20$ bytes. The communication cost should be compared with the size s of the signature in use (e.g., an RSA signature with $s = 256$ bytes). Recall that $\rho = \frac{\alpha^2}{(1+\epsilon)\beta}$ is the rate of the code in use and that $c = \frac{h}{\rho} = \frac{\beta(1+\epsilon)}{\alpha^2}h$.

Independently of the choice of parameters, our scheme can be further modified in two ways, achieving different trade-offs between communication cost and computational efficiency. First, we can decrease the communication overhead, by applying the technique of [110]. The idea is that, since (at least) αn packets are guaranteed to be received intact, a significant portion of the authentication information is obtained by the decoder for free and without decoding: the (at least) αn hash values of the valid packets. Thus, less authentication information can be used and less redundancy is added to packets. To implement this idea, one has to encode the n hash values appropriately and, thus, Reed-Solomon codes are

applied twice. Interestingly, as opposed to the case of erasure codes [110], in our case where Reed-Solomon error-correcting codes are used, the decrease of the communication overhead occurs only for appropriate ranges of values for the network parameters α and β.

In particular, let $\{X, X'\} \leftarrow C[n, k+1]_q(X)$ denote the application of systematic Reed-Solomon code $[n, k+1]_q$ on word X, where X' is the added redundancy. Also let $H = h_1 \circ \ldots \circ h_n$ be the hash values of the n packets. We get the modified scheme by encoding

$$\{H, H'\} \leftarrow C[\gamma n, n+1]_{q_1}(H)$$

and then

$$\{A, A'\} \leftarrow C[n, \rho n+1]_{q_2}(A),$$

where

$$A = H' \circ \mathsf{Sign}_{SK}(H),\ \gamma = 1 - \alpha + \sqrt{(1+\epsilon)\beta},\ q_1 = 2^h,\ \rho = \frac{\alpha^2}{(1+\epsilon)\beta},\ q_2 = 2^{\left\lceil \frac{|A|}{\rho n+1} \right\rceil}.$$

As in our basic scheme, $A \circ A'$ is split in n equal shares A_i and packet p_i corresponds to authenticated packet $a_i = GID \circ i \circ p_i \circ A_i$. At the decoder, by the network reliability (at least αn packets will be valid) it is guaranteed that a constant size list of candidate strings for $H' \circ \mathsf{Sign}_{SK}(H)$ is produced; also, list-decoding is transformed to unambiguous decoding by verifying a constant number of signatures. Furthermore, the receiver is always capable to list-decode the packet hashes H. If in total δn packets reach the receiver, $\delta \leq \beta$, then Corollary 4.2.2 holds, since, $t \geq \alpha n + (\gamma - 1)n \geq \sqrt{(1+\epsilon)\delta} n$. The per-packet communication overhead of this scheme is $\frac{(\gamma-1)h}{\rho}$. When $\gamma < 2$, with this scheme we save in communication overhead. That is, for network parameters α and β in appropriate ranges so that $\beta < \frac{(\alpha+1)^2}{1+\epsilon}$ we can decrease the communication cost by the constant factor γ at the cost of increasing the computational cost by roughly a factor of 2, since two applications of Reed-Solomon codes are required.

Also, by decreasing the field size, we can reduce the cost of performing field operations. For instance, we could split the authentication information into $\gamma\rho n + 1$ substrings of size ℓ, $\gamma > 1$ (e.g., $\gamma = 10$), consider each substring as a field element in $\mathbb{F}_q$, with $q = 2^\ell$, encode with a $[\gamma n, \gamma\rho n + 1]_q$ Reed-Solomon code, and split the augmented authentication information into n pieces (each of γ field elements). In this way, the communication cost stays the same, but field operations become faster. The number of field operations at the encoder or decoder is increased by only a constant factor. Depending on the hardware architecture, this modification may be useful. A drawback here is that one injected packet by the adversary is now affecting the decoding algorithm by a factor γ.

4.7.2 Comparison with Other Schemes

We compare our schemes with various classes of proposed multicast-authentication schemes.

Sign-All and Merkle Tree Schemes. The sign-all and Merkle-tree [148] authentication schemes are resilient to fully adversarial networks. The sign-all scheme involves one signature (resp. verification) operation per packet and a communication overhead that is equal to the signature size. Depending on the specific signature scheme in use, the parameters of our scheme or the architecture, both communication and computational costs of our scheme are comparable to the corresponding costs of the sign-all scheme.

Very short signature schemes have recently been proposed [13]. While the length of a signature can be as low as 160 bits, the security of this signature scheme is only proven in the random oracle model, and only under a strong assumption (Diffie-Hellman assumption in gap-DH groups, see Boneh and Franklin [12] for more on these groups). Signing every packet with this short signature, therefore, has a communication advantage over our construction, but loses in provable security. On the other hand, signing every packet with a provably secure signature scheme, such as the Cramer-Shoup [27] signature scheme or its modification due to Fischlin [43], will add about 500 bytes to each packet—which is more than what we have for reasonable α and β.

Additionally, signing every packet is undesirable in practice. Indeed, by signing every packet separately we lose both in efficiency and in architecture design since the secret key operations are computationally expensive and require extra need of security. Invoking a signature operation involves fetching the private key and temporarily storing it in the main memory of the system. When secret-key operations are performed at high rates, the secret key resides almost exclusively in the memory of the system increasing the danger of the key being compromised to other running processes in the system. Special-purpose hardware can be used to overcome this problem, but of course at a higher cost. In terms of secure architecture design costs, and also for provable security or efficiency reasons, the sign-all approach is inferior to ours.

Finally, since one signature verification must be performed for each received packet, valid or not, the sign-all solution suffers by the following denial-of-service attack at the receiver: by injecting invalid packets an adversary can increase the computation resources spent at the receiver for signature verifications. In our scheme, where signature dispersal is used, no such attack is possible.

On the other hand, the Merkle-tree scheme [148] has better time complexity than our

scheme. For a group of packets of size n, only $2n$ hash computations and one signature computation (resp. verification) are performed at the sender (resp. receiver). However, the Merkle-tree scheme has communication cost that grows with the number of packets, thus, this scheme is not scalable. Our scheme is efficient in terms of communication cost: packets have constant authentication overhead.

Another drawback for the Merkle-tree scheme operating in a fully adversarial network is the *signature flooding* attack, identified and described in [68]: in a way similar to the denial-of-service attack against the "sign-all" scheme, injected packets cause a signature verification at the receiver for the Merkle-tree scheme as well. Of course, we have to note that at the receiver, by appropriately caching hash values, we can significantly resist against the signature flooding attack: once the first valid packet is verified, its (authenticated) hashes are stored and subsequent packets need only be verified with respect to the hashes they carry and not with respect to the signature they carry. Because of that, injected packets that are received afterwards do not cause signature verifications. Although this kind of attack is thus terminated after the first correct signature verification, still, in a fully adversarial network with packets rearrangements, injected packets will precede the valid ones. In our (α, β)-network model, the Merkle-tree scheme [148] needs $(\beta - \alpha)n$ signature verifications. Instead, our scheme performs only a constant number of signature verifications at the receiver.

Graph-Based Schemes. These schemes [51, 100, 116, 136] assume the reliable receipt of a signature packet. However, a fully adversarial network will capture the signature packet and invalidate the scheme. Even if the signature packet is assumed to arrive intact, any efficient scheme in terms of communication overhead (i.e., with constant overhead for packet) will have the undesirable property that $O(1)$ critical packets can be adversarially chosen to disconnect from the authentication chain the signature node (packet). In the piggybacking scheme in [100], this number of critical packets can be $O(n)$ at the expense of a communication overhead of $O(n)$ per packet. Our scheme does not have these drawbacks since the signature is dispersed among all the packets. As opposed to graph-based authentication where the authentication of a packet crucially depends on other packets (with packets closer to the signature packet being more important), our scheme is symmetric in this context: all packets share the authentication information.

Erasure-Code Schemes. The first two erasure-code based proposed schemes [110, 111] make use of erasure codes to tolerate packet losses, up to a constant fraction. However,

no packet injections are tolerated: a single injected packet suffices to fail the decoding procedure. For networks where packets get only lost, they perform slightly better than our scheme in terms of communication cost and time complexity. This is due to the fact that erasure codes are more efficient than error-correcting codes in terms of time complexity and space requirement. Moreover, erasure codes can tolerate more symbol deletions than the theoretical limit $d/2$ for error-correcting codes (d is the diameter of the code). In our authentication scheme, tolerating injected packets comes at this small price of having *slightly* worse performance than erasure-code schemes.

In [68], Karlof *et al.* using distillation codes address the vulnerability to packet injections that any scheme based on erasure-codes has, but their proposed scheme has high communication overhead and is thus less scalable, because a Merkle hash tree is used to "filter out" the injected packets (and thus the communication cost is $O(\log n)$). For such a logarithmic communication overhead, the scheme by Wong and Lam [148] may be actually preferable since it has both lower time complexity and better resiliency to adversarial network behavior. In terms of computational effort at the sender and receiver this scheme is similar to our scheme except from the following two points regarding the computational effort at the receiver. In [68], partitioning the packets into groups introduces an extra computational overhead and the total number of hashing values computed is by a logarithmic factor larger. In our scheme, the constants involved in the quadratic decoding process are higher than in the scheme by Karlof *et al.*

Finally, in [61], the shared channel model that is used does not tolerate adversarially chosen packet losses.

Other Schemes. TESLA [115, 116] and the scheme by Xu and Sandhu [149] have very different assumptions from our model. They are both based on MACs and on strong time-synchronization requirements about the nodes of the networks that do not fit our model. For instance, in [149], the routers of the networks are considered trusted entities.

Tables 4.2 and 4.3 summarize the above discussion, where selected schemes are compared with our scheme AuthECC. In particular, Table 4.2 compares our scheme with the sign-all solution and various selected schemes that are not (α, β)-correct and secure. We consider two graph-based authentication schemes, one of constant degree (expander construction [136]) and one of $O(n)$ degree (piggybacking scheme with parameterized performance [100], where we assume a constant number of classes), and one erasure scheme (optimized in terms of communication scheme [110]). Table 4.3 compares our scheme with the only two (α, β)-correct and secure previous approaches, namely the schemes by Wong and Lam [148] and

	Sign-all	GB [136]	GB [100]	Erasure [110]	AuthECC
Delay (Sender)	1	n	n	n	n
Computation (Sender)					
Sign	n	1	1	1	1
hash	—	$O(n)$	$O(n^2)$	n	n
field op	—	—	—	$O(n \log n)$	$O(n \log n)$
Communication	sn	$O(hn)$	$O(hn^2)$	$\frac{1-\alpha}{\alpha} hn$	$\frac{\beta(1+\epsilon)}{\alpha^2} hn$
Delay (Receiver)	1	n	n	n	βn
Computation (Receiver)					
Verify	n	1	1	1	$O(1)$
hash	—	$O(n)$	$O(n^2)$	n	βn
field op	—	—	—	$O(n^2)$	$\tilde{O}(n^2)$
Secret key protection	—	•	•	•	•
Resiliency					
Chosen packet loss	•	—	•	•	•
Chosen packet injection	•	•	•	—	•
Signature dispersal	•	—	—	•	•

Table 4.2: Comparison of selected multicast authentication approaches, no (α, β)-correct and secure, with respect to various aspects of efficiency, security and resiliency. By Sign, we denote a signature operation, Verify denotes a signature verification, hash denotes the total hashing cost, where we consider that the complexity of hashing a string is a linear function of the string size. Also, we use the following notation: n is the number of packets in the data stream, s is the signature size and h is the hash size. Both the communication overhead and the computational costs refer to n packets.

Karlof *et al.* [68].

4.8 Conclusions

In this chapter, we propose a new general framework for the multicast authentication problem, where the network is controlled by a computationally bounded adversary that has great power in modifying the transmitted stream. Our model is realistic in terms the of adversarial behavior. The limitations on the adversary's power, characterized by the survival and flood rates, exclude from consideration only degenerate cases, where the authentication problem actually disappears.

Our work establishes a new direction in data-stream authentication by going beyond erroneous networks and addressing fully adversarial networks. Based on a novel combination of primitives from coding theory and cryptography, our authentication technique realizes an

	Merkle [148]	Distillation Code [68]	AuthECC
Delay (Sender)	n	n	n
Computation (Sender)			
Sign	1	1	1
hash	$2n$	$3n$	n
field op	—	$O(n \log n)$	$O(n \log n)$
Communication	$(s + h \log n)n$	$(\frac{1}{\alpha} + \log n)hn$	$\frac{\beta(1+\epsilon)}{\alpha^2}hn$
Delay (Receiver)	1	βn	βn
Computation (Receiver)			
Verify	$(\beta - \alpha)n$	$\frac{\beta}{\alpha}$	$O(1)$
hash	$2n$	βn	βn
field op	—	$O(n^2)$	$\tilde{O}(n^2)$
Secret key protection	•	•	•
Resiliency			
Chosen packet loss	•	•	•
Chosen packet injection	•	•	•
Signature dispersal	•	•	•

Table 4.3: Comparison of the three (α, β)-correct and secure multicast authentication schemes with respect to various aspects of efficiency, security and resiliency. Again, both the communication overhead and the computational costs refer to a group of n packets.

authenticated error-correcting code that constitutes a new general-purpose authentication tool. Our scheme is efficient and practical. It is as secure as the "sign-all" solution, but more efficient in both computational effort and communication overhead. Its constant communication overhead makes it scalable and preferable to the other approaches [68, 148]. When compared with the Merkle-tree based scheme, the $O(n^2)$ time complexity of our scheme at the receiver is a shortcoming; however, it is possible that in practice this may not be a serious concern. Additionally, our scheme can be tuned by the network parameters α and β and distinguishes between the less expensive detection of an attack by the adversary and the more expensive task of verification.

Remaining open problems are as follows. First, it is worth investigating the practical performance of our authentication approach by implementing it and conducting an experimental study. Also, a natural question to explore is whether the decoding procedure can be simplified and whether the time complexity can be improved. One other question is whether other classes of error-correcting codes can be employed in our framework.

Moreover, in this chapter we showed a connection between coding theory and cryptography. In particular, we employed cryptographic primitives to unambiguously list-decode

an error-correcting code. It would be very interesting to study whether there are other connections between the two areas. Finally, it is interesting to explore the use of our technique in other data authentication problems.

A preliminary version of the results of this chapter appears in publication [82].

Chapter 5

Authentication of Distributed Data

5.1 Introduction

The problem of data authentication is a fundamental one from both theoretical and practical aspect. From a theoretical point of view, data authentication introduces new dimensions in both algorithm design and cryptography. On one hand, known data management and data structuring techniques often need to be reexamined, in the new data dissemination settings, where the data distributor and the data owner are different entities. On the other hand, directly applying traditional and well-studied message authentication techniques for data authentication—where data cannot be treated as a whole—is often inadequate to provide efficient solutions. From a practical point of view, more and more in distributed and pervasive computing environments, information is delivered through untrusted computing entities, raising crucial security threats with respect to data authenticity. An important and very popular paradigm for implementing data dissemination in distributed environments is through the use of peer-to-peer networks.

Peer-to-peer (p2p) networks provide the basis for the design of fully decentralized distributed systems, where data and computing resources are shared among participating peers. Properties of such systems include scalability, self-stabilization, data availability, load balancing, and efficient searching. As p2p networks become more mature and established, a growing number of new applications emerge for them, with an increasing need for assuring security.

In this chapter, we study data authentication in p2p networks, where data originated at a trusted source is shared and dispersed over remote and untrusted network nodes and queried and retrieved by end-users through network's API. We focus our study on p2p

systems realizing a *distributed hash table* (DHT), which supports the basic put-get functionality over distributively stored data objects. We are interested in guarding users of p2p systems against misbehaving or malicious network nodes that falsify their actions after put operations or their responses to get operations.

However, current authentication techniques for contents of DHTs are static, centralized and, more importantly, often insecure. For instance, existing DHTs that support data authentication use signatures on a per-object basis. This can not guarantee full protection against malicious network nodes, since *replay attacks* can be easily launched, where old, invalid objects are still incorrectly verifiable. Moreover, there is currently no distributed implementation of the widely-used *Merkle's authentication tree* [95].

We introduce a new model for *distributed data authentication* in p2p networks that is based on DHTs, thus advancing previous client-server models for data authentication, and present an efficient implementation in this model of an authentication scheme for securely performing dictionary operations, thus, presenting the first efficient *distributed authenticated dictionary*. In essence, we present a generic technique for efficiently building an authentication structure on top of a broad class of DHTs, that extends their functionality to *authenticated operations* put, get *and* remove, thus providing a transparent security layer for target applications. By using only the basic functionality of object location, our scheme achieves generality and is easily applicable to existing DHT implementations. Our authentication scheme is based on the design of an efficient *distributed Merkle tree (DMT)*—the first distributed version of an authentication tree—that can be also used as a general-purpose distributed tree or verification structure. Our distributed authentication schemes can be used for authenticating contents, more generally, in any overlay network.

Towards achieving general results on distributed data authentication, we additionally study the problem of data authentication from a new perspective. In particular, we study the problem of authenticating general queries over structured data in the RAM model of computation. We formally define the problem in its general form and put forward a new framework for data authentication, where the answer validity rather than the querying process is actually authenticated, and we show that our approach achieves generality. We introduce the notion of reducibility of query authentication primitives and show that the authentication of any query type that can be answered based on the evaluation of relations over the data elements, is reduced to the authentication of membership queries. Using this, we prove general possibility results for the data authentication problem, under general assumptions, as well as, characterization theorems about the use of cryptographic techniques for this problem. Our authentication framework enjoys important properties and certain

advantages over previous approaches, including: generality, expressiveness and sufficient conditions for the design of new efficient, or super-efficient, authenticated data structures. An important consequence of our results is that, for any query problem, there exists a distributed authenticated data structure in the new model introduced in this chapter.

5.1.1 Perspective and Motivation

Data storage and retrieval are essential tasks in p2p systems, where large data collections (e.g., documents, media files, database records) are shared over a network among participating peers, that is, machines in general owned by unknown, and thus untrusted, individuals. In this highly distributed setting, ensuring correct and trustworthy system functionality, in the presence of faulty or malicious network nodes actively seeking to disrupt the system, is an important and challenging task, especially because p2p systems impose no restrictions on who may become a member.

In this setting, data authentication is a fundamental security problem, where we are interested in securely and efficiently authenticating contents stored in a p2p system. For instance, adversarial network nodes may wish to degrade the performance of a p2p storage system by providing false responses to data queries; e.g., they may respond with data that appears to be a file of interest, such as a video of a scientific lecture or a financial data file, but is instead of degraded quality, incorrect or virus infected. We wish to ensure the integrity of shared data and to provide cryptographically sound techniques that allow someone to verify that data retrieved from the system is authentic, unaltered in any possible way. Moreover, in a dynamic setting, where contents evolve over time through updates, we want to also ensure that data items retrieved by queries have the most up-to-date versions.

We consider the standard query model in p2p storage systems, where a DHT stores key-value pairs of the type (k, x) (keys are unique identifiers and values are associated with keys) and supports operation $\mathsf{put}(k, x)$ (which inserts a new pair in the system) and query $\mathsf{get}(k)$ (which returns the value associated with key k). Despite its basic functionality, this is an important distributed data structure and various efficient implementations of DHTs (e.g., [124, 138]) provide the core framework for designing and implementing more complex distributed applications (e.g., [29, 65, 126]). As systems based on DHTs gain popularity and grow in scale and complexity, the demand of security for them increases. Thus, we also wish to achieve high information assurance at the application level, by designing efficient and easily usable cryptographic techniques over DHTs that will guarantee the integrity of stored data and the authenticity of retrieved data.

A straightforward approach to the authentication of queries is to individually sign each data item stored in the DHT: when data source S wishes to add (k, x), it computes the signature σ of pair (k, x) using its private key and inserts $(k, (\sigma, x))$ into the data structure, and a query for key k now returns the pair (σ, x), where signature σ allows one to verify whether x is the valid answer.

However, this "sign-all" approach introduces significant storage overhead and is vulnerable to *replay attacks* for old values, because it does not provide any mechanism for invalidating old signatures on currently invalid pairs (that have been expired or removed from the DHT or whose value has been modified). Therefore, in response to a $\mathsf{get}(k)$ operation, a malicious network node can return an invalid (old or out-of-date) value that is still verifiable—since it carries a signature by the data source! This is a serious attack that is easy to perform (e.g., by caching and never deleting old pairs) and compromises security, by allowing the authentication of falsified queries. Note that, by definition and also for efficiency reasons, DHTs support no explicit item deletion, but only keep a soft state in the system, where old inserted data items are expired after a time interval and removed from the system, and to maintain these items, one has to essentially reinsert them. Nevertheless, even when some form of item deletion is supported, replay attacks are still possible: invalid signed pairs can simply be cached and never be deleted. In general, we need a mechanism which ensures that only recent signatures are used to validate answers to queries and which authenticates that—in addition to item insertions and retrievals—item deletions are correctly handled by the system.

Replay attacks can be prevented by introducing time-stamps in the signed values and a validity period, called *time quantum*, so that only verifiable signatures of the most recent time quantum are accepted. Even with this extension, "sign-all" solutions incur a significant computational overhead: after each time quantum, each of all the valid pairs that currently reside in the system need to be retrieved, resigned and reinserted in the DHT by the source.

On the other hand, it is preferable to maintain at all times a global, correct authentication state of the system, that includes only the currently valid data items and essentially authenticates that data is properly updated. This is achieved by *signature amortization*, the state-of-the-art technique for dynamic data authentication, where a data source S signs only one digest (short cryptographic description) of the entire collection of (valid) stored data items owned by S. The canonical method for amortizing one signature over a large data set is Merkle's authentication tree [95]; however, there is currently no distributed implementation of this scheme.

Unfortunately, existing p2p storage systems and DHT implementations that support an

authentication service for the stored data are all using "sign-all" techniques. Thus, replay attacks are feasible and, if time-stamps are used to solve the problem, this leads to inefficient and impractical authentication schemes due to the need for signature refreshing.

5.1.2 Previous and Related Work

We next review previous work on distributed data authentication.

Authentication trees. The Merkle tree [95] is a simple, and widely-used in security applications, cryptographic construction for efficiently certifying set membership. The idea is to use a tree and a cryptographic collision-resistant hash function (e.g., SHA-1) to produce a short cryptographic description of a large data set. Elements of the set are stored at the leaves of the tree and internal nodes store the result of applying the hash function to the concatenation of the values stored at the children nodes. The root value is signed and, when verified, the collision-resistant property propagates authentication from the root to the leaves. Certifying that an element is in the set is performed by using a *verification path*, which consists (of the hash values) of the siblings of the nodes on the path from the leaf associated with the element to the root, to recompute the authentic root value. Updates in the Merkle tree are handled with complexity proportional to the height of the tree [103]. An extension to the symmetric-key setting is given in [64], where it is shown that verification along a path can be performed in parallel. No distributed implementation for Merkle trees currently exists.

Authenticated data structures. These authentication structures provide a client-server model (e.g., [103, 141]) for authenticating data that is queried not from the trusted data source, but from different, untrusted, entities (e.g., remote servers). This model augments a data structure such that along with an answer to a query, a cryptographic proof is provided that can be used to verify the answer authenticity. The technique of signature amortization is used, similarly to the Merkle tree, but specially designed according to the supported query type. A significant body of work has been done on developing efficient authenticated data structures for various type of queries (e.g., [7, 32, 34, 59, 83, 86]). The related model of *outsourced database systems* studies the special case where SQL queries (essentially, range queries over indexes) are issued over databases published at remote sites (e.g., [78, 102, 106]). Both models involve untrusted hosts each maintaining the entire data set, thus, they cannot capture the needs of data authentication over p2p networks, where data is shared and distributed on a per-item basis. Also, our work provides a useful framework for the design

of new efficient (distributed) authenticated data structures of any query type. Finally, multiple-source extensions of authenticated data structures are studied in [55, 107].

DHTs and p2p storage systems. There is a large and growing literature on p2p networks. One popular class of such networks is that of DHTs, fundamental distributed structures that make use of consistent hashing to efficiently support queries for exact matches with data keys (e.g., Chord [138], Koorde [67], Pastry [129], Symphony [85] and [124]). Searching these structures is based on randomized distributed routing techniques and, for a broad class of them, an object is located with $O(\log n)$ expected communication steps, where n is the number of participating nodes. With advances in distributed object searching and the development of DHTs, several practical distributed storage systems over p2p networks have been designed and implemented that support efficient data retrieval (e.g., PAST [38], CAN [124], CFS [29], PIER [65] and OpenDHT [126]).

Trees over DHTs. The development of DHTs was followed by the design of various search trees and aggregation trees built over DHTs or other type of distributed trees (e.g., [26, 44, 66, 79, 123, 150]). However, these trees can not be used to implement a distributed Merkle tree, since most of these constructions are static (they do not support dynamic updates) and they are either search trees or special-purpose trees that are actually not appropriate to realize an authentication tree—which is sensitive to node losses or structural changes because of the use of the cryptographic hash function. BATON tree [66], although dynamic, can not be used for our purposes (e.g., is not built on DHTs and also, based on AVL trees, is not appropriate for our design goals).

Security in p2p systems. Some security issues related to p2p systems are discussed in [133], where the authentication problem is treated simply using per-item signatures. Although with respect to routing and searching, numerous DHTs have been shown to tolerate significant network-node failures—random (e.g., [67, 124, 129, 138]) or malicious (e.g., [40, 41, 74, 130])—data authentication has not been systematically studied on p2p networks. Existing p2p storage systems (e.g., [29, 38, 117, 124, 126]) support an elementary authentication service for retrieved data which is of the "sign-all" type, where stored contents are individually signed by their source. Often, authentication involves the so-called *self-certified data* [45], where large data items (e.g., a file system) get partitioned into blocks, which are stored as separate objects in the system and are bound together using collision-resistant hashing in some tree-like hierarchy, and where the root-block is signed. Although

this technique resembles a Merkle tree, it only implements signature amortization among a large item and not among all data items owned by a source, which are still separately signed. Additionally, this authentication structure is static (no updates are supported) and, generally, unbalanced (e.g., file systems are usually flat). Overall, currently used authentication solutions are vulnerable to replay attacks (even if item removal is supported, as, e.g., in [126]) and lack efficiency for supporting signature refreshing and updates.

5.1.3 Authentication Model and Contributions

We introduce a new model for distributed data authentication, where data is stored and queried in a totally distributed fashion and retrieved data is accompanied by proofs that verify its authenticity. Our model captures the security requirements for data authentication that arise in p2p distributed storage systems and extends previous authentication models that are based on the client-server computing paradigm. In particular, our authentication model consists of:

- A *data source* S maintaining a (structured) data set D;
- A *distributed p2p network* $\mathcal{N}$ that stores set D on behalf of source S and supports authenticated queries about D, providing both *answers* to queries and *proofs* of answers' validity; D is dynamic and evolves in time through updates submitted by source S to network $\mathcal{N}$.
- *Users* who issue queries about D by accessing network $\mathcal{N}$ and verify the answers using the proof.

We are interested in designing secure *distributed authentication schemes* that involve totally decentralized data management and impose low computational, communication and storage overhead to the participating parties and the underlying network. Informally, an authentication scheme is secure, if for any query issued by a user to network $\mathcal{N}$, it is computationally infeasible for any malicious party controlling $\mathcal{N}$ to succeed in causing a user to accept (verify as correct) an incorrect answer. Also, cost parameters we try to minimize are: (1) *storage cost*, the amount of information stored at S, $\mathcal{N}$ and a user; (2) *update cost*, the computational and communication costs incurred at S and $\mathcal{N}$ when updates to data set D occur; (3) *query cost*, the computational and communication costs incurred by $\mathcal{N}$ to answer queries; (4) *verification cost*, the computational cost incurred by a user to verify the correctness of an answer.

Our model drastically differs from those of authenticated data structures (ADS) and outsourced databases (ODB) (see Figure 5.1) in that distribution of data and authentication

information over a network is performed on a data-item basis, not on a data-set one. Also, the users and the source do not have access to the structure of the network and interact with it only through its interface. Thus, we extend the client-server model of data authentication to a distributed authentication model that operates over a p2p network.

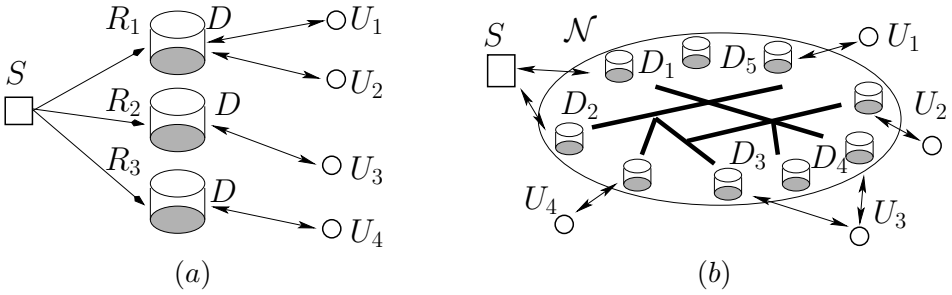

Figure 5.1: (a) Authenticated data structure (ADS): responder-servers R_i store set D and answer queries by users on behalf of source S. (b) Authentication over p2p networks: data and authentication information are dispersed in network $\mathcal{N}$; updates and queries are performed by $\mathcal{N}$ after contacting any node of it.

In our work, we consider the underlying p2p network to be any DHT. By designing our distributed authentication schemes using this popular and widely accepted distributed data structure, we allow this to leverage a broad class of existing p2p architectures, thus providing p2p systems with a transparent secure layer at the application level. We achieve generality by building our authentication scheme over the primitive (and common to all DHTs) search operation locate, which returns the network node corresponding to a given abstract object identifier. Since our constructions do not depend on the details of the DHT implementation, we gain simplicity, extensibility and usability; for instance, we can strengthen the resilience against malicious nodes by using a DHT that authenticates routing information, and our scheme can secure existing applications, providing an easily installed content authentication service.

Note that our scheme inherits the following properties shared by most DHT implementations: (1) a DHT with n network nodes uses $O(\log n)$ storage per node and performs a locate operation (also, put and get) in $O(\log n)$ network hops (node-to-node communication steps) with high probability; (2) node additions, deletions, and failures are handled dynamically through a distributed algorithm that incrementally updates the routing information; (3) some form of redundancy is used, which replicates data objects to a constant number of neighboring nodes so that node failures are tolerated also with respect to the data stored at them; and (4) caching techniques are used to improve data retrieval.

We next summarize our contributions in this model. We present the first efficient scheme for implementing a *distributed Merkle tree* (DMT), using only the object-location functionality exported by a DHT. Our scheme has certain properties that allows its efficient distribution over a p2p network and is specially designed to support locality for answer verification and facilitate the use of caching, thus achieving extra efficiency and resilience against malicious nodes. We analyze its performance and compare it with naive implementations. Our tree construction, designed for both bottom-up and top-down access, constitutes a new, general-purpose, dynamic and efficient distributed tree.

We present an efficient *authenticated distributed hash table* (ADHT), which extends (non-authenticated) DHTs in various ways. In particular, using our DMT, we present an authentication structure that provides *authenticated* and efficient versions of operations get, put, and also operation remove, supporting *authenticated deletions*, the first of this type. Our ADHT provides efficient distributed storage, secure against replay attacks. We compare ADHT with the "sign-all" solution with respect to various costs.

Finally, we present a secure and efficient distributed authentication scheme for dictionary operations. In particular, using our ADHT we implement the first efficient *distributed authenticated dictionary*. In a totally distributed setting over a p2p network with n nodes and using only the basic object-location operation, we show how one can efficiently authenticate membership queries in a fully dynamic set of m data elements in $O(\log n \log m)$ time using $O(m \log m)$ storage, with similar complexities for supporting updates.

Overall, we summarize how our work is qualitatively compared with other existing authentication methods (authenticated data structures and outsourced databases) with respect to distributed data authentication in Table 5.1. Our methods are the first to provide secure and efficient data authentication in totally decentralized environments over p2p networks, capturing the security and computing needs of numerous real-life applications.

	Decentralized	Replay Safe
"Sign-all" DHTs	•	
ADS, ODB		•
Our results	•	•

Table 5.1: Qualitative comparison of our methods with existing data authentication models.

Additionally, we generalize our results on (distributed) data authentication to hold for any type of query. In particular, we show that for any query problem there exists a distributed authenticated data structure that authenticates answers to these queries. We

achieve this result by showing the following important (and of independent interest) result: the authentication of general queries can be reduced to the authentication of set membership queries.

5.1.4 Chapter Structure

In Section 5.2 we present our first main result, the implementation of a fully dynamic distributed Merkle tree over a p2p network and we analyze its performance. In Section 5.3 we show how our distributed Merkle tree can be used to realize an authenticated distributed hash table, which in turn can support a more general data authentication scheme for membership queries. Also, Section 5.4 describes another implementation of a distributed Merkle tree that achieves load-balance. Finally, Section 5.5 describes in detail our results on authentication of general queries and, in particular, a new framework for designing efficient authentication schemes which also proves that the authentication of any query can be reduced to the authentication of set-membership queries. We conclude and discuss open problems in Section 5.6.

5.2 Distributed Merkle Tree

We now present a distributed Merkle tree (DMT), that is, an efficient distributed implementation of a Merkle tree built over a p2p network realizing a DHT. Numerous security protocols and cryptographic constructions are based on Merkle trees, thus a DMT yields distributed versions of such protocols and constructions. Our design goals are as follows: the tree must be balanced and efficiently maintainable and verification paths (membership proofs) should be located and updates should be implemented in a distributed way. Additionally, the cost parameters we wish to minimize are: the *path location cost*, the cost for constructing (locating in the network) a verification path (proof), the *update cost*, the cost for maintaining the authentication structure after updates on the data set, and the *storage cost*. Both the location and update costs each consists of (1) processing cost, i.e., computational cost for the participating nodes in the system, and (2) communication cost, i.e., cost of location operations or direct communications between nodes. We are particularly interested in facilitating the location (and creation) of the verification paths of our tree. We wish to use only the locate operation provided by the underlying DHT; with n network nodes, this operation takes $O(\log n)$ time.

We first describe some simple and inefficient solutions that still give us an insight of what an efficient scheme should achieve, assuming a basic scenario, where each network

node i has an object x_i and we wish to distribute a balanced Merkle tree built on top of data set $\{x_1, \ldots, x_n\}$.

1st approach: Tree replication. The first approach is to build a regular hash (Merkle) tree on top of x_i's and then store the hash values of the tree as new "regular" objects in the system. The first problem to consider is how the hash values are indexed, i.e., with which keys they are stored in the system. The hash value is a value that is unknown to network nodes, thus the value itself cannot be used as a key. A straightforward solution to overcome this problem is to replicate the tree structure to all involved network nodes and then to use unique identifiers (according to some fixed encoding) for storing hash values in the DHT. So, if nodes have a view of the current hash tree, then we have a functional DMT. The cost to construct a verification path is $O(\log n)$ locate operations, that is $O(\log^2 n)$ time and communication cost. However, the cost to maintain the tree, after updates, at each network node is high: an update triggers information of $O(\log n)$ size to reach each network node, which requires the existence of a flooding-type broadcast capability over the DHT and $O(n \log n)$ communication cost. Also, the $O(n^2)$ total storage is prohibitive.

2nd approach: Path replication. Each node stores the entire verification path of the object it stores. Thus, the path location cost is $O(1)$, but now the update cost amounts to $O(n)$ locate operations (each new hash value must reach $n-1$ other nodes) or $O(n \log n)$ time and communication complexity. There are extra difficulties, such as how a new hash value is computed and by whom; this involves some specific communication protocol between nodes (e.g., Merkle tree traversal techniques). The total storage cost is $O(n \log n)$.

Our approach: Route distribution. We design an efficient dynamic DMT using route distribution. The idea is as follows. Let T be a balanced binary tree defined on top of elements $x_1, \ldots, x_n$. Each tree node u has a tree id $id(u)$. Tree T is used also as a hashing structure (i.e., as a Merkle tree). That is, a cryptographic hash function is used to label each tree node u with a hash value $L(u)$ (the value that we get by applying the hash function to the labels of its children). Hash values (tree node labels) are stored in the DHT as regular values keyed by the corresponding tree id; i.e., label $L(u)$ of tree node u is stored at the network node U corresponding to tree id $id(u)$. We also store at U the labels of the children of u. Consider element x stored at leaf node w and let $p = (w, u_1, \ldots, u_k, r)$ be the path from w to the root r of T. The node of the network storing element x is storing information related to path p of tree T. The stored information at w includes: (1)

the structural information of path p, i.e., left-right relation of nodes in the path p; (2) the balancing information of nodes in path p, i.e., information that is used for restructuring the tree and maintaining its balance; and (3) sufficient information for locating the hash values of p, namely ids $id(u_1), \ldots, id(u_k), id(r)$. Note that the verification path is completely accessible by this information, which does not include any hash values (tree labels).

This authentication structure allows queried nodes to report *immediately* the $O(\log n)$ tree nodes storing the hash values in the path. Then the user has to contact $O(\log n)$ nodes, by performing $O(\log n)$ locate operations; or alternatively, the queried network node collects the proof, not the user. In any case, the path location cost is $O(\log^2 n)$. We emphasize that using route distribution, that is, maintaining the invariance that each network node in the system knows the route for its verification path, we can achieve extra efficiency, as we will discuss at the end of this section. The authentication structure uses $O(n \log n)$ total storage, thus it is space-optimal, since the $O(\log n)$ storage per-network node overhead matches the per-node storage overhead of the DHT itself (for keeping routing information). Regarding updates of hash values (e.g., after element x_i changes value), new hash values along the corresponding verification path p can be computed using $O(\log^2 n)$ communication cost ($O(\log n)$ location operations suffice in updating $O(\log n)$ hash values). Regarding the update of the tree itself (e.g., structural updates for balancing purposes), note that using its balancing information a node can locally compute the new tree structure in $O(\log n)$ time, but then it has to accordingly advertise the changes to all other involved nodes, a seemingly very expensive task. However, we note that although all hash values in p change after every update, not every node of p change balancing or structural information and we take advantage of this fact. Only nodes that need restructuring must be updated and their changes must be advertised. Using $BB[\alpha]$ trees, which are weight-balanced trees enjoying important balancing properties, we can actually achieve that *on average* $O(1)$ *rotations* occur after an update and they occur more often at nodes closer to leaves than at nodes higher in the tree. Each such rotation involves communication cost proportional to $O(k \log k)$, where k is the size of the subtree rooted at the place where the rotation took place. We thus expect on average a very good performance in an amortized sense, since expensive reconstructions happen rarely. We next give the detailed description of our scheme and its complete analysis.

5.2.1 An Efficient Distributed Merkle Tree

We consider the more general case, where an authentication structure over $m \leq n$ data items is built over a DHT and we design our DMT using the primitive locate operation over a p2p network. We consider the basic case where m data items owned by a source are stored in the network and, without loss of generality, we assume that objects are stored at distinct network nodes. Our results generalize to the cases where more than one data items are stored at nodes and also where more than one sources produce these items.

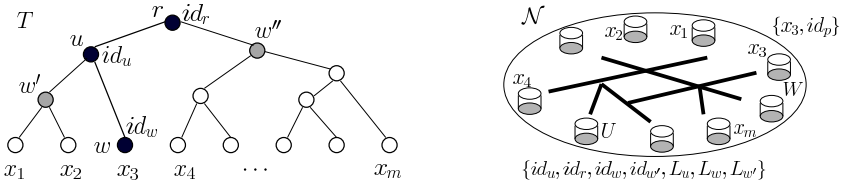

Figure 5.2: (a) Balanced hash tree T over data items $x_1, \ldots, x_m$: tree node u, storing label L_u, is identified by id_u and mapped to network node U; leaf node w corresponding to data item x_3 with verification path p is mapped to network node W. (b) Tree distribution over the network: network node U corresponding to tree node u stores $\{id_r, id_u, id_w, id_{w'}, L_u, L_w, L_{w'}\}$ and local structural information about u; network node W corresponding to w stores $\{x_3, id_p\}$ and structural and balancing information about p.

Our scheme is described as follows (see Figure 5.2). For convenience, tree nodes are denoted by lower case letters and network nodes by capital ones. Let T be a balanced binary tree build over a dynamic data set of m data items $x_1, \ldots, x_m$ and let h be a cryptographic hash function. Tree T is used as a hashing structure in the standard way: each tree node u in T has a unique id $id(u)$ (drawn from some space) and is associated with (or stores, conceptually) a label $L(u)$, which equals to hash $h(L(v_1)\|L(v_2))$ of the hash values that are associated with (stored at) its children v_1 and v_2, and each leaf stores the hash value $h(x_i)$ of the corresponding object x_i. We augment the hashing structure as follows: we require that internal tree node u with children v_1 and v_2 also stores the hash values of v_1 and v_2.

Each tree node u is mapped to a network node $U = f(u)$ through a function f. Node U stores the (three) hash values associated with node u, the tree node ids of the parent tree node and the children of u and local structural information about node u. Moreover, a leaf node w_i, corresponding to object x_i, is also mapped to a network node $W_i = g(x_i)$ through function g.[1] Node W_i stores the following information: the object x_i, and information

[1]We impose no restrictions on functions $f(\cdot)$ and $g(\cdot)$. In general, $f = g$; these are known functions used by the underlying p2p network to map objects to network nodes (many DHTs use the SHA-1 function).

related to (verification) path p_i in T from w_i to the root r of T; in particular:

- the ids of the tree nodes of path p, denoted as id_p;
- the structural and balancing information of tree nodes in p, that is, for each tree node u in p with children v_1 and v_2, W_i stores: (1) whether v_1 or v_2 belongs in p; (2) the balancing information of node u, which is basically a pair (b_1, b_2) of balancing information related to subtrees defined by v_1 and v_2 respectively (for $BB[\alpha]$ trees, this is information related to tree-weights).

Our scheme distributes tree nodes and verification paths over a p2p network, correctly implementing a DMT at only logarithmic per-node storage overhead. Our tree is designed mainly for bottom-up use, which is appropriate for most security-related and cryptographic applications (although it can be easily extended to support also top-down traversal, as a search tree). Accordingly, our tree is accessed very efficiently: given a data item, the corresponding verification path is distributively retrieved using $O(\log m)$ location operations. Finally, we choose our tree T to be a weight-balanced tree and, in particular, a $BB[\alpha]$ tree. Our choice is related to the efficiency of the scheme with respect to the update costs incurred after changes in the tree occur due to updates in our data set (items are inserted or deleted or change value). We consider two types: structural updates (due to re-balancing changes) and hash updates (due to rehashing).

We now discuss the scheme correctness, i.e., we show that the above scheme implements a distributed Merkle tree. We describe how the tree is efficiently accessed, how it is maintainable after updates and also the associated computational, communication and storage costs. First, regarding the storage efficiency, we can easily see that our scheme requires $O(m \log m)$ storage. Indeed, internal tree nodes are stored in the network each using $O(1)$ information and m leaf nodes are represented each using $O(\log m)$ information, since T is balanced. Now, assume that a tree T built over $X = \{x_1, \ldots, x_n\}$ is distributed over a network $\mathcal{N}$ and consider the task of accessing or locating the verification hash path corresponding to data item $x_i \in X$ and path p_i, initiated by network node M. Node M first locates the network node W_i that stores x_i (through mapping $g(\cdot)$), then W_i reports to M (through a direct connection) the ids of the tree nodes in the corresponding path p_i of T. Then M can locate the $O(\log m)$ network nodes storing hashing information related to the verification path of x_i; the mapping $f(\cdot)$ have to be used first to map tree ids to network

We use distinct functions to denote the possibility that more efficient schemes can be designed by having f and g satisfying some relation.

nodes. After contacting $O(\log m)$ nodes, node M has retrieved all the verification information. Note that once a network node is located and contacted, not only the corresponding hash value of the tree node is retrieved but also the hash values of the children nodes. Thus, overall, retrieving the verification hashing path of an item takes $O(\log n \log m)$ cost. In particular, it involves $O(\log m)$ location operations and $O(\log m)$ communication cost (through direct network connections).

Next, consider the problem of maintaining the tree balanced after an update in the hash tree. The simplest update corresponds to simply changing an object x_i to a new object x_i', where $g(x_i) = g(x_i')$, that is when the storage location of x_i does not change [2]. We call these updates *hash updates*, since only the hash values are updated. It is easy to see that only $O(\log m)$ hash values need to recomputed, whereas the structure of the tree stays the same. This can be done by node $N = g(x_i)$ (or any other node that initiates the update, locates N and contacts N) locating and contacting the network node U corresponding to the lowest in T node u in p_i and notifying it about the change; then through $O(\log m)$ node locations and contacts every node in p_i updates the hash value it stores and notifies the network node storing its parent in T about the update. Thus, the update cost for this particular type of update is $O(\log n \log m)$, since locating a network node takes $O(\log n)$ communication cost. Now consider the general case of an update, that is, an operation of the type $\mathsf{insert}(\cdot)$ or $\mathsf{delete}(\cdot)$ on the tree T. (Note that a $\mathsf{replace}(\cdot)$ operation when $g(x_i) \neq g(x_i')$ corresponds a series of two such operations.) Then, not only $O(\log m)$ hash values need to be updated, but also the tree T needs structural update, due to re-balancing operations. The distributed update process is as follows. Network node W_i responsible for the update on data item x_i performs gradually the update and in a bottom-up fashion, according to its corresponding path p_i. That is, a leaf node is deleted or created and the path is checked using a bottom-up traversal of it, for any necessary restructuring operations (i.e., rotations). Note that all necessary information is available at node W_i for this check (the balancing information of p_i is stored at W_i). Node W_i traverses path p_i and if no rotation is needed while examining tree node u, then the network node $U = f(v)$ corresponding to u is contacted (after it is first located) so that its hash value is recomputed. If additionally a rotation is needed at node u, then node U is notified appropriately by W_i and $f(v)$ executes the rotation by contacting (after first locating) the appropriate nodes among its neighboring in T network nodes. Node W_i is notified about the structural change, i.e., it learns the ids and the balancing information of the new nodes in p_i'. (Alternatively, once the rotation

[2] This is not an extreme special case, but rather of typical that often occurs when for instance data objects are actually key-value pairs and only the value is being updated.

is complete, node W_i updates the balancing information of the affected by the rotation nodes by contacting them.) Then node W_i goes on to the node higher in p_i. Thus, the new path p_i' can be computed completely after $O(\log m)$ location operations and $O(\log m)$ communications between network nodes, that is, the cost for updating the verification path p_i to the new path p_i' is $O(\log n \log m)$. However, since verification paths are distributed over the network $\mathcal{N}$ and if path p_i structurally changed to p_i' then it must be advertised to the network nodes which leaf nodes in T that belong in subtrees affected by the rotations are mapped to through $g(\cdot)$. We refer to this cost as *structural update cost*. Note that for general trees T the structural update cost can be of order $O(m \log m)$, involving $O(m)$ network node locations: indeed, a rotation at level k of T requires $O(2^k)$ location operations, proportional to the size of the affected subtree of T. This is because, the change due to a rotation must be distributed to the appropriate nodes of network in a top-down fashion by a series of node locations and communications. However, recall that our scheme uses a weight-balanced $BB[\alpha]$ tree as T, where the weight of a tree node is the number of leaves in the subtree defined by this node and α is a balance parameter. Our choice is justified by the following lemma. Over a linear number of update operations, our scheme requires only a logarithmic number of node locations.

Lemma 5.2.1. *For a series of $O(m)$ update operations on an initially empty data set, the above DMT T based on a weight-balanced $BB[\alpha]$ tree, with $\alpha \in (\frac{1}{4}, 1 - \frac{\sqrt{2}}{2})$, has $O(\log n \log m)$ amortized structural update cost. In particular, during this series of tree updates, structural updates at level k of T with cost $O(2^k)$ occur with frequency $O(\frac{1}{2^k})$.*

Proof. The proof is based on the update technique in our scheme and the properties of $BB[\alpha]$ trees (e.g., see analysis in [89]). If path p_i is structurally updated to p_i', let u^* be the node of p_i where a rotation took place and no other rotation occurred at an ancestor node of u. Let T_u^* be the subtree in T defined by u^*. Then all network nodes L_u^* corresponding to leaf nodes in T_u^* must update their paths, because for all these paths at least one tree node has changed (due to the rotation at u^*). The update should also propagate to the neighboring tree nodes of u^*. This update in T can be done (once for all, for the entire update due to all rotations) by distributing the updates to nodes in L_u^* through network nodes corresponding to subtree T_u^*. The distribution occurs in a top-down fashion and by network nodes locating the nodes corresponding to their children and communicating to them the relevant updates [3]. The whole process is complete by using $O(|T_u^*|)$ node locations and $O(|T_u^*|)$ communication.

[3] Alternatively, this can be done by threading the tree T such that leaf nodes are connected, that is, node W_i corresponding to leaf x_i storing the tree ids of the neighboring leaf nodes in T.

Thus, overall we have that the structural update cost is $O(\log n \times (\log m + |T^*|))$, where $|T^*|$ is the size of the maximum subtree in T affected by the update. Again, the $\log n$ factor is due to location operations. Since T is a weight-balanced $BB[\alpha]$ tree [4] with parameter α in an appropriate range and a rotation at node u in T incurs $O(|T_u|)$ node locations, using the analysis in [89], we get that the total node locations for updating all verification paths in our hash tree, for a sequence of t update operations (insertions or deletions) on an initially empty hash tree, is $O(t \log t)$. Thus, for the same series of update operations, the total structural update cost is $O(\log n \times t \times \log t)$. Then, for $t = O(m)$, we get that the amortized overall structural update cost is $O(\log n \log m)$ over a sequence of operations of size linear on m. Moreover, using the additional property shown in [89], namely that costly rotations at levels close to the root occur rarely, (in fact with frequency inversely proportional to the corresponding subtree size), the proof is completed. □

Using Lemma 5.2.1, the following summarizes the efficiency of our scheme and our main result. We call a network optimal if location operations take $O(\log n)$, where n the network size.

Theorem 5.2.1. *There exists a scheme for implementing a distributed Merkle tree T over a peer-to-peer network $\mathcal{N}$ with the following properties. If m is the size of the set over which tree T is built and n is the total number of nodes in the network $\mathcal{N}$, with $m \leq n$, then:*

1. *The distributed Merkle tree uses space $O(m \log m)$, distributed over $O(m)$ network nodes, and incurs $O(\log m)$ storage overhead per network node.*
2. *A verification path has size $O(\log m)$ and can be accessed with $O(\log m)$ locate operations; thus, for an optimal network $\mathcal{N}$, the expected computational and communication cost for accessing a verification path is $O(\log n \log m)$.*
3. *A hash update on the distributed Merkle tree involves $O(\log m)$ location operations; thus, for an optimal network $\mathcal{N}$, the expected computational and communication cost of a hash update is $O(\log n \log m)$.*
4. *A structural update on the distributed Merkle tree involves $O(m \log m)$ location operations, amortized over a series of $O(m)$ structural updates on an initially empty tree; thus, for an optimal network $\mathcal{N}$, the expected amortized computational and communication cost of a structural update is $O(\log n \log m)$.*

[4] Actually, the distributed Merkle tree is an *augmented* such tree where rotations at node u cost $O(|T_u|)$.

Improvement through caching. We now discuss a simple extension that under a reasonable assumption can improve the costs for path location and updates of our scheme. This improvement is applicable because of the design of our DMT based on route distribution. Namely, assuming that network-node failures occur less often than queries and updates on the DMT, we can improve the efficiency of our scheme as follows. The goal is to transform the multiplicative $O(\log n)$ factor (introduced due to locate operations for retrieving or updating hash values) into an additive term in the complexity of our scheme. This is easily achieved by caching network node ids: the idea is to have each network node corresponding to a leaf of the tree to cache in its memory the $O(\log m)$ network node ids that store the hash values of its corresponding verification path. That is, once such network node is first contacted, its id is remembered. Of course, since network nodes can fail or go down, it is possible that cached nodes are no longer nodes of the network. In this case, we have a cache miss which will trigger a location operation. Although we can still use some techniques to avoid this overhead (e.g., by contacting or also caching neighboring nodes storing the same information due to redundancy), we observe that when the rate of network node failures is sufficiently small then we can actually amortize the $O(\log n)$ factor due to occasional location operations (cache misses) in the cost for operating on the tree. In particular, if network nodes fail independently with probability $O(\frac{1}{\log m})$ during the time interval of a tree traversal, then the expected number of network node failures that occur during a path location or update is $O(1)$. Thus, using caching the expected complexity for path location and updates on the tree is $O(\log n + \log m)$.

	storage	path location	hash update	structural update
tree replication	$O(m^2)$	$O(\log n \log m)^*$	$O(\log n \log m)^*$	$O(m \log n)^*$
path replication	$O(m \log m)$	$O(1)$	$O(m \log n)^*$	$O(m \log n)^*$
route distribution	$O(m \log m)$	$O(\log n \log m)^*$	$O(\log n \log m)^*$	$O(\log n \log m)^{**}$
w/ caching	$O(m \log m)$	$O(\log n)^*$	$O(\log n)^*$	$O(\log n)^{**}$

Table 5.2: Efficiency comparison of various schemes for realizing a DMT for a data set of size m over a p2p network of size $n \geq m$. We denote expected complexity using * and amortized expected complexity using **.

Table 5.2 summarizes the comparison between the various schemes for implementing a DMT. We see that our scheme provides an very efficient solution that, using caching and under reasonable assumptions, can be asymptotically optimal in an amortized sense. Finally, we note that, when $m > n$, we can appropriate extend our scheme by having each network node maintaining an additional data structure (for locating the stored elements),

and also that our scheme supports authentication of data collections of one data source; we can support multiple data sources simply by using multiple instantiations of our DMT.

5.3 Authenticated Distributed Hash Table

We now focus on the design of distributed authentication schemes in our authentication model. We first use our DMT to authenticate the basic operations of any DHTs and design an efficient *authenticated distributed hash table* (*ADHT*).

Consider a data source S that produces m data items and stores them in a DHT that supports the basic put-get functionality. We use our DMT to augment the functionality of a DHT and provide higher information assurance. We wish our new system to support the following operations:

- *Authenticated* put: any key-value pair can be inserted in the DHT by source S in a way that both the system authenticates S's identity and source S is assured about the correctness of the performed insertion.
- *Authenticated* get: any user of the system can retrieve the value that corresponds to an existing (stored in the system) given key in an authenticated way, receiving from the DHT a proof that is used for verification.
- *Authenticated* remove: a previously inserted in the system key-value pair can be removed from the DHT by source S in a way that both the system authenticates S's identity and source S is assured about the correctness of the performed data item deletion.

Implementation. We implement an authenticated distributed hash table ADHT using the following standard authentication technique. Signature amortization is achieved by the use of a Merkle tree (using a cryptographic collision-resistant hash function h) built on top the data items owned by the S. The hash of the tree root serves as the data digest and is signed by the data source. A data item is verified to be owned by S if the signed root hash value is verified to be authentic (signed by the source) and a verification path binds the item with the digest. The security of the technique follows for the security properties of the signature scheme in use and hash function h. A non-trivial part of the above design is the fact that now the system is a DHT over a p2p network. We make use of the DMT to compute the digest of the data set of S and to realize signature amortization and compute only one signature over m data items. We need to add an additional level of hashing in

the tree: the leaf node corresponding to (k, x) now stores the hash value $h(h(k)\|h(x))$. We assume that the source S keeps a copy of the signed digest of its data and that, using bootstrapping techniques, both S and the users have access (through direct connection) to an active network node of the underlying p2p network. We now briefly describe some of the details in authenticating operations put, get and remove.

Updates. Updates are performed by the source S by first contacting a network node and issuing an update request. The system then reports to S the verification path of the leaf node of the DMT that is related to the update, which further allows S to compute the new signed digest. We augment our DMT in two ways to achieve this. To make the path retrieval possible for put operations, we extend the DMT to also serve as a *search tree*, thus allowing top-down access. To implement the digest update securely, we augment the hashing scheme over the tree such that verification paths now contain all the additional information needed by S in order to locally execute the hash and structural update and compute the correct new root hash of the DMT. This is feasible because both hash updates and structural tree adjustments only happen along a path in a bottom-up fashion. The verification path is used by S to check that the received information is consistent with the current digest stored by S. Once the new digest is computed, S signs it and returns a copy to the network node. Then a regular hash-tree update is performed by the system to execute the put or remove operation. By this interaction, S needs only to keep $O(1)$ authentication information, the current signed digest. Asymptotically, no additional computational or communication cost is introduced by this extra interaction between the system and the source.

Queries. A user contacts a network node and requests the value of a key. A path retrieval query is executed over the tree by the system and what is returned to the user is: (1) the corresponding value, (2) the verification path (collection of hash values and relative information for computing the root hash) and (3) the signed digest. The user accepts the answer (value) if and only if the signed digest is valid and hashing over the value and the verification path results in a hash value that equals the root hash.

Note that in a way, the source updates the data set by effectively first querying it in a similar way that a user would do. Using Theorem 5.2.1, we can prove the following.

Theorem 5.3.1. *There exists an authenticated distributed hash table over an optimal peer-to-peer network of n nodes that supports authenticated operations* put, get *and* remove *on a data set of size $m \leq n$, such that:*

1. *The distributed authentication scheme is secure.*

2. *The storage at the source is $O(1)$; the storage at the network is $O(m \log m)$.*
3. *The query cost is $O(\log m)$, that is, $O(\log m)$ locate operations; or, equivalently, the expected time and communication complexity to answer a query is $O(\log n \log m)$.*
4. *The amortized update cost is $O(\log m)$, that is, $O(\log m)$ locate operations; or, equivalently, the amortized expected time and communication complexity of an update is $O(\log n \log m)$.*

Security. We briefly discuss the security of ADHT. This follows using standard reductions to the security of the underlying cryptographic primitives that are used in our authentication scheme, under standard hardness assumptions. That is, by using a family of collision-resistant hash functions and a signature scheme secure against adaptive chosen-message attacks, we can prove the security of ADHT. Note that as described above, the security of the source against adversarial behavior by the DHT (or the underlying network) is still captured, since the interaction between the source and the DHT is treated as a special type of querying.

Discussion. Existing data authentication approaches in DHTs use the "sign all" method for verifying contents stored by the same data source. Although some systems use signature amortization through hashing, this is performed within a large data item itself, e.g., a large file or an entire file system, not for the entire data collection that a source stores in the system. Thus, our scheme is the first to amortize one signature over any large collection of data items. DHTs typically do not support an explicit operation for item deletions (with some exceptions, e.g., OpenDHT [126]), but, instead, they often introduce a time-to-live (TTL) mechanism, so that old stored items expire and then they are automatically deleted. In this TTL-based approach, if the source needs to renew the stored data, it needs to insert it and thus to sign it again. Independently of whether or not items are expired or deleted by the system, the issue of data freshness is critical and is related to replay attacks. An old signed value may not be valid anymore with respect to the application that uses it and a signed statement may be copied and be forgotten even when it is not valid anymore. Thus, any valid signed statement should be a freshly signed statement. This can be done by signing time-stamped data or data that carries the time when its validity expires and requiring that not only a signature is verified but also it is fresh. In a system storing m data items where $O(m)$ signatures are used, the signing cost for updating them and maintaining an up-to-date state of valid items is also $O(m)$. Signing typically involves expensive computations, thus the introduced computational overhead is high. Instead, in our scheme only one statement

(the root hash) must be resigned. The signing cost is $O(1)$, at only the cost of increasing the query and update complexity by a logarithmic factor on m. Table 5.3 summarizes the comparison of our ADHT with the existing data authentication schemes.

	storage	sign cost	query cost	update cost	replay safe
"sign-all"	$O(m)$	$O(m)$	$O(\log n)^*$	$O(\log n)^*$	no
ADHT	$O(m \log m)$	$O(1)$	$O(\log n \log m)^*$	$O(\log n \log m)^{**}$	yes
ADHT-c	$O(m \log m)$	$O(1)$	$O(\log n + \log m)^*$	$O(\log n + \log m)^{**}$	yes

Table 5.3: Comparison of ADHT (plain scheme and scheme with caching) with "sign-all" schemes for authenticating queries on data set of size m over a network of size n, $m \leq n$. We denote expected complexity using * and amortized expected complexity using **.

Data Authentication in Peer-to-Peer Systems. An immediate application of our ADHT is a *distributed authenticated dictionary*, where membership queries on a data set are authenticated. Suppose that keys are drawn from a totally ordered universe. Our DMT is built on top of key-value pairs in a sorted sequence according to their keys and it is appropriately extended to also serve as a *search tree* (with top-down traversal). Additionally, to support authentication of negative answers, the source inserts in the system pairs of key-value pairs such that the keys are consecutive in the ordering used in the Merkle tree. This scheme has asymptotically the same performance as the ADHT described above, given by Theorem 5.3.1. Note that our distributed tree can be used in other (not necessarily security-related) applications.

Load-balance issues. Although our authentication structure achieves load balance with respect to data distribution over the p2p network (this is guaranteed by the properties of the underlying DHT), as described, it does not achieve load balance with respect to network access. For instance, network nodes that store the tree root are accessed much more often than other network nodes. This turns out to be an important issue that appears to hold in general: all existing techniques for achieving authentication over DHTs that use signature amortization, including our technique or techniques based on self-certified data, introduce congestion at certain network nodes. The problem is challenging, since load-balancing and efficient content authentication in p2p systems correspond to contradictory design goals: signature amortization introduces heavily accessed points in the system, whereas for load-balancing we wish network nodes to be accessed with uniform, rather than skewed, distribution.

However, we propose the following simple solution for load-balance in an amortized sense: after any update in the structure, we choose new tree node ids for all nodes in the corresponding verification path. New ids are chosen according to a random but specified way, such that no significant communication overhead is introduced in the structure. Effectively, over time, we expect to achieve smoother (closer to uniform) access patterns for network nodes.

The next section present an authentication structure that inherently (i.e., by construction) achieves load-balance.

5.4 Load-Balanced Distributed Authentication

In this section, we study the problem of authenticating content stored in peer-to-peer networks, thus ensuring that data storage and retrieval are secure and trustworthy. As before, we are interested in authenticating content distributively stored in network nodes forming an overlay network, e.g., through the use of any distributed hash table (DHT). That is, we would like to prove that data files claimed to have been added by the source were really put in the p2p network by the source and have not been modified.

We consider a standard query model where an underlying DHT stores key-value pairs of the type (k, x), each added through operation $proper(k, x)$, where keys are unique identifiers and values are associated with keys. The DHT supports query $\mathsf{get}(k)$, which returns the value associated with key k. The authentication problem that arises in this case is *data integrity*, simply ensuring that any query returns the correct, authentic corresponding value. So, if (k, x) is the currently valid pair that has been inserted in the system, in terms of security and information assurance, the property that (at least) must be satisfied is that any query on key k must return the correct corresponding value x.

A straightforward approach to the authentication of queries is to individually sign each item stored in the data structure. Namely, when data source S wishes to add (k, x), it computes the signature σ of the pair (k, x) using its private key and inserts $(k, (\sigma, x))$ into the data structure. A query for key k now returns the pair (σ, x), where the signature σ allows one (that knows the corresponding public key) to verify whether x is the valid answer. Unfortunately, this "sign-all" approach introduces significant performance overhead and it allows for replay attacks for old values. Indeed, this approach does not provide a mechanism for invalidating old signatures on pairs that have been removed from the DHT or whose value has been modified. Therefore, we end up with the following crucial security threat: a malicious network node can easily perform a "replay" attack, by simply reporting on a get

operation an invalid, *still verifiable* value, that corresponds to a old (expired or out-of-date) key-value pair.

In general, we need to provide *signature freshness*, that is, of mechanism that ensures that only recent, valid signatures are used to validate answers to queries. This can be easily achieved by introducing time-stamps in the signed values, so that only recent (up to a convention) valid signatures are accepted by the verification algorithm. Still, even with this extension, the signature refreshing cost is linear in the number of the current items in the DHT: if m items exist in the system, at any fixed time quantum, the signature refreshing cost is $O(m)$, since all valid pairs need to be resigned. *Signature amortization*, the technique of signing only one, specially designed for the authentication purposes, digest of the entire collection of stored data items, owned by the same source, seem to be the right avenue for achieving efficiency with respect to the overheads due to signature-refreshing. At the same time, we wish our solutions to achieve *load balancing*, that is, to evenly distribute the workload related to authentication across the network nodes. Note that the "sign-all" naive solution does not achieve signature freshness and amortization; however, although inefficient, it achieves load-balancing in a trivial way.

Achieving both load balancing and efficiency in content authentication in p2p systems turns out to be a challenging problem since these two requirements seem to imply contradictory design goals. The primary technique used for achieving efficient authentication is signature amortization, where the data source signs a digest of the entire data set that is computed using a hierarchical hashing structure (e.g., a Merkle tree). However, signature amortization leads to authentication information that is heavily centralized, meaning that the nodes of hashing structure close to the root and the signature over the digest are accessed more often for answer verification. At the same time, p2p systems should achieve network-traffic load-balancing, [5] meaning that data should be accessed without creating hot-spots, i.e., network nodes accessed with skewed, rather than uniform, distribution. Thus, realizing efficient authentication schemes for p2p networks that preserve load-balancing is a particularly challenging task. Note that the "sign-all" naive solution, although inefficient, achieves load-balancing in a trivial way.

In the rest of this section, we present a new, efficient, load-balanced distributed authentication scheme for a DHT with n network nodes. Our scheme is built on top of the basic DHT operation $\mathsf{locate}(k)$, which returns the network node (id) storing the given key k (object identifier). For typical DHT implementations, operation locate takes $O(\log n)$

[5] Load-balance with respect to (uniform) data distribution is another desired property (that is satisfied by the use of a DHT).

time. Our scheme extends the underlying DHT by supporting authenticated versions of operations *proper* and `get` while preserving efficiency and load balancing.

For simplicity, in the following discussion, we assume that a single data source is inserting items in the system. For more data sources, we simply make use of multiple invocations of our scheme (it is easy to see, this is a simple extension). We only require that the public key of each data source storing data in the DHT is known to any entity querying the DHT. Additionally, following the *public parameter model* used in the cryptographic literature, we assume the existence of some *public information* that is associated to each data source and that can be easily accessed and updated independently of the underlying DHT structure. More importantly, the size of this information is only *logarithmic* in the number of data items stored by the source in the system. Note that this public information imposes no limitations in our construction. For instance, in practice this assumption is easily and efficiently implementable through a web service that posts a small amount of data regarding a data source to a web-site.

The performance and security properties of our scheme can be summarized as follows. Given a distributed hash table H with n network nodes such that H supports object location in $O(\log n)$ time using $O(\log n)$ storage overhead per-network node, we can build over H a distributed authentication structure for storing m data items such that:

- The answer to a query can be authenticated with $O(\log n \log m)$ expected time complexity and $O(\log m)$ space complexity.
- The insertion of new data items has $O(\log n \log m)$ expected amortized time complexity.
- The public information has size $O(\log n)$.
- The signature refreshing cost is $O(\log m)$.
- The system is secure against replay attacks.

5.4.1 Hashing Scheme

We next present our new scheme for realizing a distributed authentication structure having two main design goals: *signature amortization*, which will incur low signature refreshing cost, and *load-balancing*, which will not create any hot spots in the underlying p2p network.

The main idea is to use Merkle trees and additionally a technique that uses redundancy such that accessing the Merkle tree can be done in many different ways. By choosing the

particular verification path in a randomized fashion we effectively achieve destroying any hot spots. Naturally enough though, the redundancy used by the new structure slightly increases the storage needs and the computational overhead due to updates (both only by a logarithmic factor). Although our static and semi-dynamic structure achieves optimal space-complexity trade-offs, we still discuss techniques that allows us to make it fully dynamic in reasonable ways.

Our data structure achieves signature amortization by applying a hashing scheme over the data items stored in the DHT. The main idea in our construction is to use a hashing scheme G of high expansion rate (using a structure that resembles the FFT computation graph or a butterfly network), such that for any data element, there exist many equivalent verification paths. We distribute DAG G to the network nodes of the underlying DHT by appropriately indexing the digests and storing them as special data items. We preserve the structure of the hashing scheme G in the DHT as follows: each network node storing the digest of node v in G also stores the keys that can be used to retrieve the digests of the immediate successors and predecessors of v. We then randomize the generation of verification paths to achieve a uniform coverage of the network nodes. Thus, we implement signature amortization and at the same time provide load-balancing.

We build a hashing scheme over m in total data items and distribute this structure to the n network nodes of the underlying DHT. Instead of using a tree for authenticating a data set, we now essentially use m trees that share $m \log m$ tree nodes. In essence, we use a graph structure rather than a tree structure to store (distribute) the data digest.

We now describe our hashing scheme G for m data items and its embedding into a DHT with n network nodes (see Figure 5.3(a)). Let us assume for simplicity (and without loss of generality, as we will see), that $m = 2^k$. The nodes of G are partitioned into $k + 1$ levels, where each level has m nodes. The nodes at level 0 are sources, each associated with a data item. The nodes of the remaining levels have each two predecessors nodes at the previous level. The parent-child relation is defined such that the nodes at level k are the roots of m perfect binary trees over the data set. To formally define the edges of DAG G, let us number the nodes on each level and denote with $v_{i,j}$ the j-th node of G on level i, $i = 0, \ldots, k$, $j = 0, \ldots, m - 1$. For $i > 0$, node $v_{i,j}$ has two incoming edges from nodes $v_{i-1,j}$ and $v_{i-1,j+\delta(i,j)}$, where $\delta(i, j) = (-1)^{\lfloor j/2^{i-1} \rfloor} 2^{i-1}$. Let h be a cryptographic collision-resistant hash function. For $i = 0$, $d(v_{i,j}) = h(k||x)$, where (k, x) is the data item associated with $v_{i,j}$. For $i > 0$, we have $d(v_{i,j}) = h(d(v_{i-1,j}) \;||\; d(v_{i-1,j+\delta(i,j)}))$. By symmetry, the nodes of G at level i store 2^{k-i} distinct digests. The data source signs the single digest stored at nodes of level k and makes it available as public information. Each node is indexed by a unique id,

where node $v_{0,j}$ that is associated with item (k, x) is indexed by k, and is inserted in the DHT using as the key this id and as the value the digest and ids of the predecessors and successors (constant-size information).

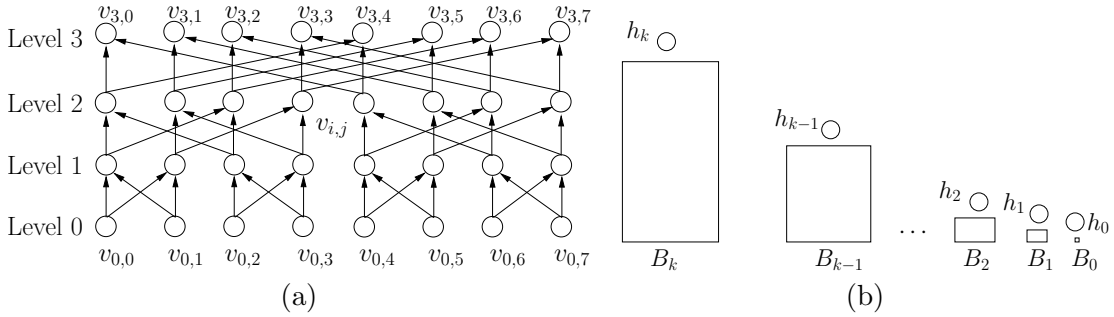

Figure 5.3: (a) The static hashing scheme yielding a single digest of the collection of data items, stored at the top nodes. (b) Dynamization of the hashing scheme, where the static scheme is separately applied to blocks that form a partition of the data items.

5.4.2 Query and Verification

We now describe how get operations are handled. We begin by performing a query according to the underlying DHT structure (e.g., as discussed in previous sections). Given that data element (k, x) stored at network node W is located by the DHT, node W initiates a randomized process for generating a verification path for (k, x). Namely, W flips a coin to determine which of its two parents at level 1 (next node in the path) to contact next (through a location operation, first). In general, a network node V at level j randomly chooses the next network node (to be contacted while forming the verification path) independently and with probability $\frac{1}{2}$. Thus, any query results in a verification path of length $O(\log m)$, using $O(\log m)$ location operations, resulting in $O(\log m \log n)$ computation and communication costs. Through the randomized search process, every verification path for a fixed data element is actually an independent and identically distributed random variable and no hot-spots are created while accessing the authentication structure. The verification path is returned by the DHT and given this, one can authenticate the answer of operation get by processing the digests contained in the path, verifying the publicly available signed digest and checking their consistency. The total storage required is $O(m \log m)$; that is, assuming perfect mapping functions from keys to network nodes (usually through a cryptographic hash function), the storage is logarithmic in n per network node, when $m = O(n)$—i.e., still optimal, since most DHTs use routing tables of logarithmic size. Using a caching technique

as in Section 5.2, we can further improve the creation of the verification paths.

5.4.3 Updates

The data structure described in the previous section is static. To support updates, we modify it using a dynamization technique due to Overmars [109], which allows to transform a static data structure into a corresponding dynamic structure. The idea is to partition a data set of size m into sequence of $O(\log m)$ *blocks*, where the size of each block is twice the size of the previous block, and to completely rebuilt blocks after updates, as necessary. We apply this technique to support insertions of data items with new keys.

Let D be a set of m data items and let $(b_k, b_{k-1}, \ldots, b_1, b_0)_2$ be number m written in binary, with $b_k = 1$. Note that items in D are not assumed to be sorted. [6]. We partition D into $\lfloor \log m \rfloor + 1$ blocks $B_0, B_1, \ldots, B_k$, each a subset of D, according to the weights of the bits of m, i.e., $|B_i| = b_i \cdot 2^i$. Let then $G(i)$, $0 \leq i \leq k$, denote the hashing DAG described in previous section that is built for the elements of block B_i. DAG $G(i)$ has $b_i \cdot 2^i \cdot i$ nodes. DAGs $G(0), G(1), \ldots, G(k)$ are used separately as authentication structures: that is, for $i = 0, \ldots, k$, if $b_i = 1$, the source signs the top-level digest h_i of DAG $G(i)$ (see Figure 5.3(b)) and each $G(i)$ is distributed over the network nodes as before. For any queried data item in block B_i, the corresponding verification path in $G(i)$ is retrieved using $O(i)$ location operations. Accordingly, $O(\log m)$ signed time-stamped digests (one for each block) are made available as public information.

We perform insertions of data items through operations *proper* as follows. Let i be the smallest i such that $b_i = 0$ or $i = k + 1$ if no such i exists. To insert an item x into D, we merge DAGs $G(0), G(1), \ldots, G(i-1)$ to create DAG $G(i)$ for the new block $B_i = B_0 \cup \ldots \cup B_{i-1} \cup x$. Note that $|B_i| = 1 + \sum_{j=0}^{i-1} 2^j = 2^i$. Omitting details, we have that the insertion of a data item into a set of size m stored into a DHT of size n takes $O(\log m \log n)$ expected amortized time. Accordingly, we update the public information: the data source creates new fresh time-stamps and re-signs the publicly available digests. This occurs for all blocks after every update of a block, independently of whether or not the corresponding block structure has been altered in the most recent update. Thus, at any point in time, we maintain $O(\log m)$ fresh signed digests as public information. At asymptotically no additional cost and using similar ideas with the verification of queries, the data source can verify the correctness of an operation *proper* performed by the DHT: any

[6] Recall that any element is located using a particular mapping (usually a hash function). Thus, ordering is not needed for locating an existing element.

change in the hashing scheme is checked for consistency with the $O(\log m)$ signed digests.

We can also support *delayed deletions*, defined in our context as item removals that do not actually occur on-line, but instead occur at some future time and during the insertion of new items. Asymptotically, these deletions incur no additional communication or computational cost. In particular, we schedule the deletion of an item in block B_i during the construction phase of a new DAG $G(j)$, $j > i$, where j depends on the exact state of the authentication structure. This deletion procedure requires minor modifications to the above insertion algorithm. Replay attacks are eliminated by having the data source S performing controlled delayed deletions of items before they are replaced by new items in the system. Moreover, using delayed deletions, our structure supports data item expiration and content revocation: we remove expired or revoked items during the construction of some particular new DAG $G(j)$. In this case, our structure has the following important *self-correction* property that limits the window of opportunity for replay attacks: any expired or revoked item is automatically removed from the structure the first time that the corresponding block containing the item is restructured (rebuilt). Thus, the system supports item expiration/revocation in the sense that no old item can stay forever in the system; in particular, no item can be more than $m/2$ steps old, where m is the current number of items, and depending in the exact application, items can be scheduled to leave the storage system such that no replay-attacks can be launched by the DHT. In conclusion, our authentication structure is efficient and load-balanced and mitigates replay attacks, thus outperforming the "sign-all" insecure solution.

Also, if we allow the insertion of new data items that replace old, then we can work around the replay attack by introducing time-stamps at the data item level, that is as an extra field in the data item. But still, we have the following property: the system naturally supports item expiration in the sense that no item can be more than $m/2$ steps old.

Lemma 5.4.1. *An insertion of a data item in a dynamic graph authentication structure G of size m distributed over a network of size n takes $O(\log m \log n)$ expected amortized time.*

Proof. We do not exactly destroy the graph structures of the blocks that we need to discard after an update, but rather we compose them into the new graph structure $G(j)$. It is easy to see that we need only $O(2^j)$ location operations [7] to update the graph distribution of the graph structure over the network. To facilitate the new block creation, we encode in

[7] If instead we used the naive method to rebuild the new structure from scratch, then we would need $O(j2^j)$ location operations, resulting in $O(\log^2 m \log n)$ total cost.

the hash keys information about the block that they correspond to, and also their level and position in the blocks. Using analysis as in [109], we get that for a structure of size $m = O(2^{k+1})$, the total number of location operations due to insertions is bounded by $\sum_{i=0}^{k} 2^{k-i}O(2^i) = O(k2^k) = O(m \log m)$, which gives an amortized per insertion location cost of $O(\log m)$. A location operation in the underlying DHT takes $O(\log n)$ expected time to perform. □

Overall, we have the following result with respect to our distributed authentication structure. We call *optimal* a distributed hash table over n network nodes where objects are located in $O(\log n)$ time at $O(\log n)$ per-network node storage overhead.

Theorem 5.4.1. *There exists a distributed authentication structure for storing m data items over any optimal distributed hash table of size n, such that:*

- *Stored content is authenticated with $O(\log n \log m)$ expected time complexity and using $O(\log m)$ space complexity.*
- *New data items can be inserted at $O(\log n \log m)$ expected amortized time complexity.*
- *The public information has size $O(\log n)$.*
- *The signature freshness cost is $O(\log m)$.*
- *The system is secure against replay attacks.*

5.5 Authentication of General Queries

Motivated by the our goal, namely to extend the results that we have seen in this chapter to general queries, in this section we study the problem of data authentication from a totally new perspective.

In particular, we study data authentication over structured data—where data is disseminated by issuing queries—from a theoretical point of view. Aiming to general results, we use a very general computational model, the RAM model, and a very general data type and query model, where data is organized according to the relational data model, slightly modified to fit the RAM model. We provide a formal definitional framework for the problem of authenticating answers to queries, identify its inherent relationship with the concept of answer certification and put forward a new approach for solving the problem. Central idea in our work is the following: in contrast to previous general approaches, where the

algorithm that answers a query is essentially being authenticated, we propose an answer-based approach, where, instead, the information necessary for the answer verification is being authenticated. Interestingly, this new approach has connections with certifying algorithms [90], which have been extensively researched. We introduce the related concept of certification data structures, structures that are able to provide answer certification, and we prove a direct connection between them and authenticated data structures, structures that provide authenticated queries but in the bounded computational model. We show that any certification data structure has a corresponding and of the same complexity authenticated data structure. This way, we can exploit the computational gap that has been observed between answering a query or solving a problem and checking or certifying the correctness of the answer or of the computation.

More importantly, we introduce the important concept of (query) problem reducibility in data authentication. Informally, this means that a query of type A is authenticated reduced to query of type B, when an authenticated data structure for B leads to an authenticated data structure for A. We show an important reduction: any query problem in our query model is authenticated reduced to the fundamental set membership problem[8]. This reduction is important as an independent result, since it gives us general possibility results for the design of authenticated data structures. However, this also has an important consequence with respect to the problem of distributed data authentication; overall, we have that for any query problem there exists an efficient distributed authentication structure in the new model introduced in this chapter.

Our results on the design of general authenticated data structures, which are of independent interest, are summarized as follows. We define certification data structures, which extend the concept of certifying algorithms to data structures. We provide a rigorous cryptographic framework for the analysis of authenticated data structures by defining query authentication schemes. Also, we define reductions between query authentication schemes. We present a method for constructing an authenticated data structure from a certification data structure that preserves super-efficient verification. Namely, a certification data structure with verification time asymptotically less than query time yields an authenticated data structure with the same property. We finally show that the authentication of general queries can be reduced to the authentication of set membership queries.

The section is organized as follows. Section 5.5.1 reviews some previous and related work

[8]This is a non-trivial reduction, meaning that efficiency is preserved in our reduction. A trivial reduction authenticates the answer to a query by authenticating all possible query-answer pairs, which for most query problems is a set of infinite cardinality.

on the subject. Section 5.5.2 provides definitions about our query model, Section 5.5.3 describes certification data structures and their properties and Section 5.5.5 provides our definitional framework, introducing query authentication schemes and their security requirements. In Section 5.5.6 we introduce the reducibility among authentication schemes and prove our second main result, namely, that the authentication of any query problem is reduced to the authentication of set-membership problem.

5.5.1 Previous and Related Work

We review some previous and related work on the subject of this section, i.e., authentication over structured data.

General Authentication Techniques. There has been also substantial progress in the design of generic authentication techniques, that is, development of general authentication frameworks that can be used for the design of authenticated data structures for authenticating concrete queries, or design of general authentication patterns that authenticate classes of queries. Work of this type is as follows. In [86] it is described how by hashing over the search structure of data structures in a specific class a broad class we can get authenticated versions of these data structures. The class of data structures is such that (*i*) the links of the structure form a directed acyclic graph G of bounded degree and with a single source node; and (*ii*) queries on the data structure correspond to a traversal of a subdigraph of G starting at the source. The results hold for the pointer machine model of computation, where essentially the entire search algorithm is authenticated. This way, an answer carries a proof that is proportional to the search time spent for generating the answer itself, and the answer verification has analogous time complexity. The method only handles static problems. In [59], it is shown how extensions of hash trees can be used to authenticate abstract properties of data that is organized as paths, where the properties are decomposable, i.e., the properties of two subpaths can be combined to give the property of the resulting path. Also the authentication of the general fractional cascading data-structuring technique [23] is presented. This technique can lead to authentication of data structures that involve iterative searches over catalogs. The underlying model is same as before, i.e., the pointer machine model. Although, the techniques do not explicitly authenticate the corresponding search algorithm, the complexity of the resulted authenticated data structures is of the same order of magnitude as the searching algorithm. Finally, in [108] a general technique is described for designing consistency proofs for committed databases—a different problem than data authentication. However, the technique can be extended to provide a general framework for

designing static authenticated data structures (that actually enjoy additional properties). The authentication technique is similar to the one in [86]: the searching algorithm that is used to produce the answer is authenticated. Also, the used model is the pointer machine; the RAM model can be captured at a $O(\log M)$ overhead, where M is the total memory used. Our results operate on the RAM model, thus, they include a broader class of both static and dynamic query problems and can lead to more efficient constructions, where the answer validity and not the algorithm is verified. Finally, in [142] it is shown that for the dictionary problem and hash-based data authentication, the querying problem and the authentication problem are equivalent. That is, for authenticated dictionaries of size n, all costs related to authentication are at least logarithmic in n in the worst case.

Consistency Proofs and Privacy. Recently, the study of an additional security property related to authenticated data structures has been initiated. Assuming a more adversarial for the user setting, one can consider the case where the data source can act unreliably. The new requirement is then data consistency, namely, the inability of the data source to provide different, i.e., contradictory, verifiable answers to the same query. Buldas *et al.* [17] study this issue for hash trees and show how to enforce data consistency by augmenting hash trees. In [98] zero-knowledge sets are introduced, where a prover commits to a value for a set and membership queries can be verified by a verifier consistently (and in zero-knowledge). In [108] consistency proofs are extended to range queries and where also sufficient conditions are given for schemes to achieve consistency. The works in [98, 108] provide privacy-preserving verification but involve computationally more expensive operations.

Certifying Algorithms and Checking Primitives. Extensive work on certifying algorithms [16, 14, 42, 91, 93, 88] model a computational gap between the computation of a program and the verification of this correctness. This is related with the idea behind our authentication framework. Our methodology to decouple the searching algorithm from the answer verification is modeled through a certification data structure, defined in Section 5.5.3, which can be viewed as an extension of methodology of the certifying algorithms for data structures. Related also work appears in [8, 9, 53, 104].

5.5.2 Preliminaries

We start by defining our data querying model, which is based on the RAM model of computation.

Definition 5.5.1 (Structured Data Set)**.** *A* structured data set *(or, simply, a* data set*)* $S = (\mathcal{E}, \mathcal{R})$ *consists of: (i) a collection* $\mathcal{E} = \{E_1, \ldots, E_t\}$ *of sets of data elements such that, for* $1 \leq i \leq t$*, set* E_i *is a subset of a* universe $\mathcal{U}_i$*, and (ii) a collection* $\mathcal{R} = \{R_1, \ldots, R_k\}$ *of indexed sequences of tuples of data elements such that, for* $1 \leq i \leq k$*, sequence* $R_i = (R_i[1], \ldots, R_i[m_i])$ *consists of* m_i *distinct* p_i*-tuples from* $E_{j_1} \times \ldots \times E_{j_{p_i}}$*, where* $1 \leq j_1 \leq \ldots \leq j_{p_i} \leq t$ *and* $p_i < p$ *for some integers* p*,* m_i*. The* size n *of data set* $S = (\mathcal{E}, \mathcal{R})$ *is defined as* $n = \sum_{i=1}^{t} |E_i|$*. Also, we assume that* t*,* k *and* p *are constants (with respect to* n*).*

Our definition shares concepts from the relational data model for databases (see, e.g., [73]). A relation, mathematically defined as a subset of the Cartesian product of sets, is typically viewed as a set of tuples of elements of these sets. Our model uses instead indexed sequences of tuples. Namely, each member R_i of $\mathcal{R}$ is an array of tuples, where each tuple can be indexed by an integer. In this way, our model achieves generality: on one hand, we can express very general data organization paradigms, on the other hand, we capture algorithms and data structures in the RAM model of computation. For instance, a graph $G = (V, E)$ may correspond to data set $S_G = (\mathcal{E}, \mathcal{R})$, where $\mathcal{E} = V$ and $\mathcal{R}$ consists of a single sequence of indexed pairs representing relation E (edges in G). More complex graphs, e.g., with edge directions, weights, costs or associated data elements, can be represented by appropriately including new primitive data-element sets in $\mathcal{E}$ and corresponding sequences in $\mathcal{R}$ describing data elements' structure and various relations among them.

Definition 5.5.2 (Querying Model)**.** *Let* $S = (\mathcal{E}, \mathcal{R})$ *be a structured data set. A* query operation Q_S *on* S *is a computable function* $Q_S : \mathcal{Q} \rightarrow \mathcal{A}_S$*, where* $\mathcal{Q}$ *is the query space (the set of all possible queries* q *of specific type that can be issued about* S*) and* $\mathcal{A}_S$ *is the answer space (the set of all possible answers to queries on* S *drawn from* $\mathcal{Q}$*). The* answer *of a query* $q \in \mathcal{Q}$ *under* Q_S *is* $Q_S(q) \in \mathcal{A}_S$*. An element* $a \in \mathcal{A}_S$ *of the answer space is the* correct answer *for query* q *if and only if* $Q_S(q) = a$*.*

Observe that the above definitions capture general query operations[9] on data sets that are based on relations. The only requirement is that any query in the query space is mapped to a unique answer in the answer space and that any answer corresponds to some query[10]. For instance, if $S_G = (\mathcal{E}, \mathcal{R})$ represents a monotone subdivision of the plane into

[9] Alternatively but less conveniently, query operation Q_S can be defined independently of the data set S, such that the answer to query q is $Q(S, q)$. In this case, the query and answer spaces are also independent of S.

[10] Unique answers are used without loss of generality. Of course, there are query problems for which Q_S is a mapping not a function. That is, more that one answers can exist for a given query. For instance, a path query on a graph, given two vertices asks for any connecting path, if it exists. We can appropriately

the polygons induced by the vertices and edges of a planar graph G, the point location query operation maps a point in the plane (query) to the unique region of the subdivision (answer) containing it. Regarding the complexity of query answering, we only require that query operation Q_S is efficiently computable. Typically, function Q_S is evaluated on query $q \in \mathcal{Q}$ by a query answering algorithm that operates over S through an appropriate for the type of queries in $\mathcal{Q}$ query data structure.

Definition 5.5.3 (Query Data Structure)**.** *A* query data structure $D(Q_S)$ *for query operation* $Q_S : \mathcal{Q} \to \mathcal{A}_S$ *on data set* $S = (\mathcal{E}, \mathcal{R})$ *consists of a structured data set* $(\mathcal{E}_Q, \mathcal{R}_Q)$*, such that* $\mathcal{E} \subset \mathcal{E}_Q$ *and* $\mathcal{R} \subset \mathcal{R}_Q$ *and an algorithm* Answer*, which on input a query* $q \in \mathcal{Q}$ *and data set* $(\mathcal{E}_Q, \mathcal{R}_Q)$ *returns* $Q_S(q) \in \mathcal{A}_S$ *in time polynomial in* n *and* $|q|$ *by accessing and processing tuples in* $\mathcal{R}$*.*[11] *We write* $D(Q_S) = (\mathcal{E}_Q, \mathcal{R}_Q, \mathsf{Answer})$*.*

On input query q, algorithm Answer operates over S through the use of $D(Q_S)$: by processing relations in $\mathcal{R}_Q$, Answer accesses relations in $\mathcal{R}$, evaluates conditions over elements in S and produces the answer. For instance, for a point location algorithm that is based on segment trees and operates on planar subdivision $S_G = (\mathcal{E}, \mathcal{R})$, data set $(\mathcal{E}_Q, \mathcal{R}_Q)$ represents a two-level search structure locating points in logarithmic time; here, data set S_G includes information about the regions defined by the edges of graph G.

A data set S is *static* if it stays the same over time and *dynamic* if it evolves over time through *update operations* performed on S. An update operation U_S for S is a function that given an update $y \in \mathcal{Y}$, where $\mathcal{Y}$ is the set of all possible updates, results in changing one or more data elements in $\mathcal{E}$ and accordingly one or more tuples in $\mathcal{R}$. If S is static (resp. dynamic), data set $(\mathcal{E}_Q, \mathcal{R}_Q)$ can be constructed (resp. updated) by some algorithm Constr_Q (resp. Update_Q) that runs on input S (resp. S and $y \in \mathcal{Y}$) in polynomial time in n.

Our data querying model achieves generality by combining the expressiveness of relational databases with the power of the RAM computation model. By using index-annotated relations, complex data organizations are easily represented and accessed. For instance, indirect addressing is supported by treating indexes as a distinct data type which is included in $\mathcal{E}$, thus our model strictly contains the pointer machine model.

We next review the cryptographic primitives that we will use beyond signatures and hash functions (for these see, e.g., Section 4.2).

---

augment the query space for this type of queries to include the index of the answer (according to some fixed ordering) that we wish to obtain.

[11] By Definition 5.5.1, for any data set $(\mathcal{E}, \mathcal{R})$ of size n, the total number of relations that exist in $\mathcal{R}$ (and thus can be possibly accessed by Answer) is $O(n^p) = poly(n)$. This implies that the storage needed for data set $(\mathcal{E}, \mathcal{R})$ is related to its size according to a polynomial.

The following cryptographic primitive is based on the Merkle hash tree [95].

Definition 5.5.4 (Hash Tree)**.** *For a set of n elements a hash tree is a binary tree, where each node stores a hash value computed using a collision-resistant hash function. At leaf nodes the hash of the corresponding element is stored; at internal nodes the hash of the concatenation of the hash values of the children nodes.*

Finally, we review dynamic accumulators. We here use a standard definition similar to the one in [18].

Definition 5.5.5 ((One-way) Dynamic Accumulator)**.** *An accumulator for a family of inputs $\{\mathcal{X}_k\}$ is a family of families of functions $\mathcal{G} = \{\mathcal{F}_k\}$ with the following properties.*

Efficient Generation *There is an efficient algorithm Gen that on input 1^κ generates a random element f of $\mathcal{F}_k$, an auxiliary information aux_f and a trapdoor information trd_f. Both aux_f and trd_f have sizes that are linear in k.*

Efficient Evaluation *Function f is a computable function $f : \mathcal{A}_f \times \mathcal{X}_k$, where $\mathcal{A}_f$, a set of accumulation values that can be efficiently sampled and $\mathcal{X}_k$, the proposed set of elements to be accumulated, constitute the input domain of f. Function f is polynomial-time computable given the auxiliary information aux_f.*

Quasi-Commutativity *For all $f \in \mathcal{F}_k$, $a \in \mathcal{A}_f$ and $x_1, x_2 \in \mathcal{X}_k$, it holds that*

$$f(f(a, x_1), x_2) = f(f(a, x_2), x_1).$$

Witnesses *Let $a \in \mathcal{A}_f$ and $x \in \mathcal{X}_k$. A value $w \in \mathcal{A}_f$ is called a* witness *for x in a, under f, if $a = f(w, x)$.*

Updates *Let $X \subset \mathcal{X}_k$, $x \in X$, $a_0, a_X, w \in \mathcal{A}_f$, such that $f(a_0, X) = f(w, x) = a_X$. Let $OP = \{insert, delete\}$ be the set of update operations on set X, such that $insert(\bar{x}) = X \cup \{\bar{x}\}$, $\bar{x} \in \mathcal{X}_k - X$ and $delete(\bar{x}) = X - \{\bar{x}\}$, $\bar{x} \in X$. An one-way accumulator is dynamic if there exist efficient algorithms U_{op}, W_{op}, $op \in OP$, such that:*

- *$U_{op}(trp_f, a_X, \bar{x}) = a_{\bar{X}} \in \mathcal{A}_f$ such that $a_{\bar{X}} = f(a_0, op(\bar{x}))$, that is $a_{\bar{X}} = a_{X \cup \bar{x}}$ or $a_{\bar{X}} = a_{X - \bar{x}}$,*
- *$W_{op}(f, aux_f, a_X, a_{\bar{X}}, x, \bar{x}) = w' \in \mathcal{A}_k$ such that $a_{\bar{X}}$ is as above and $a_{\bar{X}} = f(w', x)$.*

Security *An accumulator is one-way (secure) if the following holds true. Let $\mathcal{A}'_f \times \mathcal{X}'_k$ denote the domains for which the computational procedure for function $f \in \mathcal{F}_k$ is*

defined. That is, in principle, $\mathcal{A}'_f \supseteq \mathcal{A}_f$ and $\mathcal{X}'_k \supseteq \mathcal{X}_k$. For all probabilistic polynomial-time adversaries Adv_k

$$\Pr[f \leftarrow Gen(1^k); a_0 \leftarrow \mathcal{A}_f; (x, w, X) \leftarrow Adv_k(f, aux_f, \mathcal{A}_f, a_0) : X \subset \mathcal{X}_k; w \in \mathcal{A}'_f; x \in \mathcal{X}'_k; x \notin X; f(w, x) = f(a_0, X)] = \nu(k).$$

5.5.3 Certification Data Structures

In this section, we explore the decoupling of query answering and answer validation (and, accordingly, answer verification). We start by defining the notion of *answer testability*, formally expressed through a *certification data structure*. Intuitively, this notion captures the following important property in data querying: query operations on any data set return validated answers, that is, answers that can be tested to be correct given a (minimal) subset of specially selected relations over elements of the data set. In essence, queries are certified to return valid answers; actually this holds in a safe way (i.e., cheating is effectively disallowed).

Definition 5.5.6 (Certification Data Structure)**.** *Let $D(Q_S) = (\mathcal{E}_Q, \mathcal{R}_Q, \mathsf{Answer})$ be a query data structure for query operation $Q_S : \mathcal{Q} \rightarrow \mathcal{A}_S$ on data set $S = (\mathcal{E}, \mathcal{R})$ of size n. A* certification data structure *for S with respect to query data structure $D(Q_S)$ is a triplet $C(Q_S) = ((\mathcal{E}_C, \mathcal{R}_C), \mathsf{Certify}, \mathsf{Verify})$, where $(\mathcal{E}_C, \mathcal{R}_C)$, called the* certification image *of S, is a structured data set and* Certify *and* Verify *are algorithms such that:*

Answer tests: *On input query $q \in \mathcal{Q}$ and data sets $(\mathcal{E}_Q, \mathcal{R}_Q)$ and $(\mathcal{E}_C, \mathcal{R}_C)$, algorithm* Certify *returns answer $a = Q_S(q)$ and an* answer test *τ, which is a sequence of pairs (i, j), each indexing a tuple $R_i[j]$ of $\mathcal{R}_C$. Answer test τ defines a subset $\mathcal{R}_C(\tau) \subseteq \mathcal{R}_C$, called the* certification support *of answer a.*

Answer testability: *On input query $q \in \mathcal{Q}$, data set $(\mathcal{E}_C, \mathcal{R}_C)$, answer $a \in \mathcal{A}_S$ and answer test τ, algorithm* Verify *accesses and processes* only *relations in $\mathcal{R}_C(\tau)$ and returns either* 0 *(rejects) or* 1 *(accepts).*

Completeness: *For all queries $q \in \mathcal{Q}$, it holds that*

$$\mathsf{Verify}(q, \mathcal{R}_C, \mathsf{Certify}(q, (\mathcal{E}_Q, \mathcal{R}_Q), (\mathcal{E}_C, \mathcal{R}_C))) = 1.$$

Soundness: *For all queries $q \in \mathcal{Q}$, answers a, answer tests τ, when* $\mathsf{Verify}(q, \mathcal{R}_C, a, \tau) = 1$, *$a = Q_S(q)$.*

Regarding complexity measures for certification data structure $C(Q_S)$, we say:

- $C(Q_S)$ *is* answer-efficient *if the time complexity $T_C(n)$ of* Certify *is asymptotically at most the time complexity $T_A(n)$ of* Answer*, i.e., $T_C(n)$ is $O(T_A(n))$;*

- $C(Q_S)$ *is* time-efficient *(resp.* time super-efficient*) if the time complexity $T_V(n)$ of* Verify *is asymptotically at most (resp. less than) the time complexity $T_A(n)$ of* Answer*, i.e., $T_V(n)$ is $O(T_A(n))$ (resp. $o(T_A(n))$); and analogously,*

- $C(Q_S)$ *is* space-efficient *(resp. space super-efficient) if the space requirement $S_C(n)$ of $(\mathcal{E}_C, \mathcal{R}_C)$ is asymptotically at most (resp. less than) the space requirement $S_Q(n)$ of $(\mathcal{E}_Q, \mathcal{R}_Q)$, i.e., $S_C(n)$ is $O(S_Q(n))$ (resp. $o(S_Q(n))$). If S is static, data set $(\mathcal{E}_C, \mathcal{R}_C)$ can be constructed by some algorithm* Constr_C *that runs on input S in polynomial time in n.*

For simplicity, the above definition corresponds to the static case. The dynamic case can be treated analogously. Informally, an update algorithm Update_C is responsible to handle updates in data set S by accordingly updating $C(Q_S)$; that is, it produces the updated set $(\mathcal{E}'_C, \mathcal{R}'_C)$ and, in particular, the set of tuples where $\mathcal{R}'_C$ and $\mathcal{R}_C$ differ at. Algorithm Update_C additionally produces an *update test* (as the answer test above, a set of indices for tuples in $\mathcal{R}_C$) that validates the performed changes. Similarly, an update testing algorithm Updtest, on input an update $y \in \mathcal{Y}$, set $\mathcal{R}_C$, a set of tuples (changes in $\mathcal{R}_C$) and an update test, accepts if and only if the tuples correspond to the correct, according to y, new or deleted tuples in $\mathcal{R}_C$. Similarly, we can define *update efficiency* and *update-testing (super-)efficiency* for $C(Q_S)$, with respect to the time complexity of Update_C and Updtest respectively, as they asymptotically compare to Update_Q.

Certification data structures introduce a general framework for studying data querying with respect to the answer validation and correctness verification. They support certification of queries in a computational setting where the notions of query answering and answer validation are conceptually and algorithmically separated in a clean way. In particular, answer validation is based *merely* on the certification image $(\mathcal{E}_C, \mathcal{R}_C)$ of data set $S = (\mathcal{E}, \mathcal{R})$; the two data sets are related by sharing tuples, possibly, through a subset relation. Also, query certification depends *only* on the certification support of the answer, i.e., subset $\mathcal{R}(\tau)$.

Our first result shows that for every query structure there is an efficient certification structure, that is, a completeness result showing that all queries can be certified without loss of efficiency.

Lemma 5.5.1. *Any query data structure for any query operation on any structured data set admits an answer-, time-, update-, update-testing- and space-efficient certification data structure.*

Proof. We first discuss the static case. Let $S = (\mathcal{E}, \mathcal{R})$ be a structured data set of size n and $Q_S : \mathcal{Q} \to \mathcal{A}_S$ be a query operation on S. Let $D(Q_S) = (\mathcal{E}_Q, \mathcal{R}_Q, \mathsf{Answer})$ be a query data structure for Q_S. We now describe a certification data structure $C(Q_S) = ((\mathcal{E}_C, \mathcal{R}_C), \mathsf{Certify}, \mathsf{Verify})$ for S with respect to query data structure $D(Q_S)$. First we set $(\mathcal{E}_C, \mathcal{R}_C) = (\mathcal{E}_Q, \mathcal{R}_Q)$. Algorithm Certify is an augmented version of Answer. Given a query $q \in \mathcal{Q}$ and sets $(\mathcal{E}_C, \mathcal{R}_C)$, $(\mathcal{E}_Q, \mathcal{R}_Q)$, Certify creates an empty sequence τ of indices of tuples in $\mathcal{R}_C$ and then it runs Answer on input $(q, (\mathcal{E}_Q, \mathcal{R}_Q))$ to produce the answer $Q_S(q)$. Also, any time algorithm Answer accesses a tuple $R_i[j]$ in $\mathcal{R}_Q$, algorithm Certify adds (i, j) to the end of sequence τ. When Answer terminates, so does Certify, and returns the output $a = Q_S(q)$ produced by Answer and sequence τ as the corresponding answer test.

We define algorithm Verify as an augmentation of Answer operating as follows. On input a query $q \in \mathcal{Q}$, set $(\mathcal{E}_C, \mathcal{R}_C)$, an answer a and a sequence τ, algorithm Verify starts executing algorithm Answer on input $(q, (\mathcal{E}_Q, \mathcal{R}_Q))$ and checks the execution of Answer subject to sequence τ. That is, each time Answer retrieves a tuple $R_i[j]$ in $(\mathcal{E}_Q, \mathcal{R}_Q)$, Verify removes the first element of τ and compares it to (i, j), rejecting the input if the comparison fails. When Answer terminates, the answer computed by Answer is compared with the answer provided as input: if the two answers agree (are equal) then Verify accepts its input, otherwise it rejects.

We now show that the completeness and soundness conditions are satisfied. Completeness is easily seen to hold, since the tuple-access trail of the same—correctly implementing query operation Q_S—algorithm Answer on executions of the same input is tested by algorithm Verify. Thus, we are guaranteed that Certify reports the correct for its input query answer and an answer test that when feeds the computation of Verify does not lead to rejection. With respect to soundness, we easily see that this requirement also holds: when algorithm Verify accepts on input $(q, \mathcal{R}_C, a, \cdot)$, then it is always the case that $a = Q_S(q)$. Indeed, when operating on the valid data set and on input q, algorithm Answer returns the unique, correct answer for q. Finally, it is easy to see that our certification data structure is answer-, time- and space-efficient. This follows from the fact that for any inputs Certify and Verify do a total amount of work that is only by a constant factor more than the work of Answer, thus $T_C(n) = O(T_A(n))$ and $T_V(n) = O(T_A(n))$, and the fact that $(\mathcal{E}_C, \mathcal{R}_C) = (\mathcal{E}_Q, \mathcal{R}_Q)$, thus $S_C(n) = O(S_Q(n))$. Observe that each pair (i, j) in the answer

test τ is accessed in constant time.

The dynamic case is treated analogously. This time instead of the query answering algorithm Answer, we augment the update algorithm Update_Q of the query data structure to define the update and the update testing algorithms, Update_C and Updtest respectively, of certification data structure $C(Q_S)$. The completeness, soundness and complexity properties hold in a similar way as in the static case. □

In addition to showing that Definition 5.5.6 is meaningful, this result proves the generality and feasibility of answer testability for any computable query in a very general querying and computational model (relation-based data queried on RAM machines) . We believe that this offers a new characterization of data querying algorithms and a useful framework for the design and analysis of such algorithms. Moreover, a certification data structure naturally finds application in the Server-Client model of computation, extending data querying in the following setting. A server maintains a data set $S = (\mathcal{E}, \mathcal{R})$ and a client issues queries about S, where the server (prover) cooperates with the client (verifier) and the certification structure merely allows the client to validate the answer computed by the server. By allowing the client to maintain the certification image of S and considering a dynamic setting, we achieve an extension where the client also checks the validity of updates. For instance, one application of the above functionalities is database outsourcing, where the client owns data set S and checks the *consistency* of S by maintaining only the certification image of S.

Relation to Certifying Algorithms. Our certification data structures are related to, and inspired by, certified algorithms (see, e.g., [75, 90]). Both model the property of answer testability (of a program or an algorithm for a data structure) as distinct from algorithm execution. The main difference, though, is that here we model the intrinsic property of a data structure to provide proof of correctness for verification (and authentication, as we will see) purposes. Certifying algorithms are designed to guard against an erroneous implementation of an algorithm. Thus, we can view Definition 5.5.6 as an extension of the theory of certifying algorithms for data structures. Indeed, certifying algorithms for data structures use the implementation of a data structure as a black box and add a wrapper data structure to catch errors. Instead, in Definition 5.5.6, the data structure is augmented to facilitate the certification process.

5.5.4 Time Super-Efficient Certification Data Structures

In this section, we describe examples of time super-efficient certification data structures, further justifying the importance of the notion of answer testability. For time super-efficient certification structures, although the certification image may be as large as the query structure, the certification support of the answer to any query has size asymptotically less than the "searching trail" of the query answering algorithm Answer. In this case, a super-efficient certification data structure exploits this gap in certifying queries.

A very simple case is the dictionary problem, where $S = (\mathcal{E}, \mathcal{R})$ is an ordered key-value set of size n: $\mathcal{E}$ is a set K of n key elements with a totally ordering and a set V of n values, and $\mathcal{R}$ consists of two indexed relations, the key-value relation R_{KV} and the successor relation R_S over keys. The query operation Q_S has query space the universe that key elements are drawn from and answer space the set of all possible key-value pairs; to any query (key) q, Q_S maps the answer (key-value pair) (k, v) if $q = k$, $q \in K$ and $(k, v) \in R_{KV}$ (v is the value of k), or the answer $\perp$ (denoting negative membership answer) if no such condition is satisfied. Consider any search tree that implements the dictionary query data structure. Then the set $(\mathcal{E}_Q, \mathcal{R}_Q)$ of a query data structure is an augmentation of $(\mathcal{E}, \mathcal{R})$, for instance $\mathcal{E}_Q$ now includes tree nodes and pointers, or $\mathcal{R}_Q$ now includes the node-data and parent-child relations. There exists a time super-efficient (and space-efficient) certification data structure for the dictionary problem. Set $(\mathcal{E}_C, \mathcal{R}_C)$ is simply $(\mathcal{E}, \mathcal{R})$. On input a query q, algorithm Certify returns as an answer test the indices in $\mathcal{R}_C$ of two tuples: if $q \in K$, the indices of tuple $< q, suc(q) >$ of the successor relation R_S and of tuple $< q, v >$ of the key-value relation R_{KV} are returned, otherwise, the indices of tuples $< x, suc(x) >, < y, suc(y) > \in R_S$, such that x is the maximum element and y is the minimum element satisfying $x < q < y$, according to the total order of K. Algorithm Verify, accesses these tuples and accepts or rejects accordingly. For instance, if $a = \perp$ and the indices of two tuples $< x, suc(x) >, < y, suc(y) >$ of the successor relation are in answer test τ, then it accepts if $x < q < y$ and $suc(x) = y$; Verify rejects in all other cases. It is easy to see that the completeness and soundness conditions hold. We note that the soundness property depends on both the answer testing algorithm Verify and on the certification image $(\mathcal{E}_C, \mathcal{R}_C)$. For instance, although a different (than the successor) relation could satisfy the completeness property, this choice may not satisfy soundness. For instance, the "odd-rank-difference" relation (two keys have ranks in the sorted set $\mathcal{E}$ with odd difference), which includes the successor relation, satisfies only the completeness condition. Note that $T_V(n) = O(1)$ although $T_A(n) = O(\log n)$; also $S_C(n) = O(S_Q(n)) = O(n)$. The dynamic

extension of this certification data structure is straightforward. We note that the successor relation can be used to support in a very similar way a time super-efficient certification data structure for one-dimensional range searching.

Also, consider the point location problem, where we ask to find the region of a planar subdivision of size n containing a given query point. Using existing efficient point-location algorithms (e.g., [119]), point location queries can be answered in time $O(\log n)$. A time super-efficient certification data structure stores the trapezoidal decomposition of the subdivision. Each trapezoid is expressed as a tuple of five data elements: two vertices (defining the top and bottom sides), two edges (defining the left and right sides), and a region (containing the trapezoid). The answer test is the index of the trapezoid containing the query point, which can be computed by a simple modification of the query algorithm. The inclusion of the answer point in the answer test trapezoid is tested in $O(1)$ time. That is, again, $T_V(n) = O(1)$ although $T_A(n) = O(\log n)$. This certification data structure has also a dynamic extension. Additional examples include data structures for other geometric problems (e.g., convex hull).

5.5.5 Authenticated Data Structures

In this section, we formally describe a general model for data authentication in untrusted and adversarial environments by introducing *query authentication schemes*, cryptographic primitives (algorithms that use cryptography to satisfy certain properties) for the authentication of general queries over collections of structured data. Conceptually, query authentication schemes extend certification structures in that answer validation is not performed in a collaborative setting; instead, the prover may be adversarial and answer verification is now achieved in the bounded computational model. In particular, we examine data authentication in a non-conventional setting, where the creator (or owner) of a data set is not the same entity with the one answering queries about the set and, in particular, the data owner does not control the corresponding data structure that is used to answer a query. In this setting, an intermediate, untrusted party answers the queries about the data set that are issued by an end-user. We formally define this model of data querying.

Definition 5.5.7 (Three-Party Data Querying Model)**.** *A* three-party data querying model *consists of a* source $\mathcal{S}$*, a* responder $\mathcal{R}$ *and a* user $\mathcal{U}$*, where:*

1. *Source $\mathcal{S}$ creates (and owns) a dynamic data set S, which is maintained by query data structure $D(Q_S)$ for query operation $Q_S : \mathcal{Q} \rightarrow \mathcal{A}_S$ on S.*

2. *Responder $\mathcal{R}$ stores S, by maintaining a copy of $D(Q_S)$ and some auxiliary information $aux(S)$ for S.*

3. *User $\mathcal{U}$ issues queries about S to responder $\mathcal{R}$ by sending to $\mathcal{R}$ a query $q \in \mathcal{Q}$.*

4. *On a query $q \in \mathcal{Q}$ issued by $\mathcal{U}$, $\mathcal{R}$ computes answer $a = Q_S(q)$ and sends a to $\mathcal{U}$.*

5. *On an update $y \in \mathcal{Y}$ for S issued by the source, S and $D(Q_S)$ are appropriately updated by $\mathcal{S}$ and $\mathcal{R}$.*

The model achieves generality and has many practical applications. Regarding data authentication, we wish that the user can verify the validity of the answer given to him by the responder. For this verification process, we wish that the responder, along with the answer, gives to the user a proof that can be used in the verification. To capture this verification feature, we define the notion of a *query authentication scheme*.

Definition 5.5.8 (Query Authentication Scheme)**.** *A* query authentication scheme *for query operation $Q_S : \mathcal{Q} \to \mathcal{A}_S$ on structured data set S is a quadruple of PPT algorithms* $(\mathsf{KeyG}, \mathsf{Auth}, \mathsf{Res}, \mathsf{Ver})$ *such that:*

Key generation *The key generation algorithm* KeyG *takes as input a security parameter 1^κ, and outputs a key pair (PK, SK). We write $(PK, SK) \leftarrow \mathsf{KeyG}(1^\kappa)$.*

Authenticator *The authenticator algorithm* Auth *takes as input the secret and public key (SK, PK), the query space $\mathcal{Q}$ (or an encoding of the query type) and data set S of size n and outputs an authentication string α and a verification structure V, that is $(\alpha, \mathsf{V}) \leftarrow \mathsf{Auth}(SK, PK, \mathcal{Q}, S)$, where $\alpha, \mathsf{V} \in \{0,1\}^*$.*

Responder *The responder algorithm* Res *takes as input a query $q \in \mathcal{Q}$, a data set S of size n and a verification structure $\mathsf{V} \in \{0,1\}^*$ and outputs an answer-proof pair $(a, p) \leftarrow \mathsf{Res}(q, S, \mathsf{V})$, where $a \in \mathcal{A}_S$ and $p \in \{0,1\}^*$.*

Verifier *The verifier algorithm* Ver *takes as input the public key PK, a query $q \in \mathcal{Q}$, an answer-proof pair $(a, p) \in \mathcal{A}_S \times \{0,1\}^*$ and an authentication string $\alpha \in \{0,1\}^*$ and either accepts the input, returns 1, or rejects, returns 0, that is, we have that $\{0,1\} \leftarrow \mathsf{Ver}(PK, q, (a, p), \alpha)$.*

Updates *For the dynamic case, we additionally require the existence of an update algorithm* Auth_U *that complements algorithm* Auth *and handles updates. In particular,* Auth_U *given update $y \in \mathcal{Y}$, it updates the authentication string and the verification structure: $(\alpha', \mathsf{V}') \leftarrow \mathsf{Auth}_U(SK, PK, \mathcal{Q}, S, y, \alpha, \mathsf{V})$.*

We now define the first requirement for a query authentication scheme, which is *correctness*. Intuitively, we wish the verification algorithm to accept answer-proof pairs generated by the responder algorithm and these answers always to be correct. We also discuss the *security* requirement of any query authentication scheme. Starting from the basis that in our three-party data querying model, the user $\mathcal{U}$ trusts the data source $\mathcal{S}$ but not the responder $\mathcal{R}$, it is the responder that can act adversarially. We first assume that $\mathcal{R}$ always participates in the three-party protocol, i.e., it communicates with $\mathcal{S}$ and $\mathcal{U}$, as the protocol dictates. Thus, we do not consider denial-of-service attacks; they do not form an authentication attack but rather a data communication threat. However, $\mathcal{R}$ can adversarially try to cheat, by not providing the correct answer to a query and forging a false proof for this answer. Accordingly, the security requirement is that given any query issued by $\mathcal{U}$, no computationally bounded $\mathcal{R}$ can reply with a pair of answer and an associated proof, such that both the answer is not correct and $\mathcal{U}$ verifies the authenticity of the answer and, thus, accepts it. The above requirements are expressed as the following two conditions for query authentication structures.

Definition 5.5.9 (Correctness)**.** *A query authentication scheme* $(\mathsf{KeyG}, \mathsf{Auth}, \mathsf{Res}, \mathsf{Ver})$ *is* correct *if for all queries* $q \in \mathcal{Q}$*, if* $(\alpha, \mathsf{V}) \leftarrow \mathsf{Auth}(SK, PK, \mathcal{Q}, S)$ *and additionally* $(a, p) \leftarrow \mathsf{Res}(q, S, \mathsf{V})$*, then with overwhelming probability it holds that* $1 \leftarrow \mathsf{Ver}(PK, q, (a, p), \alpha)$ *and* $Q_S(q) = a$*.*

A query authentication scheme $(\mathsf{KeyG}, \mathsf{Auth}, \mathsf{Res}, \mathsf{Ver})$ for query operation $Q_S : \mathcal{Q} \rightarrow \mathcal{A}_S$ on structured data set S is said to be *secure*, if no probabilistic polynomial-time adversary $\mathcal{A}$, given any query $q \in \mathcal{Q}$, the public key PK and oracle access to the authenticator algorithm Auth, can output an authentication string α, an answer a' and a proof p', such that a' is an incorrect answer that passes the verification test, that is, $a' \neq Q_S(q)$ and $1 \leftarrow \mathsf{Ver}(PK, q, (a', p'), \alpha)$. More formally:

Definition 5.5.10 (Security)**.** *A query authentication scheme* $(\mathsf{KeyG}, \mathsf{Auth}, \mathsf{Res}, \mathsf{Ver})$ *is* secure *if no probabilistic polynomial-time adversary* $\mathcal{A}$ *can win non-negligibly often in the following game:*

1. *A key pair is generated:*
$$(PK, SK) \leftarrow \mathsf{KeyG}(1^{\kappa}).$$
2. *The adversary* $\mathcal{A}$ *is given:*
   - *The public key* PK *as input.*

- *Oracle access to the authenticator, i.e., for* $1 \leq i \leq poly(k)$*, where* $poly(\cdot)$ *is a polynomial, the adversary can specify a structured data set* S_i *of size* n *and obtain* $(\alpha_i, \mathsf{V}_i) \leftarrow \mathsf{Auth}(SK, PK, \mathcal{Q}, S_i)$*. However, the adversary cannot issue more than one query with the data set* S_i*. That is, for all* $i \neq j$*,* $S_i \neq S_j$*.*
- *A query* $q \in \mathcal{Q}$*.*

3. At the end, $\mathcal{A}$ *outputs an authentication string* α*, an answer* a' *and a proof* p*.*

The adversary wins *the game if the following violation occurs:*

Violation of the security property: *The adversary did manage to construct an authentication string* α *in such a way, that given a query* $q \in \mathcal{Q}$*, the adversary outputs an incorrect answer-proof pair* (a', p') *that passes the verification test. Namely, the adversary wins if* one *of the following hold:*

- *The authenticator was never queried with* S *and yet the verification algorithm does not reject, i.e.,* $1 \leftarrow \mathsf{Ver}(PK, q, (a', p'), \alpha)$*.*
- *The authenticator was queried with* S *and yet* $a' \neq Q_S(q)$ *and the verification algorithm accepts, i.e.,* $1 \leftarrow \mathsf{Ver}(PK, q, (a', p'), \alpha)$*.*

Definition 5.5.11 (Authenticated Data Structure)**.** *An* authenticated data structure *for queries in query space* $\mathcal{Q}$ *on a data set* S *is a* correct *and* secure *query authentication scheme* $(\mathsf{KeyG}, \mathsf{Auth}, \mathsf{Res}, \mathsf{Ver})$*, or, as it is implied, a scheme where, given an authentication string* α*, for algorithm* Ver *it holds that, for all queries* $q \in \mathcal{Q}$*, with all but negligible probability (measured over the probability space of the responder algorithm):*

$$Q_S(q) = a \text{ if and only if there exists } p \text{ s.t. } 1 \leftarrow \mathsf{Ver}(PK, q, (a, p), \alpha).$$

5.5.6 Authentication Reductions and General Authentication Results

We are now ready to use the definitional framework of the previous sections and describe and prove the main results of our work. The road map is as follows. First we introduce the notion of reducibility in data authentication, namely by defining reductions between query authentication schemes. We then prove, using our framework of certification data structures, that the authentication of any query in our model is reduced to the authentication of *set membership* queries. In fact, we need to authenticate only positive answers—that is, relation $\in$ and not $\notin$ needs to be authenticated. We then present implications of this result, in terms of concrete constructions. Using certification structures, we provide a general methodology

for constructing correct and secure query authentication schemes and we show that any search structure for any query type in our querying model can be transformed into an authenticated data structure. Also, based on super-efficient query certification, we develop a new approach for data authentication, where only the information necessary for the answer verification is authenticated, and not the entire information used by search algorithm, which leads to a powerful framework for the design of authentication structures with super-efficient verification.

Let $QAS(Q_S, S)$ denote a query authentication scheme (or QAS) for query operation Q_S and data set S. Intuitively, authenticated reductions among QASs allow the design of a QAS using no other cryptographic tools but what another QAS provides and in a way that preserves correctness and security.

Definition 5.5.12 (Reductions of Query Authentication Schemes)**.** *Let S and S' be data sets, $Q_S : \mathcal{Q} \to \mathcal{A}_S$, $Q'_S : \mathcal{Q}' \to \mathcal{A}'_S$ be query operations on S and S' respectively, and $QAS(Q_S, S) = (\mathsf{KeyG}, \mathsf{Auth}, \mathsf{Res}, \mathsf{Ver})$, $QAS(Q'_S, S') = (\mathsf{KeyG}', \mathsf{Auth}', \mathsf{Res}', \mathsf{Ver}')$ be query authentication schemes for Q_S on S and Q'_S on S' respectively. We say that $QAS(Q_S, S)$ is* authenticated reduced *to $QAS(Q'_S, S')$, if key generation algorithms KeyG and KeyG' are identical, $QAS(Q_S, S)$ uses the public and secret keys generated by* KeyG never explicitly*, but* only implicitly *through black-box invocations of algorithms Auth', Res' and Ver', and $QAS(Q_S, S)$ is correct and secure whenever $QAS(Q'_S, S')$ is correct and secure.*

A general query authentication scheme. Let $S = (\mathcal{E}, \mathcal{R})$ be a structured data set and $Q_S : \mathcal{Q} \to \mathcal{A}_S$ be any query operation. Let $D(Q_S) = (\mathcal{E}_Q, \mathcal{R}_Q, \mathsf{Answer})$ be a query data structure for Q_S. By Lemma 5.5.1, we know that there exists a certification data structure $C(Q_S) = ((\mathcal{E}_C, \mathcal{R}_C), \mathsf{Certify}, \mathsf{Verify})$ for S with respect to Q_S. Let $Q_\in : \mathcal{Q}(\mathcal{R}_C) \to \{\mathsf{yes}, \mathsf{no}\}$ be the set-membership query operation, where the query space $\mathcal{Q}(\mathcal{R}_C)$ is the indexed tuples that exist in $\mathcal{R}_C$. Assuming the existence of a secure and correct $QAS(Q_\in, \mathcal{R}_C) = (\mathsf{KeyG}', \mathsf{Auth}', \mathsf{Res}', \mathsf{Ver}')$, we next construct $QAS(Q_S, S) = (\mathsf{KeyG}, \mathsf{Auth}, \mathsf{Res}, \mathsf{Ver})$. We note that, in essence, our construction is parameterized by $(\mathsf{KeyG}', \mathsf{Auth}', \mathsf{Res}', \mathsf{Ver}')$, an QAS for set membership queries. We define our query authentication scheme $(\mathsf{KeyG}, \mathsf{Auth}, \mathsf{Res}, \mathsf{Ver})$ for Q_S and S as follows.

(A) Key-generation algorithm. By definition it is the same as KeyG', thus, $SK = SK'$ and $PK = PK'$.

(B) Authenticator. The authenticator algorithm Auth using S and Constr_C computes the structured data set $S_C = (\mathcal{E}_C, \mathcal{R}_C)$ of the corresponding certification structure $C(Q_S) = ((\mathcal{E}_C, \mathcal{R}_C), \mathsf{Certify}, \mathsf{Verify})$. Then Auth runs algorithm Auth' on input SK', PK', $Q_\in$ and $\mathcal{R}_C$. That is, algorithm Auth computes the pair $(\alpha', \mathsf{V}') \leftarrow \mathsf{Auth}'(SK', PK', Q_\in, \mathcal{R}_C)$, and then Auth outputs (α', V').

(C) Responder. The responder algorithm Res first computes the structured data sets $S_Q = (\mathcal{E}_Q, \mathcal{R}_Q)$ and $S_C = (\mathcal{E}_C, \mathcal{R}_C)$ using S and algorithms Constr_Q and Constr_C. Then, on input q, S_Q and S_C it simply runs algorithm $\mathsf{Certify}$ to produce its pair (a, τ). Then Res constructs the certification support $\mathcal{R}_C(\tau)$ of answer a by accessing set $\mathcal{R}_C$ with the use of indices in τ. For every tuple $< t >$ in $\mathcal{R}_C$, algorithm Res runs the responder algorithm Res' on inputs $< t >$, $\mathcal{R}_C$ and V' to get $(a'(t), p'(t)) \leftarrow \mathsf{Res}'(< t >, \mathcal{R}_C, \mathsf{V}')$ and, if $(t_1, \ldots, t_{|\tau|})$ is the sequence of tuples accessed in total, Res creates sequence $p' = (p'(t_1), \ldots, p'(t_{|\tau|}))$, sets $p = (\tau, \mathcal{R}_C(\tau), p')$ and finally outputs (a, p).

(D) Verifier. The verifier algorithm Ver first checks if the proof p and answer a are both well-formed and, if not, it rejects. Otherwise, by appropriately processing the proof p, algorithm Ver runs algorithm Verify on inputs q, $\mathcal{R}_C(\tau)$, a and τ. Whenever algorithm Verify needs to access and process a tuple $< t_i >$, where $< t_i >$ is the i-th tuple accessed by Verify, algorithm Ver runs algorithm Ver' on inputs PK', $< t_i >$, $(\mathsf{yes}, p'(t_i))$ and α' and if $0 \leftarrow \mathsf{Ver}'(PK', < t_i >, (\mathsf{yes}, p'(t_i)), \alpha')$, algorithm Ver rejects. Otherwise, Ver continues with the computation. Finally, Ver accepts if and only if Verify accepts, i.e., if and only if $1 \leftarrow \mathsf{Verify}(q, \mathcal{R}_C(\tau), a, \tau)$.

We have thus constructed $QAS(Q_S, S)$, where Q_S is a general query operation of set S, parameterized by $QAS(Q_\in, \mathcal{R}_C)$, where $Q_\in$ is the set membership query operation and $\mathcal{R}_C$ is the certification image of S with respect to the certification data structure in use. We can show the following results.

Theorem 5.5.1. *Let $QAS(Q_\in, \mathcal{R}_C)$ be any query authentication structure for set membership queries and $QAS(Q_S, S)$ our query authentication scheme constructed above. For any query operation Q_S and any data set S, $QAS(Q_S, S)$ is correct and secure if $QAS(Q_\in, \mathcal{R}_C)$ is correct and secure.*

Proof. We start by first discussing the correctness property. Suppose that query authentication scheme $(\mathsf{KeyG}', \mathsf{Auth}', \mathsf{Res}', \mathsf{Ver}')$ is correct. We want to show that $(\mathsf{KeyG}, \mathsf{Auth}, \mathsf{Res}, \mathsf{Ver})$ is correct. This easily follows from checking that the verifier Ver does not reject when given

an answer-proof pair from the responder Res, for any query issued in $\mathcal{Q}$. Indeed, from the completeness property of the certification data structure the answer testing algorithm Verify does not reject, and additionally the correctness of $(\mathsf{KeyG}', \mathsf{Auth}', \mathsf{Res}', \mathsf{Ver}')$ guarantee that Ver does not reject because of a rejection by Ver'.

For the security we argue as follows. Suppose that $(\mathsf{KeyG}', \mathsf{Auth}', \mathsf{Res}', \mathsf{Ver}')$ is secure. Assume that $(\mathsf{KeyG}, \mathsf{Auth}, \mathsf{Res}, \mathsf{Ver})$ is not secure, then with overwhelming probability responder Res responds to a query $q \in \mathcal{Q}$ incorrectly but still the verifier Ver fails to reject its input. Based on the soundness property of the certification data structure in use, we must admit that it is not algorithm Certify that cheats the verifier, that is, it is not the indices in sequence τ that cause the problem, but rather the fact that algorithm Verify runs on incorrect data. Then there must be at least one tuple in $\mathcal{R}_C$ that although it was verified to be a member of $\mathcal{R}_C$ it is not authentic, meaning that its index is correct but one or more of the data elements in the tuple have been (maliciously) altered. We thus conclude that for at least one query the verification algorithm Ver' of query authentication scheme $(\mathsf{KeyG}', \mathsf{Auth}', \mathsf{Res}', \mathsf{Ver}')$ failed to reject on an invalid query-answer pair. This is a contradiction, since this scheme is assumed to be secure. □

Theorem 5.5.2. *For any query operation Q_S on any structured data set S, there exists a secure and correct query authentication structure $QAS(Q_S, S)$ based on a certification data structure $C(Q_S)$. Moreover, $QAS(Q_S, S)$ is authenticated reduced to any secure and correct query authentication structure $QAS(Q_\in, \mathcal{R}_C)$ for the set membership query operation $Q_\in$ on some certification image $\mathcal{R}_C$ of $C(Q_S)$.*

Proof. The result follows by our construction $QAS(Q_S, S)$ and the fact that there exist secure query authentication schemes $QAS(Q_\in, \cdot)$ for membership queries on any data set: in particular, digital signatures, Merkle's hash tree and one-way accumulators provide a correct and secure implementation of $QAS(Q_\in, \cdot)$. □

We now show what are the implications of Theorems 5.5.1 and Theorem 5.5.2 in terms of time and space complexity. First, let us define the cost measures that are of interest in a query authentication scheme $QAS(Q_S, S) = (\mathsf{KeyG}, \mathsf{Auth}, \mathsf{Res}, \mathsf{Ver})$ for a set of size n. Let $T_a(n)$, $T_r(n)$, $T_v(n)$ denote the time complexity of algorithms Auth, Res and Ver respectively, $S_a(n)$, $S_r(n)$ denote the space complexity of Auth, Res. Also for $QAS(Q_\in, \mathcal{R}_C) = (\mathsf{KeyG}', \mathsf{Auth}', \mathsf{Res}', \mathsf{Ver}')$, let $T'_a(n)$, $T'_r(n)$, $T'_v(n)$ denote the time complexity of algorithms Auth', Res' and Ver' respectively, $S'_a(n)$, $S'_r(n)$ denote the space complexity of Auth', Res'. Recall from Section 5.5.3 that for certification data structure

$C(Q_S) = ((\mathcal{E}_C, \mathcal{R}_C), \mathsf{Certify}, \mathsf{Verify})$, $T_A(n)$, $T_C(n)$, $T_V(n)$, $S_Q(n)$ and $S_C(n)$ denote various time and space complexity measures. Also let $p(n)$ denote the proof size in $QAS(Q_S, S)$ and $p'(n)$ the proof size in $QAS(Q_\in, \mathcal{R}_C)$. We have the following.

Lemma 5.5.2. *Let S be a structured data set and $C(Q_S) = ((\mathcal{E}_C, \mathcal{R}_C), \mathsf{Certify}, \mathsf{Verify})$ be a certification data structure for S. Let n be the size of S. Additionally, let $m(n) = |\mathcal{R}_C|$ denote the size of the certification image and $s(n) = |\mathcal{R}_C(\tau)|$ the size of the certification support of an answer. For any query operation Q_S, the query authentication scheme $QAS(Q_S, S) = (\mathsf{KeyG}, \mathsf{Auth}, \mathsf{Res}, \mathsf{Ver})$ that is based on query authentication scheme $QAS(Q_\in, \mathcal{R}_C) = (\mathsf{KeyG}', \mathsf{Auth}', \mathsf{Res}', \mathsf{Ver}')$ and uses certification data structure $C(Q_S)$ has the following performance.*

1. *With respect to time complexity, it is $T_a(n) = O(T'_a(n))$, $T_r(n) = O(s(n)T'_r(n) + T_C(n))$, $T_v(n) = O(s(n)T'_v(n) + T_V(n))$;*
2. *With respect to space complexity, it is $S_a(n) = O(S'_a(n)+n+m(n))$, $S_r(n) = O(S'_r(n)+ S_Q(n) + m(n))$;*
3. *With respect to the proof size, it is $p(n) = O(s(n)p'(n))$.*

Proof. It follows directly by the construction of $QAS(Q_S, S)$ and the use of $QAS(Q_\in, \mathcal{R}_C)$ and $C(Q_S) = (\mathcal{E}_C, \mathcal{R}_C, \mathsf{Certify}, \mathsf{Verify})$. □

We can now use the above lemma to have general complexity results in terms of our parameterized query authentication scheme $QAS(Q_S, S)$. By appropriately choosing known (secure and correct) constructions for authenticating set membership queries we can achieve trade-offs on the efficiency of general query authentication schemes. Here, we are interested only in asymptotic analysis, omitting improvements of constant factors. So, we only study the related costs with respect to the set size n and not the exact implementation of the cryptographic primitives.

Theorem 5.5.3. *Let S be a structured data set and $C(Q_S) = (\mathcal{E}_C, \mathcal{R}_C, \mathsf{Certify}, \mathsf{Verify})$ be a certification data structure for S. Let n be the size of S. Additionally, let $m(n) = |\mathcal{R}_C|$ denote the size of the certification image and $s(n) = |\mathcal{R}_C(\tau)|$ the size of the certification support of an answer. For any query operation Q_S, the query authentication scheme $QAS(Q_S, S) = (\mathsf{KeyG}, \mathsf{Auth}, \mathsf{Res}, \mathsf{Ver})$ that is based on query authentication scheme $QAS(Q_\in, \mathcal{R}_C) = (\mathsf{KeyG}', \mathsf{Auth}', \mathsf{Res}', \mathsf{Ver}')$ and uses certification data structure $C(Q_S)$ has the following performance.*

Static Case *Using only signatures, we have the following performance:*

With respect to time complexity, we have that $T_a(n)$ is $O(m(n))$, $T_r(n)$ is $O(s(n)+T_C(n))$, $T_v(n)$ is $O(s(n)+T_V(n))$; with respect to space complexity, $S_a(n)$ is $O(n+m(n))$, $S_r(n)$ is $O(S_Q(n)+m(n))$; with respect to the proof size, $p(n)$ is $O(s(n))$.

Dynamic Case *Using signature amortization, we have the following performance:*

Hash Tree *With respect to time complexity, we have that $T_a(n)$ is $O(m(n))$, $T_r(n)$ is $O(s(n)\log n + T_C(n))$, $T_v(n)$ is $O(s(n)\log n + T_V(n))$; with respect to space complexity, $S_a(n)$ is $O(n+m(n))$, $S_r(n)$ is $O(n+S_Q(n)+m(n))$; with respect to the proof size, $p(n)$ is $O(s(n)\log n)$; when k tuples are updated, these can be handled in $O(k\log n)$ time.*

Dynamic Accumulator *With respect to time complexity, we have that $T_a(n)$ is $O(m(n))$, $T_r(n)$ is $O(s(n)\sqrt{n}+T_C(n))$, $T_v(n)$ is $O(s(n)+T_V(n))$; with respect to space complexity, $S_a(n)$ is $O(n+m(n))$, $S_r(n)$ is $O(n+S_Q(n)+m(n))$; with respect to the proof size, $p(n)$ is $s(n)$; when k tuples are updated, these can be handled in $O(k\sqrt{n})$ time.*

Proof. For the static case, simply the use of signatures provides a satisfactory time-space trade-off. That is, every indexed tuple in the certification image $\mathcal{R}_C$ is signed. The query authentication scheme $QAS(Q_\in, \mathcal{R}_C)$ in this case is very simple: Auth signs all tuples in $\mathcal{R}_C$ and sets α to be all these signatures with $\mathsf{V} = \perp$; Res, along with the (positive) answer to an $\in$ query, returns the corresponding tuples in $\mathcal{R}_C(\tau)$ and the corresponding signature; and Ver simply verifies a number of signatures.

For the dynamic case, the extensive use of signatures is not an efficient solution, since because of the updates on the set S, after every update all signatures have to be updated. Alternatively, signature amortization can be used, where only one digest of set $\mathcal{R}_C$ is signed (incurring $O(1)$ update (signing) cost). Two alternative options for computing the digest of set $\mathcal{R}_C$ are: (i) the use of a hash tree and (ii) the use of an accumulator. The construction of $QAS(Q_\in, \mathcal{R}_C)$ is straightforward and we omit here the details. Hash trees have linear storage needs, logarithmic access, update and verification times and logarithmic proof size. Dynamic accumulators, on the other hand, have linear storage needs, constant time verification and constant proof, at an increased cost to support updates and processes (of witnesses). Note that the use of the trapdoor information can only be used by algorithm Auth and not by algorithm Res for it would destroy the security of the scheme. In [57] some

interesting trade-offs between the update and process times costs are discussed (e.g., one can achieve a $\sqrt{n}$ trade-off). □

By Theorem 5.5.2, all query operations can be authenticated in the three-party authentication model. Theorem 5.5.3 gives a detailed complexity analysis of the authenticated data structures derived by the corresponding query authentication schemes. Still, the above description depends on the complexity of the certification data structure used. Our results hold for the RAM model of computation, which strictly includes the pointer machine model. By Lemma 5.5.1, all query problems that have a query data structure have a certification data structure and thus our results generalize previous known results. Indeed, we can show the following meta-theorem regarding the design of authenticated data structures.

Theorem 5.5.4. *Let S be a structured data set and $\mathcal{Q}_s$ be a query operation on S. If there exists a time (space) super-efficient certification data structure for Q_S, then there exists a time (space) super-efficient authenticated data structure for Q_S.*

Finally, we note that Theorems 5.2.1 and 5.5.2 can be combined to give a general possibility result for the design of distributed authenticated data structures. In particular, we have seen that the authentication of any type of query is reduced to the authentication of set-membership queries. Also, we have seen a distributed implementation of an authentication tree that allows the authentication of set-membership queries. Accordingly, we get that there exist distributed authenticated data structures for any type of queries.

5.6 Conclusions

We consider the problem of data authentication in p2p storage networks. We introduce a new model for authenticating data in totally decentralized computing environments that extends the model of authenticated data structures and better captures the security needs of existing distributed systems. We design the first efficient implementations of a distributed Merkle tree (DMT). We identify inefficiencies and security problems in the authentication techniques that are currently used by existing distributed storage systems and we show how our DMT can be used in combination with any distributed hash table (DHT) to implement an efficient authenticated DHT (ADHT). Using an ADHT, we obtain the first efficient distributed authenticated dictionary. Finally, we present a new framework for the problem of data authentication in a very general data query model, prove general results on the design of efficient authentication data structures, characterize sufficient conditions for super-efficient data verification and prove that the authentication of general queries is

reduced to the authentication of set-membership queries. This last result, combined with our construction of a distributed dictionary, provides a new general result for distributed data authentication, namely, that for any data query problem there exists a distributed authentication data structure for authenticating the queries over a peer-to-peer storage network.

Open problems include the explicit design of authenticated distributed data structures for general queries (beyond dictionaries) and the study of additional security issues in this new model (e.g., Byzantine behavior).

Parts of the material presented in this chapter were developed in collaboration with Roberto Tamassia and Michael T. Goodrich.

Chapter 6

Conclusions

6.1 Summary of Results

In this dissertation, we present an extensive study on the problem of data authentication that seeks for techniques that securely and efficiently verify information disseminated in distributed untrusted or adversarial environments. In particular, the data authentication problem studies the following question: when the distributor of the data is not the source of the data, and thus is not trusted by the end user, how can data received be proven authentic? This question captures the security needs of many computing applications that exchange and use sensitive information in hostile distributed environments and its importance increases given the current trend in modern system design towards decentralized architectures with minimal trust assumptions. Moreover, solutions should not only be provably secure, but efficient and easily implementable.

With this dissertation, we contribute results on the data authentication problem in various directions.

For the case where data is structured and is being retrieved through queries, we design new efficient authenticated data structures for two broad classes of queries over graphs and geometric objects. Our main construction is the path hash accumulator, an extension of the Merkle authentication tree for authenticating decomposable queries over sequences of data elements. Using this construction, we provide efficient authenticated data structures for queries over paths in graphs and also for the iterative search problem, i.e., set-membership queries executed over many distinct dictionaries that are organized as a graph. Using these intermediate structures, new efficient authenticated data structures for path and connectivity queries in graphs and search problems in collections of two-dimensional geometric objects

are finally obtained. Our authentication structures are efficient, asymptotically introducing no additional cost to the query data structure.

We also study the complexity of the problem of designing efficient authenticated data structures, focusing on the communication and computational overhead that is inherently incurred in data authentication. We prove the first lower bound for authenticated dictionaries, that is, for the case where we want to verify answers to the set-membership problem. In particular, we show that, when only digital signatures and cryptographic hashing are used, the authentication of membership queries in a set of size n requires $\Omega(\log n)$ communication and computational costs in the worst case. This answers an open question posed by Naor and Nissim in [103]. The result is of interest since set-membership is the fundamental type of queries that we wish to authenticate. In view of this lower bounds, we design a new authenticated dictionary that is very close to the theoretically optimal construction.

For the case where data is unstructured and is transmitted over an adversarial network as a stream (in packets), we present a powerful authentication construction that authenticates received packets at asymptotically no extra communication overhead. We use a novel combination of error-correcting codes with cryptographic primitives (signatures and collision-resistant hashing), the first of this kind. Our construction essentially introduces authenticated error-correcting codes and finds applications to the problem of multicast authentication. In a network that can adversarially inject, remove or alter packets, we show that multicast streams can be efficiently authenticated so that any receiver can correctly distinguish valid packets from invalid ones.

We also study distributed data authentication, that is, the data authentication problem in the case where data is shared among remote network nodes that participate in a distributed overlay storage network (e.g., a peer-to-peer network). We effectively extend the client-server model of authenticated data structures to a totally decentralized data querying setting, where data is retrieved through any distributed object location system (e.g., a distributed hash table). We present the first distributed implementation of a Merkle tree, the fundamental authentication technique for verifying set membership. We also present an implementation that by construction achieves load-balance of the computational overhead for accessing the network that is introduced by the authentication structure. Both of our constructions are efficient and space optimal: at a logarithmic (per network node) storage cost, membership queries in a set of size m are authenticated over a network of size n with $O(\log n \log m)$ communication and computational overhead. Overall, we obtain an efficient authenticated version of a distributed hash table.

We finally develop a new framework for the design of authenticated data structures that

focuses on the following principle: the answer verification task should be separated from the answer generation procedure. In a very general computational model (RAM model), we show that every type of query admits such formulation. Overall we get two main results. First, we derive sufficient conditions for the design of authenticated data structures that have super-efficient answer verification—that is, the answer is verified in asymptotically less time than the time spent to produce it. Second, we prove that the authentication of any type of query is reduced to the authentication of set-membership queries. This last result, in combination with our distributed authenticated dictionary, allows us to design distributed authenticated data structures for any querying problem.

6.2 Future Directions

Data authentication is an exciting new dimension of data management that is well worth further exploration. Future research directions related to the subject of this dissertation include the following.

Cryptographic primitives. For efficiency, expressiveness and added security, it is very interesting to extend known and develop new cryptographic primitives that better satisfy the requirements of the problem of data authentication. Known cryptographic techniques, like signatures and cryptographic hashing, can only provide some elementary functionality, namely to authenticate message integrity and the set inclusion property. This dissertation studies hash-based data authentication and the case where signature amortization is used. That is, only one digest (or a collection of digests) of the data are signed and then all queries or other data elements are authenticated with respect to that digest(s) using collision-resistant hashing. It may well be the case, though, that this authentication technique has inherent limitations in terms of efficiency. Indeed, the results of Chapter 3 indicate that the "hash and sign" paradigm have some limitations for the set-membership problem.

This suggests that maybe new cryptographic constructions can be more appropriate and more efficient for the problem in study. For instance, consider (new) cryptographic primitives that authenticate complicated relations over data elements; in this case, we could get more expressive authenticated data structures and more compact constructions. Also, consider dynamic accumulator [18, 57], which have been shown to allow very efficient answer verification for positive set-membership queries. It is interesting to explore their other properties or uses in data authentication and to advance their functionality to more general data querying problems.

Data privacy. Data privacy, although orthogonal to data authentication, is certainly an additive desirable feature. For authenticated data structures, for instance, we can ask for the property that the proof of an answer to a query does not reveal any information about the data set not implied by the answer. In many applications this extra property may actually be very important, since privacy is often a crucial requirement. The works in [98, 108] study privacy issues in a related model where the data source commits to the data set and then a user that is querying the data set validates the answers and checks their consistency subject to the committed value. It is worth exploring practical and efficient privacy-preserving data-authentication techniques, e.g., non-interactive techniques for general queries.

Data Consistency. Another property of the works in [98, 108] is data consistency, namely the inability of the source to provide two contradictory, but verifiable, answers to the same query. It is very interesting to study this property in the context of authenticated data structures. We can consider for instance the extension of the model of authenticated data structures, where the data source does not itself constitute a completely trusted by the user entity. In this scenario, we wish at any time the user (possibly in collaboration with the data distributor) to be able to check that data received is consistent with the data set that is queried. This model would have many applications in information assurance in distributed systems.

Authentication of computation. Finally, we could consider another generalization of the problem that this dissertation studies. Data authentication verifies data and essentially the result of a query to a data structure. A natural extension is to consider authentication of the result of an entire computation. That is, consider many users submitting their individual inputs to an untrusted party that performs a computation over these inputs and, then, they collectively wish to authenticate that the returned to them result is correct (valid). Can we use techniques similar to data authentication and concepts related to authenticated data structures so that the users can authenticate a computation off-line? In a way, this problem is antisymmetric to multi-party computation, where parties jointly compute a function of their inputs, as this was computed by a trusted third party.

Bibliography

[1] William Aiello, Sachin Lodha, and Rafail Ostrovsky. Fast digital identity revocation. In *Advances in Cryptology – CRYPTO ' 98*, LNCS, pages 137–152. Springer-Verlag, 1998.

[2] Aris Anagnostopoulos, Michael T. Goodrich, and Roberto Tamassia. Persistent authenticated dictionaries and their applications. In *Proceedings of Information Security Conference (ISC 2001)*, volume 2200 of *LNCS*, pages 379–393. Springer-Verlag, 2001.

[3] C. Aragon and R. Seidel. Randomized search trees. In *Proceedings of 30th Annual IEEE Symposium on Foundations of Computer Science*, pages 540–545, 1989.

[4] Niko Barić and Birgit Pfitzmann. Collision-free accumulators and fail-stop signature schemes without trees. In Walter Fumy, editor, *Advances in Cryptology — EUROCRYPT '97*, volume 1233 of *Lecture Notes in Computer Science*, pages 480–494. Springer Verlag, 1997.

[5] Josh Benaloh and Michael de Mare. One-way accumulators: A decentralized alternative to digital signatures. In Tor Helleseth, editor, *Advances in Cryptology — EUROCRYPT '93*, volume 765 of *Lecture Notes in Computer Science*, pages 274–285. Springer-Verlag, 1994.

[6] S. W. Bent, D. D. Sleator, and R. E. Tarjan. Biased search trees. *SIAM Journal of Computing*, 14:545–568, 1985.

[7] Elisa Bertino, Barbara Carminati, Elena Ferrari, Bhavani M. Thuraisingham, and Amar Gupta. Selective and authentic third-party distribution of XML documents. *IEEE Transactions on Knowledge and Data Engineering*, 16(6):1263–1278, 2004.

[8] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. *Algorithmica*, 12(2/3):225–244, 1994.

[9] M. Blum and H. Wasserman. Program result-checking: A theory of testing meets a test of theory. In *Proceedings of 35th Annual Symposium on Foundations of Computer Science*, pages 382–393, 1994.

[10] Manuel Blum and Sampath Kannan. Designing programs that check their work. *Journal of the ACM*, 42(1):269–291, January 1995.

[11] Dan Boneh, Glenn Durfee, and Matt Franklin. Lower bounds for multicast message authentication. In Birgit Pfitzmann, editor, *Advances in Cryptology — EUROCRYPT 2001*, volume 2045 of *Lecture Notes in Computer Science*, pages 437–452. Springer Verlag, 2001.

[12] Dan Boneh and Matthew Franklin. Identity-based encryption from the Weil pairing. *SIAM Journal on Computing*, 32(3):586–615, 2003.

[13] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. In *Advances in Cryptology, ASIACRYPT 2001*, volume 2248 of *Lecture Notes in Computer Science*, pages 514–532. Springer-Verlag, 2001.

[14] J. D. Bright and G. Sullivan. Checking mergeable priority queues. In *Digest of the 24th Symposium on Fault-Tolerant Computing*, pages 144–153. IEEE Computer Society Press, 1994.

[15] J. D. Bright and G. Sullivan. On-line error monitoring for several data structures. In *Digest of the 25th Symposium on Fault-Tolerant Computing*, pages 392–401. IEEE Computer Society Press, 1995.

[16] J. D. Bright, G. Sullivan, and G. M. Masson. Checking the integrity of trees. In *Digest of the 25th Symposium on Fault-Tolerant Computing*, pages 402–411. IEEE Computer Society Press, 1995.

[17] Ahto Buldas, Peeter Laud, and Helger Lipmaa. Accountable certificate management using undeniable attestations. In *Proceedings of ACM Conference on Computer and Communications Security*, pages 9–18. ACM Press, 2000.

[18] Jan Camenisch and Anna Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In Moti Yung, editor, *Advances in Cryptology — CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 61–76. Springer Verlag, 2002.

[19] R. Canetti, T. Malkin, and K. Nissim. Efficient communication - storage tradeoffs for multicast encryption. In *Advances in cryptology (EUROCRYPT'99), LNCS 1592*, pages 459–474, 1999.

[20] Ran Canetti, Juan Garay, Gene Itkis, Daniele Micciancio, Moni Naor, and Benny Pinkas. Multicast security: A taxonomy and some efficient constructions. In *Proceedings of IEEE INFOCOM*, pages 708–716, 1999.

[21] Sean Cannella, Michael Shin, Christian Straub, Roberto Tamassia, and Daniel J. Polivy. Secure visualization of authentication information: A case study. In *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 35–37, 2004.

[22] Adlar C-F Chan. A graph-theoretical analysis of multicast authentication. In *Proceedings of 23rd International Conference on Distributed Computing Systems – ICDCS*, pages 155–162, 2003.

[23] Bernard Chazelle and Leonidas J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1(3):133–162, 1986.

[24] Bernard Chazelle and Leonidas J. Guibas. Fractional cascading: II. Applications. *Algorithmica*, 1:163–191, 1986.

[25] Robert F. Cohen and Roberto Tamassia. Combine and conquer. *Algorithmica*, 18:342–362, 1997.

[26] Adina Crainiceanu, Prakash Linga, Johannes Gehrke, and Jayavel Shanmugasundaram. Querying peer-to-peer networks using P-trees. In *Proceedings of Seventh International Workshop on the Web and Databases WebDB 2004*, pages 25–30, 2004.

[27] Ronald Cramer and Victor Shoup. Signature schemes based on the strong RSA assumption. *ACM Transactions on Information and System Security*, 3(3):161–185, 2000.

[28] T. Cucinotta, Gabriele Cecchetti, and Gianluca Ferraro. Adopting redundancy techniques for multicast stream authentication. In *Proceedings of 9th IEEE International Workshop on Future Trends of Distributed Computing Systems (FTDCS 2003)*, pages 189–201, 2003.

[29] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[30] Ivan Bjerre Damgård. A design principle for hash functions. In Gilles Brassard, editor, *Advances in Cryptology — CRYPTO '89*, volume 435 of *Lecture Notes in Computer Science*, pages 416–427, 1990.

[31] Yvo Desmedt, Yair Frankel, and Moti Yung. Multi-receiver/multi-sender network security: Efficient authenticated multicast/feedback. In *Proceedings of IEEE Conference on Computer Communications — INFOCOM '92*, pages 2045–2054. IEEE-Press, 1992.

[32] Premkumar Devanbu, Michael Gertz, April Kwong, Chip Martel, Glen Nuckolls, and Stuart Stubblebine. Flexible authentication of XML documents. In *Proceedings of ACM Conference on Computer and Communications Security*, pages 136–145, 2001.

[33] Premkumar Devanbu, Michael Gertz, Chip Martel, and Stuart G. Stubblebine. Authentic third-party data publication. In *Proceedings of Fourteenth IFIP 11.3 Conference on Database Security*, 2000.

[34] Premkumar Devanbu, Michael Gertz, Chip Martel, and Stuart G. Stubblebine. Authentic data publication over the Internet. *Journal of Computer Security*, 11(3):291–314, 2003.

[35] Olivier Devillers, Giuseppe Liotta, Franco P. Preparata, and Roberto Tamassia. Checking the convexity of polytopes and the planarity of subdivisions. *Computational Geometry: Theory and Applications*, 11:187–208, 1998.

[36] G. Di Battista and G. Liotta. Upward planarity checking: "Faces are more than polygons". In S. H. Whitesides, editor, *Proceedings of Graph Drawing*, volume 1547 of *LNCS*, pages 72–86. Springer-Verlag, 1998.

[37] G. Di Battista and R. Tamassia. On-line maintenance of triconnected components with SPQR-trees. *Algorithmica*, 15:302–318, 1996.

[38] Peter Druschel and Antony Rowstron. Past: A large-scale, persistent peer-to-peer storage utility. In *HOTOS '01: Proceedings of Eighth Workshop on Hot Topics in Operating Systems*, page 75, Washington, DC, USA, 2001. IEEE Computer Society.

[39] D. Eppstein, G. F. Italiano, R. Tamassia, R. E. Tarjan, J. Westbrook, and M. Yung. Maintenance of a minimum spanning forest in a dynamic plane graph. *Journal of Algorithms*, 13(1):33–54, 1992.

[40] Amos Fiat and Jared Saia. Censorship resistant peer-to-peer content addressable networks. In *Proceedings of Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 94–103, 2002.

[41] Amos Fiat, Jared Saia, and Maxwell Young. Making Chord robust to byzantine attacks. In *Proceedings of European Symposium on Agorithms*, pages 803–814, 2005.

[42] U. Finkler and K. Mehlhorn. Checking priority queues. In *Proceedings of 10th ACM-SIAM Symposium on Discrete Algorithms*, pages S901–S902, 1999.

[43] Marc Fischlin. The Cramer-Shoup strong-RSA signature scheme revisited. In Yvo Desmedt, editor, *Proceedings of Public Key Cryptography - PKC 2003*, volume 2567 of *Lecture Notes in Computer Science*, pages 116–129. Springer-Verlag, 2003.

[44] Michael J. Freedman and Radek Vingralek. Efficient peer-to-peer lookup based on a distributed TRIE. In *Proceedings of 1st International Workshop on Peer-to-Peer Systems (IPTPS02)*, Cambridge, MA, March 2002.

[45] Kevin Fu, M. Frans Kaashoek, and David Mazieres. Fast and secure distributed read-only file system. *Computer Systems*, 20(1):1–24, 2002.

[46] Irene Gassko, Peter S. Gemmell, and Philip MacKenzie. Efficient and fresh certification. In *Proceedings of International Workshop on Practice and Theory in Public Key Cryptography (PKC '2000)*, volume 1751 of *LNCS*, pages 342–353. Springer-Verlag, 2000.

[47] Rosario Gennaro and Pankaj Rohatgi. How to sign digital streams. In Burt Kaliski, editor, *Advances in Cryptology — CRYPTO '97*, volume 1294 of *Lecture Notes in Computer Science*, pages 180–197. Springer Verlag, 1997.

[48] Oded Goldreich. *Foundations of Cryptography*, volume II: Basic Applications. Cambridge University Press, 2004.

[49] Oded Goldreich, Ronitt Rubinfeld, and Madhu Sudan. Learning polynomials with queries: The highly noisy case. *SIAM Journal on Discrete Mathematics*, 13(4):535–570, November 2000.

[50] Shafi Goldwasser, Silvio Micali, and Ronald Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, April 1988.

[51] Philippe Golle and Nagendra Modadugu. Authenticating streamed data in the presence of random packet loss. In *Proceedings of Network and Distributed System Security Symposium —NDSS '01*, pages 13–22, 2001.

[52] M. T. Goodrich and R. Tamassia. Efficient authenticated dictionaries with skip lists and commutative hashing. Technical report, Johns Hopkins Information Security Institute, 2000. Available from `http://www.cs.brown.edu/cgc/stms/papers/hashskip.pdf`.

[53] Michael T. Goodrich, Mikhail J. Atallah, and Roberto Tamassia. Indexing information for data forensics. In John Ioannidis, Angelos Keromytis, and Moti Yung, editors, *Proceedings of International Conference on Applied Cryptography and Network Security (ACNS)*, volume 3531 of *LNCS*, pages 206–221. Springer-Verlag, 2005.

[54] Michael T. Goodrich, James Lentini, Michael Shin, Roberto Tamassia, and Robert Cohen. Design and implementation of a distributed authenticated dictionary and its applications. Technical report, Center for Geometric Computing, Brown University, 2002. Available from `http://www.cs.brown.edu/cgc/stms/papers/stms.pdf`.

[55] Michael T. Goodrich, Michael Shin, Roberto Tamassia, and William H. Winsborough. Authenticated dictionaries for fresh attribute credentials. In *Proceedings of Trust Management Conference*, volume 2692 of *LNCS*, pages 332–347. Springer, 2003.

[56] Michael T. Goodrich, Jonathan Z. Sun, and Roberto Tamassia. Efficient tree-based revocation in groups of low-state devices. In M. Franklin, editor, *Advances in Cryptology – CRYPTO*, volume 3152 of *Lecture Notes in Computer Science*, pages 511–527. Springer-Verlag, 2004.

[57] Michael T. Goodrich, Roberto Tamassia, and Jasminka Hasic. An efficient dynamic and distributed cryptographic accumulator. In *Proceedings of Information Security Conference (ISC)*, volume 2433 of *LNCS*, pages 372–388. Springer-Verlag, 2002.

[58] Michael T. Goodrich, Roberto Tamassia, and Andrew Schwerin. Implementation of an authenticated dictionary with skip lists and commutative hashing. In *Proceedings of*

2001 DARPA Information Survivability Conference and Exposition, volume 2, pages 68–82, 2001.

[59] Michael T. Goodrich, Roberto Tamassia, Nikos Triandopoulos, and Robert Cohen. Authenticated data structures for graph and geometric searching. In *Proceedings of RSA Conference—Cryptographers' Track*, volume 2612 of *LNCS*, pages 295–313. Springer, 2003.

[60] Justin Goshi and Richard E. Ladner. Algorithms for dynamic multicast key distribution trees. In *Proceedings of twenty-second Annual Symposium on Principles of Distributed Computing (PODC 2003)*, pages 243–251. ACM, 2003.

[61] Carl Gunter, Sanjeev Khanna, Kaijun Tan, and Santosh Venkatesh. DoS protection for reliably authenticated broadcast. In *Proceedings of Eleventh Annual Network and Distributed Systems Security Symposium — NDSS '04*, 2004.

[62] Venkatesan Guruswami. *List Decoding of Error-correcting Codes*. PhD thesis, Massachusetts Institute of Technology, Boston, MA, 2001.

[63] Venkatesan Guruswami and Madhu Sudan. Improved decoding of Reed-Solomon and algebraic-geometric codes. In *IEEE Transactions on Information Theory*, pages 45:1757–1767, 1999.

[64] Eric Hall and Charanjit S. Julta. Parallelizable authentication trees. In Cryptology ePrint Archive, December 2002.

[65] R. Huebsch, B. Chun, J. Hellerstein, B. Loo, P. Maniatis, T. Roscoe, S. Shenker, I. Stoica, and A. Yumerefendi. The architecture of PIER: an internet-scale query processor. In *Proceedings of 2nd Conference on Innovative Data Systems Research (CIDR)*, pages 28–43, 2005.

[66] H. V. Jagadish, Beng Chin Ooi, and Quang Hieu Vu. Baton: a balanced tree structure for peer-to-peer networks. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 661–672. VLDB Endowment, 2005.

[67] Frans Kaashoek and David R. Karger. Koorde: A simple degree-optimal distributed hash table. In *Proceedings of 2nd International Workshop on Peer-to-Peer Systems*, 2003.

[68] Chris Karlof, Naveen Sastry, Yaping Li, Adrian Perrig, and J.D. Tygar. Distillation codes & applications to DoS resistant multicast authentication. In *Proceedings of Eleventh Annual Network and Distributed Systems Security Symposium —NDSS '04*, 2004.

[69] Charles Kaufman, Radia Perlman, and Michael Speciner. *Network Security: Private Communication in a Public World*. Prentice-Hall, Englewood Cliffs, NJ, 1995.

[70] Valerie King. A simpler minimum spanning tree verification algorithm. In *Proceedings of Workshop on Algorithms and Data Structures*, pages 440–448, 1995.

[71] D. Knuth. *The art of computer programming*. Addison-Wesley, 1973.

[72] P. C. Kocher. On certificate revocation and validation. In *Proceedings of International Conference on Financial Cryptography*, volume 1465 of *LNCS*, pages 172–177. Springer-Verlag, 1998.

[73] Henry F. Korth and Abraham Silberschatz. *Database system concepts*. McGraw-Hill, Inc., New York, NY, USA, 1986.

[74] K. Kothapalli and C. Scheideler. Supervised peer-to-peer systems. In *Proceedings of 2005 International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN)*, page 6, 2005.

[75] Dieter Kratsch, Ross M. McConnell, Kurt Mehlhorn, and Jeremy P. Spinrad. Certifying algorithms for recognizing interval graphs and permutation graphs. In *SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 158–167, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[76] Hugo Krawczyk. Distributed fingerprints and secure information dispersal. In *Proceedings of 13th ACM Symposium on Principles of Distributed Computing*, pages 207–218. ACM, 1993.

[77] Maxwell Krohn, Michael Freedman, and David Mazieres. On-the-fly verification of rateless erasure codes for efficient content distribution. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 226–240, May 2004.

[78] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structures for outsourced databases. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 121–132, 2006.

[79] Ji Li, Karen Sollins, and Dah-Yoh Lim. Implementing aggregation and broadcast over distributed hash tables. *ACM SIGCOMM Computer Communication Review*, 35(1):81–92, 2005.

[80] M. Luby. LT codes. In *Proceedings of 43rd Annual IEEE Symposium on Foundations of Computer Science (FOCS '02)*, pages 271–280, 2002.

[81] Michael G. Luby, Michael Mitzenmacher, M. Amin Shokrollahi, and Daniel A. Spielman. Efficient erasure correcting codes. *IEEE Transactions on Information Theory*, 47(2):569–584, February 2001.

[82] Anna Lysyanskaya, Roberto Tamassia, and Nikos Triandopoulos. Multicast authentication in fully adversarial networks. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 241–255, May 2004.

[83] Petros Maniatis and Mary Baker. Enabling the archival storage of signed documents. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST 2002)*, page 3, Monterey, CA, USA, 2002.

[84] Petros Maniatis and Mary Baker. Secure history preservation through timeline entanglement. In *Proceedings of USENIX Security Symposium*, pages 297–312, 2002.

[85] Gurmeet Singh Manku, Mayank Bawa, and Prabhakar Raghavan. Symphony: Distributed hashing in a small world. In *Proceedings of 4th USENIX Symposium on Internet Technologies and Systems*, pages 127–140, 2003.

[86] Chip Martel, Glen Nuckolls, Premkumar Devanbu, Michael Gertz, April Kwong, and Stuart G. Stubblebine. A general model for authenticated data structures. *Algorithmica*, 39(1):21–41, 2004.

[87] Conrado Martínez and Salvador Roura. Randomized binary search trees. *Journal of the ACM*, 45(2):288–323, 1998.

[88] K. Mehlhorn, S. Näher, M. Seel, R. Seidel, T. Schilz, S. Schirra, and C. Uhrig. Checking geometric programs or verification of geometric structures. *Computational Geometry: Theory and Applications*, 12(1–2):85–103, 1999.

[89] Kurt Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*, volume 1 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Heidelberg, Germany, 1984.

[90] Kurt Mehlhorn, Arno Eigenwillig, Kanela Kanegossi, Dieter Kratsch, Ross McConnel, Uli Meyer, and Jeremy Spinrad. Certifying algorithms (A paper under construction). Manuscript, 2005. Available at `http://www.mpi-sb.mpg.de/~mehlhorn/ftp/CertifyingAlgorithms.pdf`.

[91] Kurt Mehlhorn and Stefan Näher. *Checking Geometric Structures*, December 1996. Program Documentation.

[92] Kurt Mehlhorn and Stefan Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, UK, 2000.

[93] Kurt Mehlhorn, Stefan Näher, Thomas Schilz, Stefan Schirra, Michael Seel, Raimund Seidel, and Christian Uhrig. Checking geometric programs or verification of geometric structures. In *Proceedings of 12th Annual ACM Symposium on Computational Geometry*, pages 159–165, 1996.

[94] R. C. Merkle. Protocols for public key cryptosystems. In *Proceedings of Symposium on Security and Privacy*, pages 122–134. IEEE Computer Society Press, 1980.

[95] Ralph C. Merkle. A certified digital signature. In G. Brassard, editor, *Proceedings CRYPTO '89*, volume 435 of *LNCS*, pages 218–238. Springer-Verlag, 1989.

[96] Silvio Micali. Efficient certificate revocation. Technical Report TM-542b, MIT Laboratory for Computer Science, 1996.

[97] Silvio Micali, Chris Peikert, Madhu Sudan, and David A. Wilson. Optimal error correction against computationally bounded noise. In *Proceedings of 2nd Theory of Cryptography Conference (TCC)*, pages 1–16, 2005.

[98] Silvio Micali, Michael Rabin, and Joe Kilian. Zero-ksnowledge sets. In *Proceedings of 44th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 80–91, 2003.

[99] Daniele Micciancio and Saurabh Panjwani. Optimal communication complexity of generic multicast key distribution. In *Advances in Cryptology — EUROCRYPT 2004*, volume 3027 of *Lecture Notes in Computer Science*, pages 153–170. Springer Verlag, 2004.

[100] Sara Miner and Jessica Staddon. Graph-based authentication of digital streams. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 232–246, 2001.

[101] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, New York, NY, 1995.

[102] Einar Mykletun, Maithili Narasimha, and Gene Tsudik. Authentication and integrity in outsourced databases. In *Proceeding of Network and Distributed System Security (NDSS)*, 2004.

[103] Moni Naor and Kobbi Nissim. Certificate revocation and certificate update. In *Proceedings of 7th USENIX Security Symposium*, pages 217–228, Berkeley, 1998.

[104] Moni Naor and Guy N. Rothblum. The complexity of online memory checking. In *Proceedings of 46th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 573–584, 2005.

[105] Lan Nguyen. Accumulators from bilinear pairings and applications. In *Proceedings of CT-RSA*, volume 3376 of *Lecture Notes in Computer Science*, pages 275–292. Springer-Verlag, 2005.

[106] Glen Nuckolls. Verified query results from hybrid authentication trees. In *Proceedings of Database Security 05*, pages 84–98, 2005.

[107] Glen Nuckolls, Chip Martel, and Stuart Stubblebine. Certifying data from multiple sources [extended abstract]. In *Proceedings of the 4th ACM conference on Electronic commerce*, pages 210–211, New York, NY, USA, 2003. ACM Press.

[108] Rafail Ostrovsky, Charles Rackoff, and Adam Smith. Efficient consistency proofs for generalized queries on a committed database. In *Proceedings of 31st International Colloquium on Automata, Languages and Programming (ICALP)*, pages 1041–1053, 2004.

[109] M. H. Overmars. *The Design of Dynamic Data Structures*, volume 156 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, West Germany, 1983.

[110] Alain Pannetrat and Refik Molva. Efficient multicast packet authentication. In *Proceedings of Network and Distributed System Security Symposium — NDSS '03*, 2003.

[111] Jung Min Park, Edwin K. P. Chong, and Howard Jay Siegel. Efficient multicast packet authentication using signature amortization. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 227–240, 2002.

[112] Jung Min Park, Edwin K. P. Chong, and Howard Jay Siegel. Efficient multicast packet authentication using erasure codes. *ACM Transactions on Information and System Security*, pages 6(2):258–285, May 2003.

[113] Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In Joan Feigenbaum, editor, *Advances in Cryptology – CRYPTO '91*, volume 576 of *Lecture Notes in Computer Science*, pages 129–140. Springer Verlag, 1992.

[114] A. Perrig. The BiBa one-time signature and broadcast authentication protocol. In *Proceedings of 8th ACM Conference on Computer and Communication Security*, pages 28–37, November 2001.

[115] Adrian Perrig, Ran Canetti, Dawn Song, and J.D. Tygar. Efficient and secure source authentication for multicast. In *Proceedings of Network and Distributed System Security Symposium – NDSS '01*, 2001.

[116] Adrian Perrig, Ran Canetti, J.D. Tygar, and Dawn Song. Efficient authentication and signing of multicast stream over lossy channels. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 56–73, 2000.

[117] C. Greg Plaxton, Rajmohan Rajaraman, and Andrea W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of ACM Symposium on Parallel Algorithms and Architectures*, pages 311–320, June 1997.

[118] Daniel J. Polivy and Roberto Tamassia. Authenticating distributed data using Web services and XML signatures. In *Proceedings of ACM Workshop on XML Security*, pages 80–89, 2002.

[119] F. P. Preparata. A new approach to planar point location. *SIAM Journal on Computing*, 10(3):473–482, 1981.

[120] W. Pugh. Skip list cookbook. Technical Report CS-TR-2286, Department of Computer Science, University of Maryland, College Park, MD, July 1989.

[121] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[122] Michael O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of ACM*, 36(2):335–348, 1989.

[123] Sriram Ramabhadran, Joseph Hellerstein, Sylvia Ratnasamy, and Scott Shenker. Prefix hash tree - an indexing data structure over distributed hash tables. In *Proceedings of ACM symposium on Principles of distributed computing*, pages 368–368, 2004.

[124] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard M. Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of SIGCOMM*, pages 161–172, 2001.

[125] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *SIAM Journal of Applied Mathematics*, 8(2):300–304, 1960.

[126] Sean Rhea, Brighten Godfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu. OpenDHT: A public DHT service and its uses. In *Proceedings of 2005 ACM SIGCOMM Conference*, pages 73–84, 2005.

[127] Ohad Rodeh, Kenneth P. Birman, and Danny Dolev. Using AVL trees for fault tolerant group key management. *International Journal on Information Security*, pages 84–99, 2001.

[128] Pankaj Rohatgi. A compact and fast hybrid signature scheme for multicast packet authentication. In *Proceedings of 6th ACM Conference on Computer and Communications Security*, pages 93–100. ACM, 1999.

[129] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, LNCS 2218:329, 2001.

[130] J. Saia, A. Fiat, S. Gribble, A. Karlin, and S. Saroiu. Dynamically fault-tolerant content addressable networks. In *Proceedings of 1st International Workshop on Peer-to-Peer Systems, MIT Cambridge, MA*, pages 270–279, 2002.

[131] R. Sedgewick. *Algorithms in C++*. Addison-Wesley, Reading, MA, 1992.

[132] Gustavus J. Simmons. Authentication theory/coding theory. In *Proceedings of the Conference on Advances in Cryptology (CRYPTO'84, Santa Barbara,CA), LNCS 196, Springer-Verlag*, pages 411–431, 1984.

[133] E. Sit and R. Morris. Security considerations for peer-to-peer distributed hash tables. In *Proceedings of International Workshop on P2P Systems*, pages 261–269, 2002.

[134] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer Systems Science*, 26(3):362–381, 1983.

[135] Jack Snoeyink, Subhash Suri, and George Varghese. A lower bound for multicast key distribution. In *Proceedings of INFOCOMM*, pages 422–431, 2001.

[136] Dawn Song, David Zuckerman, and J. D. Tygar. Expander graphs for digital stream authentication and robust overlay networks. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 258–27, 2002.

[137] Douglas R. Stinson. *Cryptography: Theory and Practice, Second Edition*. CRC Press Series, 2002.

[138] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

[139] G. F. Sullivan and G. M. Masson. Certification trails for data structures. In *Digest of the 21st Symposium on Fault-Tolerant Computing*, pages 240–247. IEEE Computer Society Press, 1991.

[140] G. F. Sullivan, D. S. Wilson, and G. M. Masson. Certification of computational results. *IEEE Trans. Comput.*, 44(7):833–847, 1995.

[141] Roberto Tamassia. Authenticated data structures. In *Proceedings of European Symposium on Algorithms*, volume 2832 of *Lecture Notes in Computer Science*, pages 2–5. Springer-Verlag, 2003.

[142] Roberto Tamassia and Nikos Triandopoulos. Computational bounds on hierarchical data processing with applications to information security. In *Proceedings of International Colloquium on Automata, Languages and Programming (ICALP)*, volume 3580 of *LNCS*, pages 153–165. Springer-Verlag, 2005.

[143] D. M. Wallner, E. G. Harder, and R. C. Agee. RFC 2627 – Key management for multicast: issues and architecture, September 1998.

[144] Mark Weiser. Some computer science issues in ubiquitous computing. *Communications of the ACM*, 36(7):75–84, 1993.

[145] L. Welch and E. Berlekamp. Error correction of algebraic block codes. U.S. Patent Number 4,633,470, issued December 1986.

[146] J. Westbrook and R. E. Tarjan. Maintaining bridge-connected and biconnected components on-line. *Algorithmica*, 7:433–464, 1992.

[147] C. K. Wong, M. Gouda, and S. S. Lam. Secure group communications using key graphs. *IEEE/ACM Transactions on Networking*, 8(1):16–30, 2000.

[148] Chung Kei Wong and Simon S. Lam. Digital signatures for flows and multicasts. *IEEE/ACM Transactions on Networking*, 7(4):502–513, August 1999.

[149] Shouhuai Xu and Ravi Sandhu. Authenticated multicast immune to denial-of-service attack. In *Proceedings of ACM Symposium on Applied Computing*, pages 196–200, Madrid, Spain, March 2002.

[150] C. Zhang, A. Krishnamurthy, and R. Wang. Brushwood: Distributed trees in peer-to-peer systems. In *Proceedings of 4th International Workshop on Peer-to-Peer Systems (IPTPS05)*, 2005.