Function-specific schemes for verifiable computation

BOSTON UNIVERSITY

GRADUATE SCHOOL OF ARTS AND SCIENCES

Dissertation

FUNCTION-SPECIFIC SCHEMES FOR VERIFIABLE COMPUTATION

by

DIMITRIOS PAPADOPOULOS

Diploma, National Technical University of Athens, 2010

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2016

Approved by

First Reader

Sharon Goldberg, PhD
Associate Professor of Computer Science

Second Reader

Nikos Triandopoulos, PhD
Adjunct Assistant Professor of Computer Science

Third Reader

Charalampos Papamanthou, PhD
Assistant Professor of Electrical and Computer Engineering
University of Maryland

The gap between theory and practice in theory
is not as large as the gap between theory and practice in practice.

-anonymous

Acknowledgments

These past five years have been a beautiful educational journey and numerous individuals have helped me navigate my way through it.

First and foremost, I am grateful to my advisor, Nikos Triandopoulos, without the guidance of whom I could not have faced the numerous challenges of graduate school. From day one, Nikos strove to teach me how to conduct quality research, in an honest self-critical way, and with attention to detail. He always managed to reach out to me, with a combination of firm guidance, mentoring, positive attitude, and the inherent kindness of his character. Under his dual role as a faculty member at Boston University and as a scientist conducting corporate research, he was in a unique position to help me achieve a spherical understanding of our role as researchers and the implications of our work, as well as show me a multitude of possible paths for my personal advancement in either of the two worlds. I will always be grateful for the opportunity he gave me to work with him and the valuable lessons he taught me but, most importantly, for having the honor of calling him a friend.

Secondly, I want to thank Sharon Goldberg for her invaluable assistance throughout this journey. I am grateful for the opportunity she gave me to assist her in teaching courses at Boston University, an opportunity that later evolved into a close personal collaboration. I will forever remember the long hours she put into helping me improve my work and presentation skills, and our long technical discussions. With her diverse research interests she opened up new paths for me and helped me bridge the gap between theoretical research and real-world problems.

Special thanks to Charalampos Papamanthou and Stavros Papadopoulos who have been my collaborators and coauthors in multiple works throughout these years. Charalampos has always provided constructive guidance and his counseling has not only helped me improve my research but also to better understand relations between topics that seemed disjoint to me. My close collaboration with Stavros has proven to be a valuable lesson in academic research, in more than one occasions. His strict work ethics and his hands-on approach to research will always be a bright example.

I also feel the deep need to thank Ran Canetti and Leonid Reyzin. I consider myself truly blessed to have been given the chance to discuss research topics and coauthor papers with both of them. Ran's classes at Boston University were my gateway to cryptographic research; his formal treatment of the topic and his academic rigorousness have served as a golden standard for me ever since. Throughout my years at graduate school, Leo's deep knowledge in a vast array of topics never ceased to amaze me. His feedback has been invaluable on multiple occasions. At this point, I want to stress what an amazing academic environment the BU Security group has been for me. The number of interesting ideas that bounce around its corridors on a daily basis is fascinating, but what is truly unbelievable is the extent to which everyone makes themselves available to others, in order to provide feedback, discuss ideas, or offer support. Many places can claim to have an open-door policy, but I can safely say that, at BUSec, office doors may easily be replaced with bead curtains.

I am grateful to all my other coauthors during my years as a graduate student, namely, Foteini Baldimitsi, Esha Ghosh, Ahmed Kosba, Moni Naor, Olga Ohrimenko, Omer Paneth, Mahmoud Sayed, Alessandra Scafuro, Elaine Shi, Roberto Tamassia, Sachin Vasant, and Asaf Ziv. My collaboration with each of them has greatly benefited me and helped shape my perspective on research. I also want to thank Christian Cachin and Duane Wessels for being my research mentors during my internships at IBM Research and Verisign Labs, respectively. Finally, I have to express my gratitude towards my undergraduate mentor at NTUA, Antonis Symvonis, for preparing me for this educational adventure.

The BU CS department has been an ideal academic environment for me and all faculty members are extraordinary. I am particularly thankful to Azer Bestavros, John Byers, Steve Homer, Assaf Kfoury, George Kollios, Abraham Matta, and Evimaria Terzi for their very well-taught courses and their guidance. Life as a graduate student at BU would have been much harder without the assistance of the administrative and technical staff of the department. Many thanks to Chris, Jennifer, Theresa, Ellen, Nora, Faith, Paul, and Bob, for making our lives easier.

This experience would not have been so enjoyable without a number of people that I shared it with. I want to thank Davide, Natali, Jeff, Foteini, Sokratis, Stavros, Amina, Jimmy, Amy, George, Ben, Alessandra, Sofia, Ioannis, Kostas, Anastasia, Thodoris, Nina, and Xianrui for being such great friends and for always being there, through each hardship and every celebration. A happy life begins with a happy home and, for this, I will always be grateful to Harry Mavroforakis for being the best roommate anyone could have and a true friend.

Back in Greece, my relatives and childhood friends played a huge role in supporting me and galvanizing my determination during this task. I want to thank all my buddies and family friends for their love. Special thanks to Zoi Zaharopoulou and Anastasia Patrikiou for playing a tremendous role in my upbringing, each in her own important way. Finally, I want to thank my aunt Lena, uncle Yiannis, and cousin Iria for being there for me and making each trip home a happy one.

Last but not least, none of this would have been even remotely possible without the support of my family. I will always be grateful to my mother Theopoula for her love and the unconditional sacrifices she made for my upbringing. Her ability to make every disappointment diminish while magnifying every success has been my solace throughout these years. Finally, my wife Eirini has offered her love and support and has patiently suffered with me on a daily basis; a great part of this degree belongs to her.

FUNCTION-SPECIFIC SCHEMES FOR VERIFIABLE COMPUTATION

DIMITRIOS PAPADOPOULOS

Boston University, Graduate School of Arts and Sciences, 2016

Major Professors: Sharon Goldberg, PhD
Associate Professor of Computer Science

Nikos Triandopoulos, PhD
Adjunct Assistant Professor of Computer Science

ABSTRACT

An integral component of modern computing is the ability to outsource data and computation to powerful remote servers, for instance, in the context of cloud computing or remote file storage. While participants can benefit from this interaction, a fundamental security issue that arises is that of integrity of computation: How can the end-user be certain that the result of a computation over the outsourced data has not been tampered with (not even by a compromised or adversarial server)?

Cryptographic schemes for verifiable computation address this problem by accompanying each result with a proof that can be used to check the correctness of the performed computation. Recent advances in the field have led to the first implementations of schemes that can verify arbitrary computations. However, in practice the overhead of these general-purpose constructions remains prohibitive for most applications, with proof computation times (at the server) in the order of minutes or even hours for real-world problem instances. A different approach for designing such schemes targets specific types of computation and builds custom-made protocols, sacrificing generality for efficiency. An important representative of this function-specific approach is an *authenticated data structure (ADS)*, where a specialized protocol is designed that supports query types associated with a particular outsourced dataset.

This thesis presents three novel ADS constructions for the important query types of set operations, multi-dimensional range search, and pattern matching, and proves their security under cryptographic assumptions over bilinear groups. The scheme for set operations can support nested queries (e.g., two unions followed by an intersection of the results), extending previous works that only accommodate a single operation. The range search ADS provides an exponential (in the number of attributes in the dataset) asymptotic improvement from previous schemes for storage and computation costs. Finally, the pattern matching ADS supports text pattern and XML path queries with minimal cost, e.g., the overhead at the server is less than 4% compared to simply computing the result, for all our tested settings. The experimental evaluation of all three constructions shows significant improvements in proof-computation time over general-purpose schemes.

Contents

List of Tables

List of Figures

List of Abbreviations

ADS	Authenticated data structure
CRH	Collision-resistant hash function
DTD	Document type definition
ECRH	Extractable collision-resistant hash function
EREW	Exclusive-read exclusive-write model
FFT	Fast Fourier transform
NP	Non-deterministic polynomial time
PCP	Probabilistically checkable proof
PRF	Pseudorandom function
q-**PKE**	q-Power knowledge of exponent assumption
q-**SBDH**	q-Strong bilinear Diffie-Hellman assumption
$\mathcal{SMA}$	Set membership authentication protocol
SNARK	Succinct non-interactive argument of knowledge
$\mathcal{SOA}$	Set operation authentication protocol
VC	Verifiable computation

Chapter 1

Introduction

An integral component of modern computing is the ability to outsource data and computation to powerful remote servers. Individuals frequently outsource their data to hosting services such as Dropbox and Google Drive, or have their e-mails stored remotely at Gmail servers, in order to be able to access them from everywhere. Following the Database-as-a-Service paradigm, enterprises can utilize cloud services like Amazon EC2 and S3, in order to benefit in terms of storage, computation, and elasticity of resources.

On one hand, all participants stand to gain from this model of interaction. Data owners avoid the need for building a sophisticated and potentially costly infrastructure since storage and computationally intensive tasks are offloaded to a cloud server. Moreover, the server can benefit financially from accommodating a large number of datasets from different parties. On the other hand, new security issues arise in this setting where one's data no longer resides within their "zone-of-trust". One particular such issue is that of *integrity of computation*, i.e, how can an end-user be certain that the result of a query, executed at the remote server, has not been tampered with—even if the server itself is compromised or behaves adversarially (e.g., to bias the competition among rival serviced companies). Ensuring that information remains intact in the lifetime of an outsourced dataset and that query processing is handled correctly, producing correct and up-to-date answers, lies at the foundation of secure cloud services.

In this thesis, we consider a *data owner* that outsources a dataset D to a *server*. The latter is then responsible for responding to informational queries issued by multiple *clients*, answered according to D. This model captures a variety of real-world applications such as outsourced SQL queries, streaming datasets, and outsourced file systems. Furthermore, the owner may choose to modify D, e.g., by inserting or deleting elements, and the server is notified for such changes. In practice, the clients may be collaborators or customers of the data owner, or even the owner himself accessing D from a different device via the server. In this setting clients may want to verify the integrity of the server's answers to protect themselves against servers that behave maliciously, are compromised by an external attacker, or simply provide false data due to bugs. Informally, the clients should, somehow, get a guarantee that the computation is as good as if it was performed by the trusted owner himself. Here, we present cryptographic solutions that provide such a guarantee for three types of computations over outsourced databases. In particular, we construct solutions that allow the clients to check the correctness of remotely executed: (i) nested set operations, (ii) multi-dimensional range queries, and (iii) text and XML pattern matching queries. These three types of computation capture numerous applications in practice. We elaborate more on this in Section 1.3. However, before we can proceed with this, we first need to discuss what are the existing solutions for this problem in the literature, in order to better present our contributions and how our constructions are related to prior works.

In the security literature, researchers have studied three (loosely defined) alternative approaches for achieving such an integrity property. One approach relies on trusted hardware, (e.g., [SZJvD04, PMP11, SLS$^+$05, SSW10]), which requires that the owner can install a secure module at the infrastructure of the server. Another line of works relies on replication (e.g., [CS06, CL02b, CRR13]) and assumes multiple servers, which makes detection of adversarial behavior possible in a straight forward manner (as long as the compromised servers do not collaborate). However, both of the above approaches require modifications in the setting and the mode of interaction among the parties; while they are applicable to many cases, a solution that remains faithful to the model would be more attractive in general.

A cryptographic solution to the problem requires that when answering a client query, the server also computes a *proof of integrity* for the data used to compute the answer as well as the integrity of the computation itself. This solution is purely software-based and prohibits adversarial behavior in a very strong sense: Convincing a client of a false result becomes as hard as breaking a cryptographic assumption. For this purpose, we allow the owner to perform some preprocessing on D before outsourcing it to the server, and to compute and publish a small verification state, hereafter referred to as *digest*, that is used by clients to verify the server's responses. When issuing an update query, the owner needs to also update the digest. If the digest can be made public we say that the server's proofs are *publicly verifiable*, (i.e., any party can issue queries and check the integrity of the result).

In this setting, it is important to minimize the performance overhead, that comes from deploying a cryptographic solution, (e.g., in terms of computation time, communication bandwidth, storage, etc.) in order to avoid negating the aimed benefit from outsourcing in the first place. Several different measures of efficiency need to be considered. First, we would like that the time it takes for the client to verify a proof is short, ideally, some fixed polynomial in the security parameter that is independent of the size of server's computation cost and the size of D. Second, we would like the server's computational overhead for computing proofs to be minimal. Additional efficiency considerations include the proof size and the efficiency of updates. The number of communication rounds should also be minimized. In this thesis, we concentrate on non-interactive solutions where the client sends a query and receives back an answer and a proof in one round of interaction (i.e., remaining faithful to the existing mode of interaction).

1.1 General-purpose verifiable computation

In the cryptographic literature, the problem of provably checking the integrity of arbitrary delegated computations has been formalized under the notion of *verifiable computation (VC)*. The starting point in the area has been the concept of non-interactive arguments (computationally sound proofs) [Mic00] that builds upon the earlier literature on interactive proofs [GMR89] and interactive arguments [BCC88, Kil92], and provides a solution in the random oracle model [BR93]. Since then, a series of works in the cryptographic literature (e.g., [GGP10, BCCT12, CKLR11, GGPR13, PST13]) have revisited the problem with various definitional extensions and improved constructions.[1]

In the setting of verifiable computation, a computationally weak client holds input x and wishes to compute the output of function f on it, by outsourcing the computation to a server. A VC scheme requires that the server accompanies each query result y with a cryptographic proof-of-correctness, that can be efficiently (i.e., much faster than computing $f(x)$) verified by the client. Notice that the above formulation only targets delegation of computation and not of data. The client is assumed to hold the input dataset and simply wishes to avoid the (possibly large) computation cost. However, this is not a limiting factor when it comes to delegations of storage. A folklore result from the literature shows that the client can outsource his input x ahead of time, and maintain only a succinct collision-resistant representation c, e.g., the hash $h(x)$ under a collision-resistant hash function h. If the function that

[1]We refer interested readers to [WB15] for an introduction to the concept and for an extensive overview of the existing techniques and literature.

the client wishes to have computed by the server over his input x is f, then let f' be defined as the function that upon input values x, c outputs $f(x)$ only if $c = h(x)$. Then the client can keep locally $c = h(x)$ as his digest and deploy a VC scheme for the function f' instead of f. Indeed, existing works further explore this technique (e.g., [CKLR11, WSR$^+$15, sBFR15]) to outsource data.

This approach requires a VC scheme that can accommodate the class of NP (since, computing $f'(x)$ without access to x—as is the case for the client—belongs to this class). The first such scheme was [Mic00] which utilized probabilistically checkable proofs, a fact that would significantly impact its performance in practice. One approach for designing a more efficient protocol is based on replacing computationally sound proofs with succinct non-interactive arguments of knowledge (SNARK). Good candidates for such a SNARK include the works of [BCCT12, PHGR13, BCG$^+$13, Gro16] that provide constructions with very small asymptotic overhead.[2]

Although the problem was originally studied from a theoretical perspective, the first VC implementations for arbitrary computations, were presented recently [WSR$^+$15, VSBW13, PHGR13, BFR$^+$13b, BCG$^+$13, BCG$^+$14, BCG$^+$14, BCTV14, CFH$^+$15]. However, the concrete overhead of the above implementations, when applied for outsourcing of datasets, remains well beyond the realm of reality, with proof computation times (at the cloud server) in the order of minutes or even hours for real-world problem instances (particularly as the size of the dataset grows). This overhead comes largely from the generality of existing schemes. In order to be able to accommodate large classes of functions (e.g., every function in $\mathcal{P}$), a generic function representation is necessary. Existing implementations, have to rely on (Boolean or arithmetic) circuit representation, or probabilistically check-

[2]We do not make the distinction between a SNARK with or without a long pre-processing phase since, in our setting, the owner anyhow performs a preprocessing over the entire dataset before outsourcing it.

able proofs, and both approaches may come with significant costs, heavily depending on the particular type of computation that is outsourced. For example, [PHGR13] shows that for computing a matrix by vector multiplication the proof computation for 1000 dimensions is close to 0.9ms, which is definitely not prohibitive. This type of computation, however, has a naturally “good” arithmetic circuit representation. For other computations, this cost scales much worse, e.g., [ZPK14] reports a cost off approximately 19,000 hours for performing a BFS traversal over a graph with 9,000 edges using the same system, and approximately 52 hours using the optimized VC scheme of [BCG$^+$13]. This costs stems, to a large extent, from the fact that this type of computation does not have a nice arithmetic circuit representation. In the context of set operations (a problem we address in Chapter 3), the transformation from formulas of set operations to circuits can be extremely wasteful as the number of sets participating in every query and the set sizes (including the size of the answer) may vary dramatically between queries.

1.2 Function-specific schemes for verifiable computation

There is a large number of works in the literature that take a function-specific approach for verifiable computation, i.e., they target specific types of computation and try to build more efficient schemes tailored for the problem at hand. This line of literature has evolved largely independently of the general-purpose schemes, motivated by particular real-world problems and proposing solutions with great potential for deployment in practice. As an example, some of the earliest works in the area (e.g., [NN00]) solved the problem of authenticated certificate management and revocation. Other interesting examples from the literature include solutions for matrix multiplication [FG12], polynomial arithmetic [BGV11, BFR13a], set operations [PTT11], relational database queries [YPPK09a, ZKP15], pattern matching [MND$^+$04], polynomial differentiation [PST13], and many more.

1.2.1 Authenticated data structures

One particular line of work that follows this approach is *authenticated data structures (ADS)* [Tam03], where a specific type of data structure is targeted and a customized protocol is built, that supports all query types associated with this data structure (e.g., lookups and insertions for the case of hash tables). ADS schemes can be seen as an alternative to general-purpose VC that sacrifices generality for efficiency, but they also impose additional realistic requirements such as, support for efficient updates in the dataset, and public verifiability (i.e., once the data has been outsourced, anyone can verify the integrity of an operation). The challenge then lies in identifying classes of databases and query types that are relevant in practice, and developing cryptographic protocols specifically for them, that are much faster than general-purpose VC.

Slightly more formally, an *authenticated data structure* (ADS) is a protocol for secure data outsourcing involving the owner of a dataset (also referred to as the source), an untrusted server and multiple clients that issue queries over the dataset. The interaction model is depicted in Figure 1·1. The protocol consists of a pre-processing phase where the source uses a secret key to compute some authentication information over the dataset D, outsources D along with this information to the server and publishes some public digest d related to the current state of D. Subsequently, the source can issue update queries for D (which depend on the data type of D), in which case the source updates the digest and both the source and the server update the authentication information to correspond consistently with the updated dataset state. Moreover, multiple clients (including the source itself), issue queries q addressed to the server, which responds with appropriate answer α and proof of correctness Π. Responses can be verified both for integrity of computation of q and

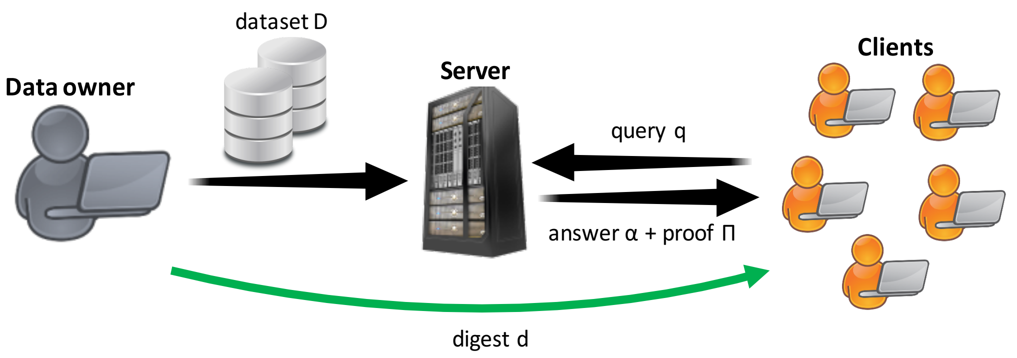


Figure 1·1: The interaction model of authenticated data structures. Initially, the data owner computes a digest d of his dataset D which he publishes. Subsequently he outsources D to a server who is responsible for handling queries q issued by multiple clients. The server is considered untrusted and is required to respond to q with the answer α together with a proof Π which is used by the clients alongside d to verify the integrity of α.

integrity of data used (i.e., that the correct query was run on the correct dataset D) with access only to public key information and digest d. From a security perspective, the service offered to clients is that the received answers are "as-good-as" being directly computed by the trusted source. A restricted version of this setting is a two-party model where the owner of D outsources it to a server and issues updates and queries, benefiting in both storage and computation cost.

ADS constructions exist for many popular data structure types, such as, lists [NN00], trees [MND$^+$04, NN00, PP15], hash tables [PTT15], skip lists [GTS01], inverted index data structures [PTT11], and graph databases [ZPK14]. The earlier of these constructions roughly follow the general blueprint of [MND$^+$04] that applies for data structures, where the query evaluation can be modeled as a search process among predefined elements (e.g., a tree traversal from its root). For each step of the search, an atomic proof must be produced by the server that validates that the correct choice at that step was made. Security in this setting is achieved by employing a cryptographic hash function to encode each link in the data structure (e.g., a parent-child node relation). This concept has been revisited recently in [MHKS14] that provides an automated compiler that can take any such data structure and provide its authenticated version. However, this approach comes with some downsides, namely that the verification cost at the client is as high as the proof construction cost at the server, and the type of computations that can be supported are limited to whatever can be expressed a search process. A different approach for building ADS schemes was proposed in [TT10], where the authentication of general query results is reduced to the certification of set membership relations among predefined (and constructed in a way related to specific problem) sets, which —through careful use of cryptographic primitives such as accumulators [BdM93, CL02a, Ngu05]— allows for faster verification and supports more general query types.

1.3 Three novel ADS constructions

In this dissertation, we present three novel ADS constructions for the cases of set operations, multi-dimensional range search, and pattern matching. The main motivation for these types of computation comes from the numerous applications they find in real-life, ranging from filtered keyword search (in the case of set operations), to SQL queries (a fundamental type of which is a range query) and problems associated with computational biology and intrusion detection (in the case of pattern matching).

Our constructions extend the known literature of function-specific VC schemes both in terms of expressiveness (for set operations) and efficiency (for range queries and pattern matching). Next, we present a brief overview of the three constructions.

1.3.1 Nested set operations

The focus of the first part of the dissertation is the problem of verifiable nested set operations in an outsourced setting. The results discussed were originally presented in [CPPT14]. The motivation for set operations comes naturally due to the numerous computations that can be mapped by them, such as a wide class of SQL database queries, authenticated keyword search with elaborate queries, access control management, and similarity measurement.

We consider a dataset that consists of multiple sets, where the clients' queries are arbitrary set operations among them, represented as formulas of *nested* unions, intersections, and set difference. The verification cost is asymptotically optimal, i.e., the same as simply parsing the query and the answer, whereas the server only suffers a poly-logarithmic overhead. The scheme also supports two types of efficient updates: source updates and server-assisted updates. The former assumes that the owner stores the dataset locally (i.e., only benefits from delegating handling the query load to the server). This greatly simplifies the update process, as the owner can simply compute the new digest himself and publish it for everyone to access, as well as notify the server for any modifications. The latter assume that the owner does not store the dataset locally; the only copy of the data resides at the server. Therefore the update process is a little more involved. First, the owner (and only him) requests an update from the server. Then, the latter performs the update and responds with a "candidate" new digest that he computed using only public key information. Subsequently, the owner runs a verification process on the new digest and either accepts it (in which case he signs and publishes it) or rejects it (in which case he suspects that the server tried to cheat).

The construction extends that of [PTT11] which can only support a single operation at a time. A trivial way to accommodate a nested query (e.g., two unions followed by an intersection) with [PTT11] would be to separately verify each intermediate result; however, this would require that all these results (the unions in the above example) are sent to the client, which increases verification cost and communication accordingly. Our result achieves verification of the validity of the final result in a way that is entirely independent of the sizes of intermediate sets. The security of our construction relies on a modified version of the extractable collision-resistant hash function (ECRH) construction, introduced in [BCCT12], that can be used to succinctly hash univariate polynomials. Finally, our experimental evaluation demonstrates the low verification cost of our scheme (less than 1 second for sets of a few thousand elements), as well as an up to three orders of magnitude improvement for the server's overhead, compared to the only other alternative approach that can accommodate this class of computation (i.e., general purpose VC).

1.3.2 Multi-dimensional range queries

The second part of the dissertation focuses on the verification of multidimensional range queries and contains results originally presented in [PPT14]. Here, the outsourced database is a table that contains tuples with multiple attribute values. A range query is defined over a choice of these attributes (referred to as query dimensions) and it is expressed as a series of pairs of minimum and maximum values, each along a certain dimension. Its result includes all the tuples whose value on all of these dimensions is within the range specified by the query. This query exists in most predominant database architectures; in relational databases this is a `SELECT...FROM...WHERE` query, whereas in scientific databases, such as SciDB, it is a part of the core `SUBARRAY` query.

Our constructions are the first where all costs (i.e., setup, storage, update, proof construction, verification, and proof size) grow only linearly with the number of dimensions. For comparison, in all existing works with provable non-trivial bounds, all these costs scaled exponentially with the number of dimensions. Our core idea is the reduction of a multidimensional range query to multiple 1-dimensional ones, in a way that allows the final result to be expressed as a combination of each of the simpler range queries. This is achieved via a fusion of existing and novel proof tools, based on bilinear accumulators. Moreover, we show how our construction can be modified to achieve faster updates (from linear in the size of the database to order of square root of the size of the database, in the worst case). Our experimental evaluation demonstrates the very low verification cost (less than 3 seconds for all our tested settings) and the corresponding proof construction cost at the server that is considerable but not prohibitive (ranging from order of milliseconds to a few minutes), even for arguably "large" queries.

1.3.3 Pattern matching

The third part of this dissertation focuses on verifiable pattern matching queries, a problem with potential applications in a wide range of topics including intrusion detection, spam filtering, web search engines, molecular biology, and natural language processing. The presented results originally appeared in [PPTT15]. The problem setting involves an outsourced textual database, a query containing a text pattern, and an answer regarding the presence or absence of the pattern in the database. In its simplest form, the database consists of a single text from an alphabet where a query for a specific pattern results in answer "match at position i", or "mismatch". More elaborate models involve queries expressed as regular expressions, and databases allowing search over (semi-)structured data (e.g., XML data).

Our ADS construction is based on an authenticated version of the suffix tree data structure and it provides precomputed (thus, fast to retrieve), constant-size proofs for any basic form of pattern matching query, at no asymptotic increase of storage. Moreover, the proof size is optimal (i.e., the total communication cost is asymptotically the same as simply transmitting the answer) and entirely independent of the size of the text, the queried pattern, or the underlying alphabet, and the verification cost is very small, and it scales quasi-linearly with the query size. Finally, the proof construction time is asymptotically the same as the time it takes to compute the result, i.e., there is no asymptotic overhead. On top of that, our experimental evaluation demonstrates that the concrete cost for poof construction is very small in practice, for all our tested problem instances. For example, it takes less than 90μs to respond to a query of size 100 characters: 80μs to simply find the (mis)match and less than 10μs to assemble the proof. As additional contribution, we show how our scheme can be modified to accommodate queries over collections of text documents, and exact path queries over XML documents, with similar efficiency properties.

1.4 Thesis outline

Chapter 2 provides some cryptographic preliminaries and establishes notation that will be used in the rest of the dissertation. Chapters 3 - 5 contain our three ADS constructions for the cases of set operations, range queries, and pattern matching respectively. Each chapter begins with an overview of the problem formulation, the main result and techniques used, and a comparison with previous works that address the problem. We then provide the main results with proofs of security and experimental evaluation. Finally, Chapter 6 reviews the main results of this thesis and discusses interesting open problems, such as the selective combination of function-specific and general-purpose VC schemes.

Chapter 2

Cryptographic Preliminaries

In this section, we present notation and cryptographic background that will be used in all of the following chapters. Additional definitions that are only used in the context of a particular scheme only, are included in the corresponding chapter.

We denote with λ the security parameter and with $\nu(\lambda)$ a negligible function. A function $f(\lambda)$ is negligible if for each polynomial function $poly(\lambda)$ and all large enough values of λ, $f(\lambda) < 1/(poly(\lambda)$. We say that an event can occur with negligible probability if its probability of occurrence is upper bound by a negligible function. Respectively, an event takes place with overwhelming probability if its complement takes place with negligible probability. In our technical exposition, we adopt the *access complexity* model: Used mainly in the memory checking literature [BEG$^+$94, DNRV09], this model allows us to measure complexity expressed in the number of primitive cryptographic operations made by an algorithm without considering the related security parameter. For example, an algorithm making k modular multiplications over a group of size $O(n)$ where n is $O(exp(\lambda))$ for a security parameter λ, runs in time $O(k \log n)$. In the access complexity model, this is $O(k)$ ignoring the "representation" cost for each group element

Collision-resistant hash functions. A collision-resistant hash function h is a function randomly sampled from a function ensemble, such that no poly-size algorithm can output x, x', such that $h(x) = h(x')$ and $x \neq x'$, except with probability $\nu(\lambda)$. More formally:

Definition 1 (Collision-resistant hash function ensemble (CRH)). *Let $d(\lambda), t(\lambda)$ be polynomial functions of λ. A function ensemble $\mathcal{H} = \{\mathcal{H}_\lambda\}_{\lambda \in \mathbb{N}}$ from $\{0,1\}^{d(\lambda)}$ to $\{0,1\}^{t(\lambda)}$ is a CRH if:*

Collision-resistance *For any poly-size adversary $\mathcal{A}$:*

$$\Pr_{h \leftarrow \mathcal{H}_\lambda}\left[x, x' \leftarrow \mathcal{A}(1^\lambda, h) \text{ } s.t. \text{ } h(x) = h(x') \wedge x \neq x'\right] \leq \nu(\lambda) \ .$$

2.1 Authenticated data structures

An authenticated data structure (ADS) is a cryptographic primitive for proving the correctness of the result of a query on a remote dataset. The interaction model assumes three types of parties: an *owner* holding a data structure D who wishes to outsource it to a *server* who is, in turn, responsible for answering queries issued by multiple *clients*. The owner runs a pre-processing step over D, producing some cryptographic authentication information $auth(D)$ and a succinct digest d of D, and signs d. The server is untrusted, i.e., it may modify the returned answer, hence it is required to provide a proof of the answer, generated using $auth(D)$, and the signed digest d. A client with access to the public key of the owner can subsequently check the proof and verify the integrity of the answer. More formally, an authenticated data structure scheme is a collection of the following six polynomial-time algorithms:

1. $\{sk, pk\} \leftarrow \textbf{genkey}(1^\lambda)$. Outputs secret and public keys sk and pk, given the security parameter l.

2. $\{auth(D_0), d_0\} \leftarrow \textbf{setup}(D_0, sk, pk)$: Computes the authenticated data structure $auth(D_0)$ and its respective digest, d_0, given data structure D_0, the secret key sk and the public key pk.

3. $\{D_{h+1}, auth(D_{h+1}), d_{h+1}, upd\} \leftarrow \textbf{update}(u, D_h, auth(D_h), d_h, sk, pk)$: On input update u on data structure D_h, the authenticated data structure $auth(D_h)$

and the digest d_h, it outputs the updated data structure D_{h+1} along with $auth(D_{h+1})$, the updated digest d_{h+1} and some relative information *upd*. It requires the secret key for execution.

4. $\{D_{h+1}, auth(D_{h+1})d_{h+1}\} \leftarrow$ **refresh**$(u, D_h, auth(D_h), d_h, upd, pk)$: On input update u on data structure D_h, the authenticated data structure $auth(D_h)$, the digest d_h and relative information *upd* output by **update**, it outputs the updated data structure D_{h+1} along with $auth(D_{h+1})$ and the updated digest d_{h+1}, without access to the secret key.

5. $\{a(q), \Pi(q)\} \leftarrow$ **query**$(q, D_h, auth(D_h), pk)$: On input query q on data structure D_h and $auth(D_h)$ it returns the answer to the query $a(q)$, along with a proof $\Pi(q)$.

6. {accept/reject} $\leftarrow$ **verify**$(q, a(q), \Pi(q), d_h, pk)$: On input query q, an answer $a(q)$, a proof $\Pi(q)$, a digest d_h and pk, it outputs either "accept" or "reject".

The notation $a(q), \Pi(q)$ symbolizes that the answer and proof are a function of the particular query q. When it is clear from the context, we will drop this notation and refer simply to a, Π. Let {accept, reject} $=$ *check*$(q, a(q), D_h)$ be a method that decides whether $a(q)$ is a correct answer for query q on data structure D_h (this method is not part of the scheme but only introduced for ease of notation.) Then an authenticated data structure scheme ADS should satisfy the following:

Correctness. We say that an ADS is *correct* if, for all $\lambda \in \mathbb{N}$, for all (sk, pk) output by algorithm **genkey**, for all $(D_h, auth(D_h), d_h)$ output by one invocation of **setup** followed by polynomially-many invocations of **refresh**, where $h \geq 0$, for all queries q and for all $a(q), \Pi(q)$ output by *query*$(q, D_h, auth(D_h), pk)$, with all but negligible probability, whenever **check**$(q, a(q), D_h)$ accepts, so does **verify**$(q, a(q), \Pi(q), d_h, pk)$.

Security. Let $\lambda \in \mathbb{N}$ be a security parameter and $(sk, pk) \leftarrow genkey(1^\lambda)$ and $\mathcal{A}$ be a poly-size adversary that is only given pk and has access to the algorithms of the ADS via an oracle $O_{\lambda,ADS}$ that accepts queries in the following model: The adversary picks an initial state of the data structure D_0 and computes $D_0, auth(D_0), d_0$ through an oracle call to algorithm **setup**. Then, for $i = 0, ..., h = poly(\lambda)$, $\mathcal{A}$ issues an update u_i for the data structure D_i and outputs $D_{i+1}, auth(D_{i+1})$ and d_{i+1} through an oracle call to algorithm **update**. At any point during these update queries, he can make polynomially many oracle calls to algorithms **query** and **verify**. Finally, the adversary picks an index $0 \leq t \leq h+1$, a query q, an answer $a(q)$ and a proof $\Pi(q)$. We say that an ADS is *secure* if for all large enough $\lambda \in \mathbb{N}$, for all poly-size adversaries $\mathcal{A}$ it holds that:

$$\Pr\left[\begin{array}{c} (q, a(q), \Pi(q), t) \leftarrow \mathcal{A}^{O_{\lambda,ADS}}(1^\lambda, pk) \text{ s.t.} \\ \text{accept } \leftarrow \mathbf{verify}(q, a(q), \Pi(q), d_t, pk) \wedge \text{reject } \leftarrow check(q, a(q), D_t)\end{array}\right] \leq \nu(\lambda),$$

where the probability is taken over the randomness of **genkey** and the coins of $\mathcal{A}$.

The above security game captures the fact that an adversary (playing the role of a corrupted server) that interacts with the trusted owner and is given oracle access to all the algorithms of the scheme, cannot come up with a fake result even if he is allowed to chose the contents of the database and the type of query himself. Observe that the only limitation of the adversary is that the digest that will be used for the verification of his final challenge, and the corresponding state of the dataset and authentication information is honestly computed, which maps the way the trusted owner would compute them in the real world.

An ADS is *static* if there is no efficient way to handle updates, i.e., the best way to accommodate such changes is to re-execute setup from scratch. A static ADS consists of only four algorithms {**genkey**,**setup**,**query**,**verify**}. In the security game above the calls to **update** are replaced by calls to **setup**.

2.2 Bilinear groups

Let $\mathbb{G}$ be a cyclic multiplicative group of prime order p, generated by g. Let also $\mathbb{G}_T$ be a cyclic multiplicative group with the same order p and $e : \mathbb{G} \times \mathbb{G} \rightarrow \mathbb{G}_T$ be a bilinear pairing with the following properties: (1) Bilinearity: $e(P^a, Q^b) = e(P, Q)^{ab}$ for all $P, Q \in \mathbb{G}$ and $a, b \in \mathbb{Z}_p$; (2) Non-degeneracy: $e(g, g) \neq 1$; (3) Computability: There is an efficient algorithm to compute $e(P, Q)$ for all $P, Q \in \mathbb{G}$. We denote with $pub := (p, \mathbb{G}, \mathbb{G}_T, e, g)$ the bilinear pairings parameters, output by a randomized polynomial-time algorithm $\mathsf{GenBilinear}$ on input 1^λ.

For cleaner presentation, in what follows we generally assume a symmetric (Type 1) pairing e. All of our constructions can be shown secure in the more efficient asymmetric group case (without a group homomorphism), with small modifications. For example, in Section 3.6 we discuss the modifications needed to implement our construction from Chapter 3, in the asymmetric pairing case (see [CM11] for a general discussion of the importance of difference types of pairings).

Our security analysis makes use of the following two assumptions over groups with bilinear pairings:

Assumption 1 (q-Strong Bilinear Diffie-Hellman [BB08])**.** *For any poly-size adversary $\mathcal{A}$, for q being a parameter of size poly(λ), and for all large enough λ, the following holds:*

$$\Pr\left[\begin{array}{c} pub \leftarrow \mathsf{GenBilinear}(1^\lambda); s \leftarrow_R \mathbb{Z}_p^*; \\ (z, \gamma) \in \mathbb{Z}_p^* \times \mathbb{G}_T \leftarrow \mathcal{A}(pub, (g, g^s, ..., g^{s^q}))\ s.t.\ \gamma = e(g, g)^{1/(z+s)}) \end{array}\right] \leq \nu(\lambda)]\ .$$

Assumption 2 (q-Power Knowledge of Exponent [Gro10])**.** *For any poly-size adversary $\mathcal{A}$, and for all large enough λ, there exists a poly-size extractor $\mathcal{E}$ such that:*

$$Pr\left[\begin{array}{c} pub \leftarrow \mathsf{GenBilinear}(1^\lambda); a, s \leftarrow_R \mathbb{Z}_p^*; \sigma = (g, g^s, ..., g^{s^q}, g^a, g^{as}, ..., g^{as^q}) \\ (c, \tilde{c}) \leftarrow \mathcal{A}(pub, \sigma); (a_0, ..., a_n) \leftarrow \mathcal{E}(pub, \sigma) \\ s.t.\quad e(\tilde{c}, g) = e(c, g^a) \quad \wedge \quad c \neq \prod_{i=0}^{n} g^{a_i s^i} \quad for\ n \leq q \end{array}\right] \leq \nu(\lambda)\ .$$

In the following, we will refer to these two assumptions as q-**SBDH** and q-**PKE**, respectively. It should be pointed out that q-**PKE** (originally introduced by Groth in [Gro10]) is a non-standard assumption, extending the knowledge-of-exponent assumption of [Dam91]. It falls within the classification of non-falsifiable assumptions [Nao03], which has been shown to be necessary in order to construct succinct non-interactive argument systems that are secure in the standard model in [GW11], or equivalently (as shown in [BCCT12]) extractable collision-resistant hash functions such our construction from Section 3.2. In practice, this means that an assumption of this type is necessary for our construction of Section 3.2. More recently, there have appeared impossibility results for knowledge assumptions with arbitrary auxiliary inputs [BCPR14, BP04]. The above formulation of q-**PKE** is not disallowed by these results, as stated (i.e., versus uniform poly-size adversaries) which means that our construction from Section 3 does not suffer from the above impossibility results. Alternatively, we could have defined it with separate auxiliary inputs for the adversary and the extractor, of bounded sizes and coming from specific benign distributions. However, in the context of proving the security of our construction from Chapter 3, we do not need auxiliary input for extraction, therefore we chose this version.

2.3 Bilinear accumulator

A bilinear accumulator [Ngu05] succinctly and securely represents a set of elements from $\mathbb{Z}_p$, operating in the setting of bilinear groups. It represents any set A of n elements from $\mathbb{Z}_p$ by its *accumulation value*, namely

$$\mathsf{acc}(A) = g^{\prod_{a \in A}(s+a)} \in \mathbb{G} \ ,$$

i.e., a single element in $\mathbb{G}$, where $s \in \mathbb{Z}_p$ is trapdoor information that is kept secret. Note that, given values $g, g^s, \ldots, g^{s^n}$ (and without revealing the trapdoor s), $\mathsf{acc}(A)$ can be computed in time $O(n \log n)$ with polynomial interpolation.

Under the q-**SBDH** assumption, the bilinear accumulator provides two security properties: (1) The accumulation function $\mathsf{acc}(\cdot)$ is *collision resistant* [Ngu05] (i.e., it is computationally hard to find different sets with equal accumulation values); and (2) it allows for *reliable verification of subset containment* [PTT11] using short computational proofs; namely, subject to $\mathsf{acc}(A)$, the proof for relation $B \subseteq A$ is defined as the *subset witness* $\mathsf{W}_{B,A} = g^{\prod_{a \in A \setminus B}(s+a)}$. That is, the relation $B \subseteq A$ can be efficiently validated via checking the equality $e(\mathsf{W}_{B,A}, g^{\prod_{b \in B}(s+b)}) \overset{?}{=} e(\mathsf{acc}(A), g)$ given accumulation value $\mathsf{acc}(A)$, set B and public values $g, g^s, \ldots, g^{s^\ell}$, where q is an upper bound on A's size n. However, it is hard to produce a fake subset witness that is verifiable when $B \subseteq A$ is false.

The following lemma formally captures the security property of the bilinear accumulator that will be crucial for the security of our schemes.

Lemma 1 (Security of bilinear accumulator [Ngu05, PTT11])**.** *Let pub* $\leftarrow$ $\mathsf{GenBilinear}(1^\lambda)$ *be a tuple of bilinear pairing parameters and* $s \in \mathbb{Z}_p^*$ *chosen uniformly at random. Under the q-**SBDH** assumption, no poly-size adversary can, upon input* $(pub, g, g^s, \ldots, g^{s^q})$ *output sets* A, B *with elements in* $\mathbb{Z}_p$ *and* $\mathsf{W} \in \mathbb{G}$*, such that: (i)* $B \nsubseteq A$*, and (ii)* $e(\mathsf{W}, \mathsf{acc}(B)) = e(\mathsf{acc}(A), g)$*, except with negligible probability.*

Chapter 3

Verifiable Set Operations

3.1 Introduction

This chapter focuses on the problem of *general set operations* in the outsourced setting. We consider a dataset D that consists of m sets $S_1, ..., S_m$, where the clients' queries are arbitrary set operations over D represented as formulas of union, intersection, and set difference gates over some selection of the sets $S_1, ..., S_m$. The motivation for set operations comes from their great expressiveness and the range of computations that can be mapped by them. Real-world applications of general set operations include a wide class of SQL database queries, authenticated keyword search with elaborate queries, access control management, and similarity measurement, hence an efficient protocol would be of great importance.

3.1.1 Overview of result

We construct a scheme for publicly verifiable secure delegation of set operations. The main advantage of our scheme over general-purpose VC is that it does not involve translating the problem to an arithmetic or boolean circuit, which greatly limits all overheads. This especially important for set operations as a circuit that computes such an operation (e.g., a union of two sets) must either perform a quadratic in the input size number of equality tests, or incorporate a sorting network that first sorts the input sets and then compares the element. The latter approach entails a large concrete cost as such a sorting network requires many gates for its implementation. On the other hand, during roof construction in our scheme the server will need to perform only $4N$ exponentiations in a group with a symmetric bilinear pairing, where N is the sum of the sizes of all the intermediate sets in the evaluation of the set formula. Keep in mind that the cost to simply compute the result is $O(N)$, which highlights that our scheme not only introduces a very small asymptotic overhead, but the hidden constants are particularly small. Moreover, the verification state is of constant size, and the proof verification time is $O(\delta + t)$ where t is the size of the query formula and δ is the answer set size. The dependence on the answer size is inherent since the client must receive the answer set from the server, i.e., the achieved overheads is, in a sense, optimal. We stress that the verification time (and proof length) do not grow with the sizes of all other sets involved in the computation.

Our scheme also supports two types of updates: source updates and server-assisted updates. In a source update, the data owner maintains an an additional update state of length $O(m)$ (m is the number of sets in the dataset) and it can add or remove a single element to a set in constant time. He then updates the server and all other clients with a new verification state. A source update does not require any party to compute any proofs. Server-assisted updates are used to perform updates that change a large number of elements in the dataset and are discussed in more detail in Section 3.5. The basic idea is for the owner to delegate the update to the server (as in [CKLR11]). The owner can set the value of every set by applying a set operation formula to the current state of the dataset. The answer to a server-assisted update query includes a new verification state and a proof that the update was performed correctly. Verifying this proof with the old verification state requires the same time as verifying informational queries and the owner does not need to store any update state (indeed, the owner does not even need to store the dataset).

While this is a more involved process compared to source updates, it has the big benefit of allowing the owner to delegate the storage of the dataset entirely to the server, i.e., he does not need to maintain a local copy anymore.

3.1.2 Overview of techniques

The starting point for the construction is the scheme of Papamanthou, Tamassia and Triandopoulos [PTT11] that supports a single set operation (one union or one intersection). For a query consisting of a single union or intersection over t sets, where the answer set is of size δ, the proof verification time in the scheme of [PTT11] is $O(t + \delta)$. The "naive" way to extend the scheme of [PTT11] to support general set operation formulas is to have the server provide a separate proof for each intermediate set produced in the evaluation of the formula. However, proving the security of this construction is problematic. The problem is that in the scheme of [PTT11] the proofs do not necessarily compose. In particular, it might be easy for a malicious server to come up with a false proof corresponding to an incorrect answer set without "knowing" what this incorrect answer is (if the malicious server would be able to also find the answer set, the scheme of [PTT11] would not have been secure). Therefore, to make the security proof of the naive scheme go through, the server would also have to prove to the client that he "knows" all the intermediate sets produced in the evaluation of the query formula. One way for the server to prove knowledge of these sets is to send them to the client, however, this will result in a proof that is as long as the entire server computation.

Knowledge accumulators. To solve this problem we need to further understand the structure of the proofs in [PTT11]. The construction of [PTT11] is based on the notion of a bilinear accumulator [Ngu05]. We can think of a bilinear accumulator as a succinct hash of a large set that makes use of a representation of a set by its *characteristic polynomial* (i.e., a polynomial that has as roots the set elements). Accumulators have homomorphic properties that allow verifying relations between sets via checking arithmetic relations between their accumulators. The main idea in this work is to use a different type of accumulator that has "knowledge" properties. That is, the only way for an algorithm to produce a valid accumulation value is to "know" the set that corresponds to that value. The knowledge property of our accumulator together with the soundness of the proof for every single operation (one union, intersection, or set difference) allows us to prove the soundness of the composed scheme. Our construction of knowledge accumulators is very similar to previous constructions of knowledge commitments in [BCCT12, Gro10]. The construction is based on the q-**PKE** assumption which is a variant of knowledge-of-exponent assumption [Dam91]. We capture the knowledge properties of our accumulator by using the notion of an *extractable collision-resistant hash function* (ECRH), originally introduced in [BCCT12].[3]

We also need to modify the way a single set operation is proven. For example, in [PTT11], a proof for a single union of sets requires one accumulation value for every element in the union. This will again result in a proof that is as long as the entire server computation. Instead, we change the proof for union so it only involves a constant number of accumulation values.

The verification state and accumulation trees. In order to verify a proof in our scheme, a client only needs to know the accumulation values for the sets that participate in the computation. Instead of containing the accumulation values of all sets in the dataset, the digest contains a single special hash of these accumulation values, making it of constant size. To produce this digest, we hash the accumulation

[3]We follow the weaker definition of ECRH with respect to auxiliary input, for which the recent negative evidence presented in [BCPR14] does not apply and the distributions we consider here are not captured by the negative result of [BP15] either.

values of the sets in the dataset using an *accumulation tree*, introduced in [PTT08]. This primitive can be thought of as a special “tree hash” that makes use of the algebraic structure of the accumulators to gain in efficiency(authentication paths are of constant length).

3.1.3 Prior work

The work of [BFR13a] also considers a practical secure database delegation scheme supporting a restricted class of queries. They consider functions expressed by arithmetic circuits of degree up to 2. Their construction is based on homomorphic MAC’s and their protocol has reasonable performance (e.g., if the computation is described as an arithmetic circuit, the proof computation is less that 3 ms per gate), however their solution is only privately verifiable and it does not support deletions from the dataset. Additionally, we note that the security proof in [BFR13a] is not based on non-falsifiable assumptions. In a sense, that work is complementary to ours, as arithmetic and set operations are two desirable classes of computations for a database outsourcing scheme.

With respect to set operations, previous works focused mostly on the aspect of privacy and less on the aspect of integrity [FNP04, ACT11, KS05, BW07]. There exists a number of works from the database community that address this problem [MND$^+$04, YPPK09a], but to the best of our knowledge, this is the first work that directly addresses the case of nested operations.

Characteristic polynomials for set representation have been used before in the cryptography literature (see for example [PTT11, Ngu05]) and this directly relates this work with a line of publications coming from the *cryptographic accumulators* literature [CL02a, Ngu05]. Indeed our ECRH construction, viewed as a mathematical object, is identical to a pair of bilinear accumulators (introduced in [Ngu05]) with related secret key values. Our ECRH can be viewed as an extractable extension to the bilinear accumulator that allows an adversarial party to prove knowledge of a subset to an accumulated set (without explicitly providing said subset). Indeed, this idea is central to all of our proofs for validity of set operation computations. It also allows us to use the notion of *accumulation trees* which was originally defined for bilinear accumulators.

Our work also highlights the relation between *bilinear accumulators* and commitment schemes originally captured in [KZG10]. In that work, commitments for polynomials over $\mathbb{Z}_p[x]$ are constructed, by essentially the same mathematical operation used to compute accumulation values. Moreover, in [Gro10] a commitment scheme for polynomially many values is presented, again using essentially the same mathematical operation (with an additional blinding factor). An inherent distinction between the two primitives comes from the different frameworks. In the case of polynomial commitment schemes, a committer wishes to produce a commitment which he will later open to a receiver, whereas accumulation values operate as "continuous" commitments to a particular set, for which a prover wishes to prove subset and set membership relations. In particular, there is no explicit hiding property required by an accumulator. However, there is an interesting duality between the two primitives at a mathematical construction level, which we believe can be exploited further. In a sense, polynomial commitments are commitments to the *coefficients* of a polynomial, whereas *accumulation values* operate as commitments to the *roots* of the characteristic polynomial of a set.

3.2 Extractable collision-resistant hash functions

These functions (or ECRH for short) were introduced in [BCCT12] as a strengthening of the notion of collision-resistant hash functions. The key property implied by an ECRH, on top of collision-resistance, is the hardness of oblivious sampling from the image space. Informally, for a function h, sampled from an ECRH function ensemble, any adversary producing a hash value η must have knowledge of a value $x \in Dom(h)$ s.t. $h(x) = \eta$. Formally, an ECRH function is defined as follows:

Definition 2 (Extractable collision-resistant hash function ensemble (ECRH) [BCCT12])**.** *Let $d(\lambda), t(\lambda)$ be polynomial functions of λ. A function ensemble $\mathcal{H} = \{\mathcal{H}_\lambda\}_{\lambda \in \mathbb{N}}$ from $\{0,1\}^{d(\lambda)}$ to $\{0,1\}^{t(\lambda)}$ is an ECRH if:*

Collision-resistance *For any poly-size adversary $\mathcal{A}$:*

$$\Pr_{h \leftarrow \mathcal{H}_\lambda}\left[x, x' \leftarrow \mathcal{A}(1^\lambda, h) \ s.t. \ h(x) = h(x') \wedge x \neq x'\right] \leq \nu(\lambda) \ .$$

Extractability *For any poly-size adversary $\mathcal{A}$, there exists poly-size extractor $\mathcal{E}$ such that:*

$$\Pr_{h \leftarrow \mathcal{H}_\lambda}\left[\begin{array}{c} y \leftarrow \mathcal{A}(1^\lambda, h); x' \leftarrow \mathcal{E}(1^\lambda, h) \\ s.t. \ \exists x : h(x) = y \wedge h(x') \neq y \end{array}\right] \leq \nu(\lambda) \ .$$

An ECRH construction from q-PKE. We next provide an ECRH construction from the q-**PKE** assumption defined above. In [BCCT12] the authors provide an ECRH construction from an assumption that is conceptually similar and can be viewed as a simplified version of q-**PKE** and acknowledge that an ECRH can be constructed directly from q-**PKE** (without explicitly providing the construction). Here we present the detailed construction and a proof of the required properties with respect to q-**PKE** for extractability and q-**SBDH** for collision-resistance.[4]

- To sample from $\mathcal{H}_l$, choose $q \in O(poly(\lambda))$, run algorithm $\mathsf{GenBilinear}(1^\lambda)$ to generate bilinear pairing parameters $pub = (p, \mathbb{G}, \mathbb{G}_T, e, g)$ and sample $a, s \leftarrow_R \mathbb{Z}_p^* \times \mathbb{Z}_p^*$ s.t. $a \neq s$. Output public key $pk = (pub, g^s, ..., g^{s^q}, g^a, g^{as}, ..., g^{as^q})$ and trapdoor information $sk = (s, a)$. It should be noted that the pk fully describes the chosen function h. Trapdoor sk can be used for a more efficient computation of hash values, by the party initializing the ECRH .

[4]It should be noted that while the construction from [BCCT12] is conceptually similar, its collision resistance cannot be proven by a reduction to q-**SBDH**; it is instead provable with a direct reduction to the computation of discrete logarithms in $\mathbb{G}$.

- To compute a hash value on $\mathbf{x} = (x_1, ..., x_q)$, output $h(\mathbf{x}) = \left(\prod_{i\in[q]} g^{x_i s^i}, \prod_{i\in[q]} g^{a x_i s^i}\right)$.

Lemma 2. *If the q-**SBDH** and q-**PKE** assumptions hold, the above is a $(2(q+1)\cdot\lambda, 4\lambda+2)$-compressing ECRH.*

Proof: Extractability follows directly from the q-**PKE** assumption. To argue about collision-resistance, assume there exists adversary $\mathcal{A}$ outputting with probability ϵ, $(\mathbf{x}, \mathbf{y})$ such that there exists $i \in [q]$ with $x_i \neq y_i$ and $h(\mathbf{x}) = h(\mathbf{y})$. We denote with $P(r)$ the q-degree polynomial from $\mathbb{Z}_p[r]$, $\sum_{i\in[q]}(x_i - y_i)r^i$. From the above, it follows that $\sum_{i\in[q]} x_i s^i = \sum_{i\in[q]} y_i s^i$. Hence, while $P(r)$ is not the 0-polynomial, the evaluation of $P(r)$ at point s is $P(s) = 0$ and s is a root of $P(r)$. By applying a randomized polynomial factorization algorithm as in [Ber71], one can extract the (up to q) roots of $P(r)$ with overwhelming probability, thus computing s. By randomly selecting $c \in \mathbb{Z}_p^*$ and computing $\beta = g^{1/(c+s)}$ one can output $(c, e(g, \beta))$, breaking the q-**SBDH** with probability $\epsilon(1-\epsilon')$ where ϵ' is the negligible probability of error in the polynomial factoring algorithm. Therefore any poly-size $\mathcal{A}$ can find a collision only with negligible probability. For a security parameter λ, the input length is $q+1$ elements from $\mathbb{Z}*_p$, each of which can be written with 2λ bits, whereas the output is two elements in $\mathbb{G}$, each of which can be represented in a compressed format by its abscissa plus one bit to indicate the sign of the root of its ordinate, for a total of $4\lambda + 2$ bits. □

One natural application for the above ECRH construction would be the compact computational representation of polynomials from $\mathbb{Z}_p[r]$ of degree $\leq q$. A polynomial $P(r)$ with coefficients $p_0, ..., p_q$ can be succinctly represented by the hash value $h(P) = (f, f') = \left(\prod_{i\in[q]} g^{p_i s^i}, \prod_{i\in[q]} g^{a p_i s^i}\right)$.

3.3 Set representation with polynomials

Sets can be represented with polynomials, using the notion of characteristic polynomial, e.g., as introduced in [FNP04, Ngu05, PTT11]. Given a set $X = \{x_1, .., x_m\}$, the polynomial $\mathcal{C}_X(r) = \prod_{i=1}^{m}(x_i + r)$ from $\mathbb{Z}_p[r]$, where r is a formal variable, is called the *characteristic polynomial* of X (when possible we will denote this polynomial simply by $\mathcal{C}_X$). Characteristic polynomials constitute representations of sets by polynomials that have the additive inverses of their set elements as roots. What is of particular importance to us is that characteristic polynomials enjoy a number of homomorphic properties w.r.t. set operations. For example, given sets A, B with $A \subseteq B$, it must hold that $\mathcal{C}_B|\mathcal{C}_A$ and given sets X, Y with $I = X \cap Y$, $\mathcal{C}_I = gcd(\mathcal{C}_X, \mathcal{C}_Y)$.

The following lemma characterizes the efficiency of computing the characteristic polynomial of a set.

Lemma 3 ([PSaUCCSL76]). *Given set $X = x_1, ..., x_n$ with elements from $\mathbb{Z}_p$, characteristic polynomial $\mathcal{C}_X(r) := \sum_{i=0}^{n} c_i r^i \in \mathbb{Z}_p[r]$ can be computed with $O(n \log n)$ operations with FFT interpolation.*

Note that, while the notion of a unique characteristic polynomial for a given set is well-defined, from elementary algebra it is known that there exist many distinct polynomials having as roots the additive inverses of the elements in this set. In particular, recall that multiplication of a polynomial in $\mathbb{Z}_p[r]$ with an invertible unit in $\mathbb{Z}_p^*$ (a scalar) leaves the roots of the resulting polynomial unaltered. We define the following:

Definition 3. *Given polynomials $P(r), Q(r) \in \mathbb{Z}_p[r]$ with degree n, we say that they are* associates *(denoted as $P(r) \approx_a Q(r)$) iff $P(r)|Q(r)$ and $Q(r)|P(r)$.*

Thus, associativity can be equivalently expressed by requesting that $P(r) = \beta Q(r)$ for some $\beta \in \mathbb{Z}_p^*$.

Note that although polynomial-based set representation provides a way to verify the correctness of set operations by employing corresponding properties of the characteristic polynomials, it does not provide any computational speedup for this verification process. Intuitively, verifying operations over sets of cardinality n, involves dealing with polynomials of degree n with associated cost that is proportional to performing operations directly over the sets themselves. We overcome this obstacle, by applying our ECRH construction (which can be naturally defined over univariate polynomials with coefficients in $\mathbb{Z}_p$, as already discussed) to the characteristic polynomial $\mathcal{C}_X$: Set X will be succinctly represented by hash value $h(\mathcal{C}_X) = \left(g^{\mathcal{C}_X(s)}, g^{a\mathcal{C}_X(s)}\right)$ (parameter q is an upper bound on the cardinality of sets that can be hashed), and an operation of sets X and Y will be optimally verified by computing only on hash values $h(\mathcal{C}_X)$ and $h(\mathcal{C}_Y)$.

A note on extractability. In the above, we are essentially using a pre-processing step representing sets as polynomials, before applying the extractable hash function on the polynomial representations. We cannot define the ECRH directly for sets since, while every set has a uniquely defined characteristic polynomial, not every polynomial is a characteristic polynomial of some set. Hence extractability of sets (using only public key information) is not guaranteed. For example, an adversary can compute an irreducible polynomial $Y \in \mathbb{Z}_p[r]$, of degree > 1, and output $h(Y)$. Since Y has no roots, no extractor (without access to the secret key) can output a set for which Y is the characteristic polynomial (it can, however, extract polynomial Y with overwhelming probability). In fact, defined directly over sets with elements from $\mathbb{Z}_p$, the function ensemble $\{\mathcal{H}_l\}_l$ with an internal computation of the characteristic polynomial, can be shown to be extractable collision-resistant under the $\mathcal{ECRH}_2$ definition recently introduced in [DFH12]. In the context of a cryptographic protocol for sets, additional mechanisms need to be deployed in order to guarantee that a given hash value corresponds to the characteristic polynomial of some set. For our ADS construction, we will combine the use of the ECRH construction for sets, with an authentication mechanism deployed by the source in a pre-processing phase. This will allow any client to verify the authenticity and freshness of the hash values corresponding to sets that are input to its query.

3.4 An ADS for hierarchical set operations

Here we present an ADS supporting hierarchical set operations. We assume a data structure D consisting of m sorted sets $S_1, ..., S_m$, consisting of elements from $\mathbb{Z}_p$,[5] where sets can change under element insertions and deletions; here, p is a λ-bit prime number and λ is a security parameter. If $M = \sum_{i=1}^{m} |S_i|$, then the total space complexity needed to store D is $O(m+M)$. The supported class of queries is any set operation formula over a subset of the sets S_i, consisting of unions and intersections.

The basic idea is to use the ECRH construction from Section 3.2 to represent sets S_i by the hash values $h(C_{S_i})$ of their characteristic polynomials. For the rest of the paper, we will refer to value $h(C_{S_i})$ as h_i, implying the hash value of the characteristic polynomial of the i-th set of D or the i-th set involved in a query, when it is obvious in the context. Recall that a hash value h consists of two group elements, $h = (f, f')$. We will refer to the first element of h_i as f_i, i.e., for a set $S_i = (x_1, ..., x_n)$, $f_i = g^{\prod_{j=1}^{n}(x_j+s)}$ and likewise for f'_i. For the authenticity of these values, an authentication mechanism similar to Merkle trees (but allowing more efficient updates) will be deployed by the source.

Each answer provided by the server is accompanied by a proof that includes a number of hash values for all sets computed during answer computation, the exact structure of which depends on the type of operations. The verification process is

[5]Actually elements must come from $\mathbb{Z} \setminus \{s, 1, ..., m\}$, because s is the secret key in our construction and the m smallest integers modulo p will be used for numbering the sets.

essentially split into two parts. First, the client verifies the validity of the hash values of the sets used as input by the answer computation process (i.e., the validity of sets specified in q) and subsequently that the hash values included in the proof respect the relations corresponding to the operations in q, all the way from the input hash values to the hash value of the returned answer α. The key technique is that by using our ECRH construction we can map relations between input and output sets in a set operation, to similar relations in their hash values. This allows the verification process to run in time independent of the cardinality of involved sets and only linear to the length of q and α making it asymptotically as fast as simply reading the input and output. In the following sections, we present the algorithms of our construction.

3.4.1 Setup and updates

During the setup phase, the source computes the m hash values $h(C_{S_i})$ of sets S_i and then deploys an authentication mechanism over them, that will provide proofs of integrity for these values under some public digest that corresponds to the current state of D. This mechanism should be able to provide proofs for statements of the form "h_i is hash of the i-th set of the current version of D."

There is a wide variety of such mechanisms that can be deployed by the owner of D and the choice must be made with optimization of a number of parameters in mind, including digest size, proof size and verification time, setup and update cost and storage size. For example, using a standard collision resistant hash function, the owner can compute the hash of the string $h_1||...||h_m$ as a single hash value. However, a single update in D will require $O(m)$ work in order to compute the updated digest from scratch. On the other hand, the owner can use a digital signature scheme to sign a hash representation of each set. This yields an update cost of $O(1)$ (a single signature computation) but the digest consists of m signatures.

Another popular authentication mechanism for proofs of membership are Merkle hash trees [Mer89] that provide logarithmic size proofs, constant time updates, and a single value digest. Such a mechanism, allows the server to provide proofs that a value h_i belongs in the set of hash values of the sets in D. An alternative to Merkle trees, introduced in [PTT08] (and specifically in the bilinear group setting in [PTT11]) are *accumulation trees*. The difference between them is that their security is based on different cryptographic assumptions (secure hashing versus bilinear group assumptions) and, arguably more importantly, accumulation trees yield constant size proofs (independently of the number of elements in the tree) and constant time updates. Another useful property of the accumulation tree is that it can be computed using the same ECRH construction we will be using for the rest of the algorithms of our scheme. Thus, we can avoid the cost of additional public/secret key generation and maintenance. In our construction, we use the accumulation tree to verify the correctness of hash values for the sets involved in a particular query. On a high level, the public tree digest guarantees the integrity of the hash values and in turn the hash values validate the elements of the sets.

An accumulation tree AT is a tree with $\lceil 1/\epsilon \rceil$ levels, where $0 < \epsilon < 1$ is a parameter chosen upon setup, and m leaves. Each internal node of T has degree $O(m^\epsilon)$ and T has constant height for a fixed ϵ. Intuitively, it can be seen as a "flat" version of Merkle trees. Each leaf node contains the (first half of the) hash value of a set S_i and each internal node contains the (first half of the) hash of the values of its children. Since, under our ECRH construction, hash values are elements in $\mathbb{G}$ we will need to map these bilinear group elements to values in $\mathbb{Z}_p^*$ at each level of the tree before they can be used as inputs for the computation of hash values of higher level nodes. This can be achieved by a function ϕ that outputs a bit level description of hash values under some canonical representation of $\mathbb{G}$ (see below). The accumulation tree primitive we are using here has been used in [PTT08, PTT11, PTT15] where the corresponding "hashing" function used was the bilinear accumulator construction from [Ngu05]. We are implicitly making use of the fact that the outputs of our ECRH construction can be interpreted as pairs of accumulation values of sets.

Now we present the setup and update algorithms or our ADS construction:

Algorithm $\{sk, pk\} \leftarrow$**genkey**(1^l). The owner of D runs the sampling algorithm for our ECRH construction, chooses an injective[6] function $\phi : \mathbb{G} \setminus \{1_{\mathbb{G}}\} \rightarrow \mathbb{Z}_p^*$, and outputs $\{\phi, pk, sk\}$.

Algorithm $\{auth(D_0), d_0\} \leftarrow$ **setup**(D_0, sk, pk). The owner of D computes values $f_i = g^{\prod_{x \in S_i}(x_i+s)}$ for sets S_i. Following that, he constructs an accumulation tree AT over values f_i. A parameter $0 < \epsilon < 1$ is chosen. For each node v of the tree, its value $d(v)$ is computed as follows. If v is a leaf corresponding to f_i then $d(v) = f_i^{(i+s)}$ where the number i is used to denote that this is the i-th set in D (recall that, by definition, sets S_i contain elements in $[m+1, ..., p-1]$). Otherwise, if $N(v)$ is the set of children of v, then $d(v) = g^{\prod_{u \in N(v)}(\phi(d(u)+s)}$ (note that the exponent is the characteristic polynomial of the set containing the elements $\phi(d(u))$ for all $u \in N(v)$). Finally, the owner outputs $\{auth(D_0) = f_1, ..., f_t, d(v)\ \forall v \in AT\ , d_0 = d(r)\}$ where r is the root of AT.

Algorithm$\{auth(D_{h+1}), d_{h+1}, upd\} \leftarrow$ **update**$(u, auth(D_h), d_h, sk, pk)$. For the case of insertion of element x in the i-th set, the owner computes $x + s$ and $\eta = f_i^{x+s}$. For deletion of element x from S_i, the owner computes $(x+s)^{-1}$ and $\eta = f_i^{(x+s)^{-1}}$. Let v_0 be the leaf of AT that corresponds to the i -th set and $v_1, ..., v_{\lceil 1/\epsilon \rceil}$ the node path from v_0 to r. Then, the owner sets $d'(v_0) = \eta$ and for $j = 1, ..., \lceil 1/\epsilon \rceil$ he sets $d'(v_j) = d(v_j)^{(\phi(d'(v_{j-1}))+s)(\phi(d(v_{j-1}))+s)^{-1}}$. He replaces node values in $auth(D_h)$ with the corresponding computed ones to produce $auth(D_{h+1})$. He

[6]The restriction that ϕ is injective is in fact too strong. In practice, it suffices that it is collision-resistant. A good candidate for ϕ is a function that uses a CRHF to hash the bit-level description of an element of $\mathbb{G}$ to $\mathbb{Z}_p^*$.

then sets $upd = d(v_0), ..., d(r), x, i, b$ where b is a bit denoting the type of operation and sends upd to server. Finally, he publishes updated digest $d_{h+1} = d'(r)$.

Algorithm $\{D_{h+1}, auth(D_{h+1}), d_{h+1}\} \leftarrow$ **refresh**$(u, D_h, auth(D_h), d_h, upd, pk)$. The server replaces values in $auth(D_h)$ with the corresponding ones in upd, d_h with d_{h+1} and updates set S_i accordingly.

The runtime of setup is $O(m + M)$ as computation of the hash values using the secret key takes $O(M)$ and the tree construction has access complexity $O(m)$ for post-order traversal of the tree as it has constant height and it has m leaves. Similarly, update and refresh have access complexity of $O(1)$.

Remark 1. *Observe that the only algorithms that make use of the trapdoor s are **update** and **setup** when computing hash values. Note though, that both algorithms can be efficiently executed without s (given only the public key).*

3.4.2 Query responding and verification

As mentioned before, we wish to achieve two verification properties: *integrity-of-data* and *integrity-of-computation*. We begin with our algorithms for achieving the first property, and then present two protocols for achieving the second one, i.e., for validating the correctness of a single set operation (union or intersection). These algorithms will be used as subroutines by our final query responding and verification processes.

Authenticity of hash values

We present two algorithms that make use of the accumulation tree deployed over the hash values of S_i in order to prove and verify that the sets used for answering are the ones specified by the query description.

Algorithm $\pi \leftarrow$ *QueryTree*$(pk, d, i, auth(D))$ The algorithm computes proof of membership for value x_i validating that it is the i-th leaf of the accumulation tree.

Let v_0 be the i-th node of the tree an $v_1, ..., v_{\lceil 1/\epsilon \rceil}$ be the node path from v_0 to the root r. For $j = 1, ..., \lceil 1/\epsilon \rceil$ let $\gamma_j = g^{\prod_{u \in N(v_j) \setminus \{v_{j-1}\}} (\phi(d(u))+s)}$ (note that the exponent is the characteristic polynomial of the set containing the elements $\phi(d(u))$ for all $u \in N(v)$ except for node v_{j-1}). The algorithm outputs $\pi := (d(v_0), \gamma_1), ..., (d(v_{\lceil 1/\epsilon \rceil - 1}), \gamma_{\lceil 1/\epsilon \rceil})$.

Algorithm $\{0,1\} \leftarrow$ *VerifyTree*(pk, d, i, x, π). The algorithm verifies membership of x as the i-th leaf of the tree by checking the equalities: (i) $e(d(v_1), g) = e(x, g^i g^s)$; (ii) for $j = 1, ..., \lceil 1/\epsilon \rceil - 1$, $e(d(v_j), g) = e(\gamma_j, g^{\phi(d(v_{j-1}))} g^s)$; (iii) $e(d, g) = e(\gamma_{\lceil 1/\epsilon \rceil}, g^{\phi(d(v_{\lceil 1/\epsilon \rceil - 1}))} g^s)$. If none of them fails, it output accept.

The above algorithms make use of the property that for any two polynomials $A(r), B(r)$ with $C(r) := A(r) \cdot B(r)$, for our ECRH construction it must be that $e(f(C), g) = e(f(A), f(B))$. In particular for sets, this allows the construction of a single-element proof for set membership (or subset more generally). For example, for element $x_1 \in X = \{x_1, ..., x_n)$ this witness is the value $g^{\prod_{i=2}^{n}(x_i+s)}$. Intuitively, for the integrity of a hash value, the proof consists of such set membership proofs starting from the desired hash value all the way to the root of the tree, using the sets of children of each node. The following lemma (from [PTT11]; slightly informal here) states the security of an accumulation tree:

Lemma 4 ([PTT11])**.** *Under the q-**SBDH** assumption, for any adversarially chosen proof π s.t. $\{j, x^*, \pi)$ s.t. VerifyTree$(pk, d, j, x^*, \pi) \rightarrow 1$, it must be that x^* is the j-th element of the tree except for negligible probability. Algorithm QueryTree has access complexity $O(m^\epsilon \log m)$ and outputs a proof of $O(1)$ group elements and algorithm VerifyTree has access complexity $O(1)$.*

Algorithms for the single operation case

The algorithms presented here are used to verify that a set operation was performed correctly, by checking a number of relations between the hash values of the input and output hash values, that are related to the type of set operation. The authenticity of these hash values is not necessarily established. Since these algorithms will be called as sub-routines by the general proof construction and verification algorithms, this property should be handled at that level.

Intersection. Let $I = S_1 \cap \ldots \cap S_t$ be the wanted operation. Set I is uniquely identified by the following two properties: **(Subset)** $I \subseteq S_i$ for all S_i and **(Complement Disjointness)** $\cap_{i=1}^{t}(S_i \setminus I) = \emptyset$. The first captures that all elements of I appear in all of S_i and the second that no elements are left out.

Regarding the subset property, we argue as follows. Let X, S be sets s.t. $S \subseteq X$ and $|X| = n$. Observe that $\mathcal{C}_S | \mathcal{C}_X$, i.e. $\mathcal{C}_X$ can be written as $\mathcal{C}_X = \mathcal{C}_S(r)Q(r)$ where $Q(r) \in \mathbb{Z}_p[r]$ is $\mathcal{C}_{X \setminus S}$. The above can be verified by checking the equality:

$$e(f_S, W) = e(f_X, g) \ ,$$

where $W = g^{Q(s)}$. If we denote with W_i the values $g^{\mathcal{C}_{S_i \setminus I}(s)}$, the subset property can be verified by checking the above relation for I w.r.t each of S_i.

For the second property, we make use of the property that $\mathcal{C}_{S_i \setminus I}(r)$ are disjoint for $i = 1, \ldots, t$ if and only if there exist polynomials $q_i(r)$ s.t. $\sum_{i=1}^{t} \mathcal{C}_{S_i \setminus I}(r) q_i(r) = 1$, i.e. the *gcd* of the characteristic polynomials of the the complements of I w.r.t S_i should be 1. Based on the above, we propose the algorithms in Figure 3·1 for the case of a single intersection.

Union. Now we want to provide a similar method for proving the validity of a union operation of some sets. Again we denote set $U = S_1 \cup \ldots \cup S_t$ and let h_i be the corresponding hash values as above. The union set U is uniquely characterized by the following two properties: **(Superset)** $S_i \subseteq U$ for all S_i and **(Membership)** For each element $x_i \in U$, $\exists j \in [t]$ s.t. $x_i \in S_j$. These properties can be verified, with values W_i, w_j for $i = 1, \ldots t$ and $j = 1, \ldots, |U|$ defined as above checking the following

```
**Algorithm**$\{\Pi, f_I\} \leftarrow$ *proveIntersection*$(S_1, ..., S_t, I, h_1, ..., h_t, h_I, pk)$.
    1. Compute values $W_i = g^{\mathcal{C}_{S_i \setminus I}(s)}$.
    2. Compute polynomials $q_i(r)$ s.t. $\sum_{i=1}^{t} \mathcal{C}_{S_i \setminus I}(r) q_i(r) = 1$ and values $F_i = g^{q_i(s)}$.
    3. Let $\Pi = \{(W_1, F_1), ..., (W_t, F_t)\}$ and output $\{\Pi, f_I\}$.
**Algorithm**$\{$accept,reject$\} \leftarrow$ *verifyIntersection*$(f_1, ..., f_t, \Pi, f_I, pk)$.
    1. Check the following equalities. If any of them fails output reject, otherwise accept:
        • $e(f_I, W_i) = e(f_i, g) \qquad \forall i = 1, ..., t$
        • $\prod_{i=1}^{t} e(W_i, F_i) = e(g, g)$
```

Figure 3·1: Algorithms for proving and verifying a single intersection

equalities (assuming h_U is the hash value of U):

$$e(f_i, W_i) = e(f_U, g) \qquad \forall i = 1, ..., t$$
$$e(g^{x_j} g^s, w_j) = e(f_U, g) \qquad \forall j = 1, ..., |U| \ .$$

The problem with this approach is that the number of equalities to be checked for the union case is linear to the number of elements in the output set. Such an approach would lead to an inefficient scheme for general operations (each intermediate union operation the verification procedure would be at least as costly as computing that intermediate result). Therefore, we are interested in restricting the number of necessary checks. In the following we provide a union argument that achieves this.

Our approach stems from the fundamental inclusion-exclusion principle of set theory. Namely for set $U = A \cup B$ it holds that $U = (A + B) \setminus (A \cap B)$ where $A + B$ is a simple concatenation of elements from sets A, B (allowing for multisets), or equivalently, $A + B = U \cup (A \cap B)$. Given the hash values h_A, h_B the above can be checked by the bilinear equality $e(f_A, f_B) = e(f_U, f_{A \cap B})$. Thus one can verify the correctness of h_U by checking a number of equalities independent of the size of U by checking that the above equality holds. In practice, our protocol for the union of two sets consists of a proof for their intersection, followed by a check for this relation. Due to the extractability property of our ECRH, the fact that h_I is included in the proof acts as a proof-of-knowledge by the prover for the set I, hence we can remove the necessity to explicitly include I in the answer.

There is another issue to be dealt with. namely that this approach does not scale well with the number of input sets for the union operation. To this end, we will recursively apply our construction for two sets in pairs of sets until finally we have a single union output. Let us describe the semantics of a set union operation over t sets. For the rest of the section, without loss of generality, we assume $\exists k \in \mathbb{N}$ s.t. $2^k = t$, i.e., t is a power of 2. Let us define as $U_1^{(1)}, ..., U_{t/2}^{(1)}$ the sets $(S_1 \cup S_2), ..., (S_{t-1} \cup S_t)$. For set U is holds that $U = U_1 \cup ... \cup U_{t/2}$ due to the commutativity of the union operation.

All intermediate results $U_i^{(j)}$ will be represented by their hash values $h_{U_i^{(j)}}$ yielding a proof that is of size independent of their cardinality. One can use the intuition explained above, based on the inclusion-exclusion principle, in order to prove the correctness of (candidate) hash values $h_{U_i^{(1)}}$ corresponding to sets U_i and, following that, apply repeatedly pairwise union operations and provide corresponding arguments, until set U is reached. Semantically this corresponds to a binary tree $\mathcal{T}$ of height k with the original sets S_i at the t leafs (level 0), sets $U_i^{(1)}$ as defined above at level 1, and so on, with set U at the root at level k. Each internal node of the tree corresponds to a set resulting from the union operation over the sets of its two children nodes. In general we denote by $U_1^{(j)}, ..., U_{t/2^j}^{(j)}$ the sets appearing at level j.

We propose the algorithms in Figure 3·2 for proof construction and verification for a single union.

Set difference. Finally, we present a protocol for the set difference operation. We denote set $X = S_1 \setminus S_2$ and let h_1, h_2, h_X be the corresponding hash values. For

For ease of notation we denote by A, B the two sets corresponding to the children nodes of each non-leaf node of $\mathcal{T}$, by U, I their union and intersection respectively and by F the final union output.

Algorithm$\{\Pi, f_F\} \leftarrow$*proveUnion*$(S_1, ..., S_t, U, h_1, ..., h_t, h_U, pk)$.

```
1. Initialize $\Pi = \emptyset$.
2. For each $U_i^{(j)}$ of level $j = 1, ..., k$, corresponding to sets $U, I$ as defined above, compute $U, I$ and values $h_U, h_I$. Append values $h_U, h_I$ to $\Pi$.
3. For each $U_i^{(j)}$ of level $j = 1, ..., k$, run algorithm proveIntersection$(A, B, h_A, h_B, pk)$ to receive $(\Pi_I, f_I)$ and append $\Pi_I$ to $\Pi$. Observe that sets $A, B$ and their hash values have been computed in the previous step.
4. Output $\{\Pi, f_F\}$. ($h_F$ has already been computed at step (2) but is provided explicitly for ease of notation).
```

Algorithm$\{$accept,reject$\} \leftarrow$ *verifyUnion*$(f_1, ..., f_t, \Pi, f_F, pk)$.

```
1. For each intersection argument $\{\Pi_I, f_I\} \in \Pi$ run verifyIntersection$(f_A, f_B, \Pi_I, f_I, pk)$. If for any them it outputs reject, output reject.
2. For each node of $\mathcal{T}$ check the equality $e(f_A, f_B) = e(f_U, f_I)$. If any check fails, output reject.
3. For each hash value $h_U \in \Pi$ check $e(f_U, g^a) = e(f'_U, g)$ and likewise for values $h_I$. If any check fails output reject, otherwise accept.
```

Figure 3·2: Algorithms for proving and verifying a single union

characterizing the set difference X we use the following two properties: $X \subseteq S_1$ and $X + (S_1 \cap S_2) = S_1$. The former states that X is a subset of S_1 whereas the latter states that the X is the complement of the intersection of the two sets, with respect to S_1.

To prove these properties we take a similar approach as for the case of set union above. Namely, we first provide a proof for the hash value h_I of the intersection I of the two sets. Given h_I, the authenticated h_1 and the claimed set difference h_X, the client can verify the correctness of the result by a checking a single pairing equation, in a very similar manner as for the case of a union of two sets, discussed above. We propose the algorithms in Figure 3·3 for proving and verifying a single set difference.

Complexity analysis of the algorithms. Let $N = \sum_{i=1}^{t} |S_i|$ and $\delta = |I|$ or $|F|$

```
Algorithm{$\Pi, f_D$} ← proveDifference($S_1, S_2, D, h_1, h_2, h_D, pk$).
    1. Run algorithm proveDifference($S_1, S_2, h_1, h_2, pk$) to receive ($\Pi_I, f_I$). Append $\Pi_I$ to $\Pi$.
    2. Compute $h_I$ and append it to $\Pi$.
    3. Output ($\Pi, f_D$).
Algorithm{accept,reject} ← verifyDifference($f_1, f_2, \Pi, f_D, pk$).
    1. Parse $\Pi = \Pi_I, h_I$.
    2. Run algorithm verifyIntersection($f_1, f_2, \Pi_I, f_I, pk$). If it outputs reject, output reject.
    3. Check the following equalities. If any of them fails output reject, otherwise accept:
        • $e(f_I, g^\alpha) = e(f'_I, g)$
        • $e(f_D, f_I) = e(f_1, g)$
```

Figure 3·3: Algorithms for proving and verifying a single set difference

respectively, depending on the type of operations. For both cases, the runtimes of the algorithms are $O(N \log^2 N \log\log N \log t)$ for proof construction and $O(t + \delta)$ for verification and the proofs contain $O(t)$ bilinear group elements. A proof of the complexity analysis for these algorithms can be found in Section 3.4.4.

It can be shown that these algorithms, along with appropriately selected proofs-of-validity for their input hash values can be used to form a complete ADS scheme for the case of a single set operation. Here however, these algorithms will be executed as subroutines of the general proof construction and verification process for our ADS construction for more general queries, presented in the next section.

Hierarchical sets operation queries

We now use the algorithms we presented in the previous subsection to define appropriate algorithms **query**, **verify** for our ADS scheme. A hierarchical set operations computation can be abstracted as a tree, the nodes of which contain sets of elements. For a query q over t sets $S_1, ..., S_t$, corresponding to such a computation, each leaf of the tree $\mathcal{T}$ contains an input set for q and each internal node is related to a set operation (union or intersection) and contains the set that results to applying this set operation on its children nodes. Finally, the root of the tree contains the output set of q. In order to maintain the semantics of a tree, we assume that each input is treated as a distinct set, i.e., t is not the number of different sets that appear in q, but the total number of involved sets counting multiples. An alternative way to visualize this would be to interpret t as the length of the set operations formula corresponding to q.[7]

Without loss of generality, assume q is defined over the t first sets of D. For reasons of simplicity we describe the mode of operation of our algorithms for the case where all sets S_i are at the same level of the computation, i.e., all leafs of $\mathcal{T}$ are at the same level. The necessary modifications in order to explicitly cover the case where original sets occur higher in $\mathcal{T}$, are implied in a straight forward manner from the following analysis, since any set S_i encountered at an advanced stage of the process can be treated in the exact same manner as the sets residing at the tree leafs. The algorithms for query processing and verification of our ADS scheme are described in Figures 3·4 and 3·5.

Intuitively, with the algorithms from the previous section a verifier can, by checking a small number of bilinear equations, gain trust on the hash value of a set computed by a single set operation. Observe that, each prover's algorithm "binds" some bilinear group elements (the first parts of the input hash values) to a single bilinear group element (the first part of the hash value of the output set). We made explicit use of that, in order to create a proof of union for more than two sets in the previous

[7]More generally q can be seen as a DAG. Here, for simplicity of presentation we assume that all sets S_i participate only once in q hence it corresponds to a tree. This is not a limitation of our model but to simplify the execution of the algorithms, every set encountered is treated uniquely. This can incur a redundant overhead in the analysis, that is avoidable in practice (e.g., by not including duplicate values in proofs).

D is the most recent version of the data structure and $auth(D), d$ be the corresponding authenticated values and public digest. Let q be a set operation formula with nested unions, intersections, and differences and $\mathcal{T}$ be the corresponding semantics tree. For each internal node $v \in \mathcal{T}$ let $R_1, ..., R_{t_v}$ denote the sets corresponding to its children nodes ($t_v = 2$ for the set difference case)and O be the set that is produced by executing the operation in v (union, intersection, or difference) over R_i. Finally, denote by $\alpha = x_1, ..., x_\delta$ the output set of the root of $\mathcal{T}$.

Algorithm $\{\alpha, \Pi\} \leftarrow$ **query**$(q, D, auth(D), pk)$.

```
1. Initialize $\Pi = \emptyset$.
2. Compute proof-of-membership $\pi_i$ for value $f_i$ by running $QueryTree(pk, d, i, auth(D))$
   for $i \in [t]$ and append $\pi_i, f_i$ to $\Pi$.
3. For each internal node $v \in \mathcal{T}$ (as parsed with a DFS traversal):
    • Compute set $O$ and its hash value $h_O = h(\mathcal{C}_O)$.
    • If $v$ corresponds to a set intersection, obtain $\Pi_v$ by running
      $proveIntersection(R_1, ..., R_t, h_1, ..., h_t, O, h_O, pk)$. For each subset witness $W_i \in \Pi$
      corresponding to polynomial $\mathcal{C}_{R_i \setminus O}$, compute values $\tilde{W}_i = g^{a\mathcal{C}_{R_i \setminus O}(s)}$.
      Let $\mathcal{W}_v = \{W_1, ..., W_{t_v}\}$. Append $\Pi_v, \mathcal{W}_v, h_O$ to $\Pi$.
    • If $v$ corresponds to a set union, obtain $\Pi_v$ by running
      $proveUnion(R_1, ..., R_t, h_1, ..., h_t, O, h_O, pk)$. Append $\Pi_v, h_O$ to $\Pi$.
    • If $v$ corresponds to a set difference, obtain $\Pi_v$ by running
      $proveDifference(R_1, R_2, h_1, h_2, O, h_O, pk)$. Append $\Pi_v, h_O$ to $\Pi$.
4. Append to $\Pi$ the coefficients $(c_0, ..., c_\delta)$ of the polynomial $\mathcal{C}_\alpha$ (already computed at step
   3 and output $\{\alpha, \Pi)\}$.
```

Figure 3·4: Algorithm for proving general set operations

section. Here we generalize it, to be able to obtain similar proofs for hierarchical queries containing intersections and unions. The proof for q is constructed by putting together smaller proofs for all the internal nodes in $\mathcal{T}$. Let Π be a concatenation of single union and single intersection proofs that respect q, i.e., each node in $\mathcal{T}$ corresponds to an appropriate type of proof in Π. The hash value of each intermediate result will also be included in the proof and these values at level i will serve as inputs for the verification process at level $i+1$. The reason the above strategy will yield a secure scheme is that the presence of the hash values serves as a proof by a cheating adversary that he has "knowledge" of the sets corresponding to these partial results.

Let d be the latest public digest. Let q be a set operation formula with nested unions, intersections, and differences and $\mathcal{T}$ be the corresponding semantics tree. For each internal node $v \in \mathcal{T}$ let $R_1, ..., R_{t_v}$ denote the sets corresponding to its children nodes ($t_v = 2$ for the set difference case), and let $\alpha = x_1, ..., x_\delta$ denote the claimed answer set for q. Finally, for internal node $v \in \mathcal{T}$, let $\eta_1, ..., \eta_{t_v}$ denote the hash values of its children node sets (η_1, η_2 for the set difference case) $\in \Pi$ (for internal nodes at level 1, the values η_i are the values f_i).

Algorithm {accept,reject} $\leftarrow$ **verify**(q, α, Π, d, pk).

1. Verify the validity of values f_i. For each value $f_i \in \Pi$ run *VerifyTree*(pk, d, i, f_i, π_i). If it outputs reject for any of them, output reject and halt.
2. For each internal node $v \in \mathcal{T}$ (as parsed with a DFS traversal):
    - Check the equality $e(f_O, g^a) = e(g, f'_O)$. If it does not hold, reject and halt.
    - If v corresponds to a set intersection:
        (a) Run *verifyIntersection*$(\eta_1, ..., \eta_{t_v}, \Pi_v, f_O, pk)$, If it outputs reject, output reject and halt.
        (b) For each pair $W_i, \tilde{W}_i \in \Pi_v$, check the equality $e(W_i, g^a) = e(\tilde{W}_i, g)$. If any of the checks fails, output reject and halt.
    - If v corresponds to a set union, run *verifyUnion*$(\eta_1, ..., \eta_{t_v}, \Pi_v, f_O, pk)$. If it outputs reject, output reject and halt.
    - If v corresponds to a set difference, run *verifyDifference*$(\eta_1, \eta_2, \Pi_v, f_O, pk)$. If it outputs reject, output reject and halt.
3. Validate the correctness of coefficients $\mathbf{c}$. Choose $z \leftarrow_R \mathbb{Z}_p^*$ and compare the values $\sum_{i=0}^{\delta} c_i z^i$ and $\prod_{i=1}^{\delta}(x_i + z)$. If they are not equivalent, output reject and halt.
4. Check the equality $e(\prod_{i=0}^{\delta} g^{c_i s^i}, g) = e(f_\alpha, g)$. If it holds output accept, otherwise reject.

Figure 3·5: Algorithm for verifying general set operations

If one of these sets is not honestly computed, the extractability property allows an adversary to either attack the collision-resistance of the ECRH or break the q-**SBDH** assumption directly, depending on the format of the polynomial used to cheat.

Observe that the size of the proof Π is $O(t + \delta)$. This follows from the fact that the t proofs π_i consist of a constant number of group elements and Π is of size $O(t)$ since each of the $O(|T|) = O(t)$ nodes participates in a single operation. Also, there are δ coefficients b_i therefore the total size of Π is $O(t + \delta)$. The runtime of the verification algorithm is $O(t+\delta)$ as steps 2,3 takes $O(t)$ operations and steps 4,5 take $O(\delta)$. A detailed proof of the complexity analysis for these algorithms can be found in Section 3.4.4.

3.4.3 Main result

We can now state the following theorem that is our main result.

Theorem 1. *The scheme* $\mathcal{AHSO} = \{$**genkey**, **setup**, **query**, **verify**, **update**, **refresh**$\}$ *is an ADS for queries* q *from the class of hierarchical set operations formulas involving unions, intersections and set differences, of length polynomial in* λ*. It is* correct *and* secure *under the* q-***SBDH*** *and the* q-***PKE*** *assumptions.*

Proof: Correctness follows by close inspection of the algorithms above. We proceed to prove the security of our scheme via a series of intermediate results. Let $\mathcal{A}_{ADS}$ be an adversary for the $\mathcal{AHSO}$ scheme. Recall the $\mathcal{A}_{ADS}$ is given a public key generated by **genkey** containing a description of an ECRH h. $\mathcal{A}_{ADS}$ then calls **setup** with the parameter D_0 and subsequently makes additional oracle calls to the algorithms **update**, **refresh**, **verify**, **proof**, of $\mathcal{AHSO}$. Finally, $\mathcal{A}_{ADS}$ outputs $\{\alpha, \Pi, q, D_k, auth(D_k), d_k\}$ where Π is a proof that contains images of the hash h. We show that there exists an extractor $\mathcal{E}$ that except with negligible probability over the choice of h, when $\mathcal{E}$ is given h, outputs a pre-image for every valid image of h in Π. We cannot directly use the extraction property of the ECRH since the adversary $\mathcal{A}_{ADS}$ is getting access to oracles for the algorithms of $\mathcal{AHSO}$ and we do not have access to the code of these oracles. The idea of this proof is to use the fact that all the algorithms of $\mathcal{AHSO}$ (except **genkey**) can be executed over the initial database D_0 in polynomial time given only the public key h (see Remark 1), and therefore there exists an adversary $\mathcal{A}'_{ADS}$ that internally emulates $\mathcal{A}'_{ADS}$ together with its oracles and outputs the same as $\mathcal{A}_{ADS}$. Let $\mathcal{A}'_i$ be the adversary that emulates $\mathcal{A}'_{ADS}$ and outputs the i'th hash value h_i in proof Π contained in the output of $\mathcal{A}'_{ADS}$. It follows from the properties of the ECRH that there exists an extractor $\mathcal{E}_i$ for $\mathcal{A}'_i$ that outputs a pre-image of h_i whenever h_i is indeed in the image of h. Therefore there exists a single extractor $\mathcal{E}$ that outputs the pre-images for all valid h_i's with overwhelming probability. Finally, observe that hash values h_i are efficiently recognizable as pairs of elements of $\mathbb{G}$, and can be efficiently checked for validity by checking the equation in $\mathbb{G}_T$, $e(f, g^\alpha) = e(f', g)$.

As a building block for our proof, we prove the following lemma:

Lemma 5. *If the q-**SBDH** assumption holds, then for any poly-size adversary $\mathcal{A}$ that upon input pk outputs $(S_1, ..., S_t, O, \Pi, f_O)$ s.t. (i)* $\text{verifyIntersection}(f(\mathcal{C}_{S_1}), ..., f(\mathcal{C}_{S_t}), \Pi, f_I, pk)$ *(resp.* verifyUnion,VerifyDifference*) accepts and (ii) $f(\mathcal{C}_O) = f_O$, $O = \cap_{i=1}^t S_i$ (resp. $O = \cup_{i=1}^t S_i$, it must be that $O = S_1 \setminus S_2$) with all but negligible probability.*

Proof: We examine the three cases separately.

Intersection. Let us assume that there exists $\mathcal{A}$ that outputs $S_1, ..., S_t, O, \Pi, f_O$ s.t. *verifyIntersection* accepts and $O \neq I := \cap_{i=1}^t S_i$, with non-negligible probability. We will construct an adversary $\mathcal{A}'$ that breaks the q-**SBDH** assumption. For ease of notation we denote $\mathcal{C}_{S_i} = Q_i(r)$ and $\mathcal{C}_O = P(r)$.

Since $O \neq I$, either it contains an element x s.t. $x \notin I$, or there exists element $x \in I$ s.t. $x \notin O$ (or both happen at the same time). Let us deal with the first case. Since $x \in O \wedge x \notin I$, there must exist set S_j s.t. $x \notin S_j$. Therefore for the term $(x + r)$ it is true that $(x + r) \not| Q_j(r)$ and $(x + r) | P(r)$. It follows that there exist efficiently computable $F(r), \kappa$ s.t. $Q_j(r) = (x + r)F(r) + \kappa$. Also let $H(r)$ be polynomial s.t. $(x+r)H(r) = P(r)$. The following equalities must hold:

$$\begin{aligned}
e(f_O, W_j) &= e(f_j, g)\\
e(g, W_j)^{P(s)} &= e(g,g)^{Q_j(s)}\\
e(g, W_j)^{(x+s)H(s)} &= e(g,g)^{(x+s)Q_j(s)+\kappa}\\
\left(e(g,W)^{H(s)}e(g,g)^{-Q_j(s)}\right)^{\kappa^{-1}} &= e(g,g)^{\frac{1}{x+s}}.
\end{aligned}$$

It follows that $\mathcal{A}'$ can, by outputting the above value break the q-**SBDH** for point x. Hence, this case can happen only with negligible probability.

It remains to deal with the second case, conditioned on the first not happening. Namely, there exists $x \in I$ that is omitted by answer O, i.e. O is a common subset of S_i but not the maximal one. There must exist $x \in I$ s.t. $x \notin O$ therefore it must be that $x \in (S_i \setminus O)$ for all $i = 1, ..., t$. Let polynomials $R_i(r) = \mathcal{C}_{S_i \setminus O}$. Observe that because the verifier accepts, it must be that $e(g, W_i) = e(g,g)^{Q_i(s)}$, hence $W_i = g^{R_i(s)}$. From the above it must hold that $R_i(r) = (x+r)R_i'(r)$ for some $R_i'(r) \in \mathbb{Z}[r]$. The following must be true:

$$\begin{aligned}
\prod_{i=1}^{t} e(W_i, F_i) &= e(g,g)\\
\left(\prod_{i=1}^{t} e(g^{R_i'(s)}, F_i)\right)^{x+s} &= e(g,g)\\
\prod_{i=1}^{t} e(g^{R_i'(s)}, F_i) &= e(g,g)^{\frac{1}{x+s}}.
\end{aligned}$$

From the above, $\mathcal{A}'$ can break the q-**SBDH** assumption for point x. It follows that O is the maximal common subset of S_i's with all but negligible probability.

If we denote the two cases as E_1, E_2, we showed that $\Pr[E_1], \Pr[E_2|E_1^c]$ are negligible probabilities. Since E_1, E_2 cover all possible cheating adversary strategies, the claim follows by a simple union bound.

Union. Let us assume that there exists $\mathcal{A}$ that outputs $S_1, ..., S_t, O, \Pi, f_O$ s.t. *verifyUnion* accepts and $O \neq U := \cap_{i=1}^{t} S_i$, with non-negligible probability. We will construct an adversary $\mathcal{A}'$ that either finds a collision in h, or breaks the q-**SBDH** assumption. For ease of notation we denote $\mathcal{C}_{S_i} = Q_i(r)$ and $\mathcal{C}_O = P(r)$. We begin by providing a proof for $t = 2$, i.e., a union of two sets $A \cup B$.

Upon receiving the output from $\mathcal{A}$, adversary $\mathcal{A}'$ runs the extractor $\mathcal{E}_{\mathcal{A}}$ (the existence of which is guaranteed by our analysis in the start of the proof of Theorem 1) for value $h_{I^*} \in \Pi$ to receive polynomial $R(r)$ s.t. $g^{R(s)} = h_{I^*}$ with overwhelming probability.

Claim 1. *$R(r) \approx_a \mathcal{C}_I$ where $I = A \cap B$, with all but negligible probability.*

Proof of Claim. The following two relations must hold:

$$e(g, W_A)^{R(s)} = e(g, g)^{Q_A(s)}$$
$$e(g, g)^{Q_A(s) \cdot Q_B(s)} = e(g, g)^{R(s) \cdot P(s)}.$$

First we will prove that $R(r)$ can be written as a product of degree 1 polynomials. Assume there exists irreducible polynomial $R'(r)$ of degree> 1 and polynomial $J(r)$ s.t. $R(r) = R'(r)J(r)$. It follows that $R(r)P(r) \neq Q_A(r)Q_B(r)$ (since only one of them has irreducible terms of degree greater than 1), however from the above equality $h(R(r)P(r)) = h(Q_A(r)Q_B(r))$ therefore by outputting $R(r) \cdot P(r), Q_A(r) \cdot Q_B(r)$ (in coefficient form), $\mathcal{A}'$ finds a collision in the ECRH. This can happen with negligible probability hence $R(r)$ can be written as a product of degree 1 polynomials with all but negligible probability.

From this it follows that $\mathcal{A}'$ can, by running a probabilistic polynomial factorization algorithm, find roots x_i s.t. $R(r) = \beta \prod_{i \in [deg(R)]}(x_i + r)$. Note that upon input polynomial $R(r)$, value β can be efficiently computed correctly by a polynomial factorization algorithm, with all but negligible probability, and the value β^{-1} is also computable efficiently since p is a prime.

Let X be the set containing the additive inverses of the roots x_i[8] and observe that $\mathcal{C}_X = \beta \prod_{i \in [deg(R)]}(x_i + r)$. If $X \neq I$, $\mathcal{A}'$ can output $\{A, B, X, Pi^* = (h_X^{\beta^{-1}}, W_A^{\beta}, W_B^{\beta}, F_A^{\beta^{-1}}, F_B^{\beta^{-1}})\}$. It is easy to verify that the above is satisfying proof for the claim that $X = A \cap B$ (i.e., *verifyIntersection* accepts), while $X \neq I$. By our previous analysis for the intersection case, this can only happen with negligible probability. This concludes the proof of the claim. $\blacksquare$

Consequently, the following must be true:

$$\begin{aligned} e(g,g)^{Q_A(s)Q_B(s)} &= e(g,g)^{R(s)P(s)} \\ e(g,g)^{\prod_{x\in A}(x+s)\prod_{x\in B}(x+s)} &= e(g,g)^{P(s)\beta\prod_{x\in A\cap B}(x+s)} \\ e(g,g)^{\prod_{x\in A\cup B}(x+s)} &= e(g,g)^{\beta P(s)}. \end{aligned}$$

In case polynomials $\mathcal{C}_{A\cup B}$ and $\beta P(r)$ are not equivalent, due to the above equality $\mathcal{A}$ can by outputting them find a collision in the ECRH. Therefore it must be that with overwhelming probability $\beta P(r) = \mathcal{C}_U$. Again, if $\beta \neq 1$ then the two polynomials form a collision for the ECRH, therefore with all but negligible probability, $O = U$.

Let us now turn our attention to the case of a generalized union over k sets (assume wlog that k is a power of 2). Consider the binary tree $\mathcal{T}$ that captures this union operation as described in Section 3.4.2. Observe that this tree consists only of $O(poly(\lambda))$ nodes ($2t-1$ in practice) hence $\mathcal{A}'$ can efficiently run an extractor for all intermediate hash values corresponding to internal nodes of $\mathcal{T}$ (as per our former analysis) to compute the related polynomials correctly, with overwhelming probability.

[8]The case where X has a root that is also a root of $\mathcal{I}$ but with cardinality > 1 can easily be dealt with as follows. Since the term $(x+s)$ appears in the exponent in both sides of the bilinear relation, $\mathcal{A}'$ can remove it from both hands, until at the end it remains only in one of them. After that happens, the consequent analysis holds. Similar argument can be made for the union case thus in the following we skip this part of the analysis.

We will prove that the polynomial $\mathcal{C}_O(r)$, corresponding to h_O, is an associate of $\mathcal{C}_U$ by showing that this is true for all intermediate polynomials and their corresponding sets. We will do this by an induction on the levels of $\mathcal{T}$.

level-1 Let $P_i^{(1)}(r)$ be the extracted polynomials for all first level nodes. Let us assume that there exists node v in the first level such that $P(r) := P_v^{(1)}(r) \not\approx_a \mathcal{C}_{U_i^{(1)}}$ where $U_i^{(1)}$ is the corresponding correct union of its two children nodes.

With a similar argument as above, $P(r)$ can be written as a product of degree 1 polynomials with all but negligible probability (otherwise a collision in the ECRH can be found). Let X be the set containing the additive inverses of the roots x_i of $P(r)$. It follows that $P(r) = \beta\mathcal{C}_X$ for some efficiently computable $\beta \in \mathbb{Z}_p^*$. Similar as above, if $X \neq U_i^{(1)}$, $\mathcal{A}'$ can output $\{A, B, X, \Pi^* = (h_X^{\beta^{-1}}, h_I^{\beta}, W_A^{\beta^{-1}}, W_B^{\beta^{-1}}, F_A^{\beta}, F_B^{\beta})\}$. It is easy to verify that this consists a satisfying proof for the claim $A \cup B = X$, which by our previous analysis can happen with negligible probability and the claim follows.

level-j Assuming that this holds for the polynomials on level j we will show that it also holds for level $j+1$. Let us assume that this not the case. It follows that there must exist node v of the tree on level $j+1$ the children of which have extracted polynomials $Q_A(r), Q_B(r)$, the corresponding extracted output polynomial is $P(r)$ and the corresponding extracted polynomial for the intersection be $H(r)$. Assuming $P(r)$ is not an associate of $\mathcal{C}_U$ we will construct an adversary that finds a collision in the ECRH similar to above.

By assumption, $Q_A(r) = \beta_A \prod_{i \in [|A|]}(x_i + r)$ and likely for $Q_B(r)$ (recall that these are associate polynomials of the correctly computed corresponding set at level j) for sets A, B. If $P(r)$ contains an irreducible factor of degree > 1, our previous analysis shows that a collision for the ECRH is found.

Therefore $P(r)$ can be written as a product of degree 1 polynomials and a scalar and there exist an efficiently computable set X and $\beta \in \mathbb{Z}_p^*$ s.t. $P(r) = \beta\mathcal{C}_X$. Similar as above, if $X \neq A \cup B$, $\mathcal{A}'$ can output $\{A, B, X, \Pi^* = (h_X^{\beta^{-1}}, h_I^{\beta/\beta_A \cdot \beta_B}, W_A^{\beta_B}, W_B^{\beta_A}, F_A^{\beta_B^{-1}}, F_B^{\beta_A^{-1}})\}$. It is easy to verify that this consists a satisfying proof for the claim $A \cup B = X$, which by our previous analysis can happen with negligible probability and the claim follows.

Since this holds for every node of level $j+1$, this concludes our induction proof.

Hence with all but negligible probability, the claim holds for the value h_O. As per the intersection case, it must be that with all but negligible probability $O = U$.

Set Difference. The argumentation for this case follows in a relatively straight forward manner from our treatment for the union case. Let us assume that there exists $\mathcal{A}$ that outputs S_1, S_2, O, Π, f_O s.t. *verifyDifference* accepts and $O \neq X := S_1 \setminus S_2$, with non-negligible probability. We will construct an adversary $\mathcal{A}'$ that either finds a collision in h, or breaks the q-**SBDH** assumption. For ease of notation we denote $\mathcal{C}_{S_i} = Q_i(r)$ for $i = 1, 2$ and $\mathcal{C}_O = P(r)$.

Upon receiving the output from $\mathcal{A}$, adversary $\mathcal{A}'$ runs the extractor $\mathcal{E}_{\mathcal{A}}$ for value $h_{I^*} \in \Pi$ to receive polynomial $R(r)$ s.t. $g^{R(s)} = h_{I^*}$ with overwhelming probability. From Claim 1 above, it follows that, with all but negligible probability, $R(r) \approx_a \mathcal{C}_I$ where $I = S_1 \cap S_2$.

Since *verifyDifference* accepts, the following must hold:

$$\begin{aligned} e(g,g)^{P(s)R(s)} &= e(g,g)^{Q_1(s)} \\ e(g,g)^{\beta P(s) \prod_{x \in S_1 \cap S_2}(x+s)} &= e(g,g)^{\prod_{x \in S_1}(x+s)} \\ e(g,g)^{\beta P(s)} &= e(g,g)^{\prod_{x \in S_1 \setminus S_2}(x+s)}. \end{aligned}$$

In case polynomials $\mathcal{C}_{S_1 \setminus S_2}$ and $\beta P(r)$ are not equivalent, due to the above equality $\mathcal{A}$ can by outputting them find a collision in the ECRH. Therefore it must be that with overwhelming probability $\beta P(r) = \mathcal{C}_X$. Again, if $\beta \neq 1$ then the two polynomials form a collision for the ECRH, therefore with all but negligible probability, $O = X$. $\square$

For the proof of our main result we make use of Lemmas 5 and 4. Let $\mathcal{A}_{ADS}$ be a poly-size adversary that upon input the public key pk of our ECRH construction, is given oracle access to all algorithms of $\mathcal{AHSO}$. $\mathcal{A}_{ADS}$ picks initial state D_0 for the data structure and computes $auth(D_0), d_0$ through oracle access to **setup**. Consequently he chooses a polynomial number of updates and with oracle access to **update** computes $D_{i+1}, auth(D_0), d_{i+1}$ for $i = 0, ..., h$. Also, he receives oracle access to algorithms **query**,**verify**,**refresh**. Finally, $\mathcal{A}_{ADS}$ outputs $\{\alpha', \Pi, q, D_k, auth(D_k), d_k\}$ where k is between 0 and $h+1$ and denotes the snapshot of the data structure to which the query q is to be applied. We want to measure the probability that **verify**$(\alpha', \Pi, q, pk, d_k)$ outputs accept and algorithm **check**(D_k, q, α') outputs reject (i.e., α' is not equal to the set produced by applying operations in q on dataset D_k).

Assuming $\mathcal{A}_{ADS}$ can succeed in the above game with non-negligible probability ϵ, we will use him to construct $\mathcal{A}'$ that finds a collision in the ECRH with non-negligible probability. $\mathcal{A}'$ works as follows. Upon input pk of ECRH, he sends it to $\mathcal{A}_{ADS}$. Following that, he provides oracle interface to $\mathcal{A}$. Finally, he receives $\{\alpha', \Pi, q, D_k, auth(D_k), d_k\}$ from $\mathcal{A}$ and runs corresponding extractor $\mathcal{E}_{\mathcal{A}_{ADS}}$ to receive hash pre-images for all hash vales in Π.

Let $S_1, ..., S_t$ be the sets in D_k over which q is defined ($t = 2$ for the case of set difference). First $\mathcal{A}'$ computes honestly q over S_i, and receives the correct output α and all intermediate sets. Then he runs **verify** on the received tuple and checks if $\alpha \neq \alpha$. If verification fails or $\alpha = \alpha'$ he aborts (i.e. he only proceeds if $\mathcal{A}_{ADS}$ wins the ADS game). Following that, $\mathcal{A}'$ checks if $f(\mathcal{C}_{S_i}) = f_i$ for $i = 1, ..., t$. If any of the checks fails, he aborts. Then $\mathcal{A}'$ compares the correctly computed set for each node

$v \in \mathcal{T}$ and the corresponding extracted polynomial which we denote by $P_v(r)$. Given polynomial $P_v(r)$ for each node, $\mathcal{A}'$ checks if it is an associate polynomial of the characteristic polynomial of the corresponding honestly computed set. If this does not hold for some node v, he aborts. Finally, he outputs the pair of polynomials $P_{root}(r), \mathcal{C}_{\alpha'}$.

First, note that $\mathcal{A}'$ runs in time polynomial in the security parameter, since both $\mathcal{A}_{ADS}$ and $\mathcal{E}_{\mathcal{A}_{ADS}}$ run in polynomial time, the set computations can be done in polynomial time and polynomial associativity is also decidable in polynomial time by long division. Regarding, his success probability in finding a collision we argue as follows.

Let E' be the event that $\mathcal{A}'$ succeeds in finding a collision and B the event that $\mathcal{A}_{ADS}$ wins the ADS game. By assumption $\Pr[B] > \epsilon$ for non-negligible ϵ, a function of λ. Observe that, conditioned on not aborting, the probability of $\mathcal{A}'$ to find a collision is at least $(1 - \nu^*(\lambda))$ where $\nu^*(\lambda)$ is the sum of the negligible errors in the output of the extractor and the randomized factorization algorithm, which by a union bound is an upper bound for the total error probability. This holds because, since $\mathcal{A}'$ did not abort, the verification succeeded and $\mathcal{A}_{ADS}$ provided a false answer which implies that the polynomials output are not equivalent yet they have the same hash values. Overall $\Pr[E'] = \Pr[E'|\neg\text{abort}]\Pr[\neg\text{abort}] \geq (1 - \nu^*(\lambda))\Pr[\neg\text{abort}]$.

Let E_V be the event that **verify** accepts during the first step of $\mathcal{A}'$ and $\alpha \neq \alpha'$. Also, let E_1 be the event that all $f(\mathcal{C}_{S_i}) = f_i$ for $i = 1, ..., t$ given that **verify** accepts and E_2 be the event that all extracted polynomials are of the form $P_v(r) \approx_a \mathcal{C}_O$ also given that **verify** accepts. Also, let E_3 be the event that the polynomials $\mathcal{C}_{\alpha^*}(r)$ and $\sum_{i=0}^{\delta-1} c_i r^i$ are equivalent given that **verify** accepts. By Lemma 4, $\Pr[E_1] > 1 - \nu_1(\lambda)$ and $\Pr[E_3] > 1 - \nu_3(\lambda)$ since, by the Schwartz-Zippel lemma [Sch80], the probability that two non-equivalent polynomials of degree δ agree on a point chosen uniformly at random is $\leq d/2^l$ in this case, which is negligible in λ. Also, by assumption $\Pr[E_V] \geq \epsilon$.

We argue about $\Pr[E_2]$ as follows:

Claim 2. $\Pr[E_2] > 1 - \nu_2(\lambda)$.

Proof of Claim. Equivalently, we will prove that for all internal nodes $v \in \mathcal{T}$, with corresponding extracted polynomial $P_v(r)$, it must be that $P_v(r) \approx_a \mathcal{C}_O$ where O is the correctly computed set corresponding to v when computing q over S_i, with all but negligible probability.

As in the proof of Lemma 5, we will prove this by an induction on the levels of $\mathcal{T}$ (in fact, since $\mathcal{T}$ is not a balanced tree, the induction is over the nodes themselves in the order they are accessed by a DFS traversal).

level-1 If the operation for v is an intersection, then if $P_v(r)$ has a factor that is an irreducible polynomial of degree > 1, then let $R_i(r), \tilde{R}_i(r)$ be the corresponding extracted polynomials for the pair of values $W_i, \tilde{W}_i$ in the proof. Since the verification process succeeds, it follows that $e(f_O, W_i) = e(f_i, g)$. Since by assumption, $f(\mathcal{C}_{R_i}) = f_i$, (slightly abusing the notation, we assume that $S_i = R_i$) it follows that the polynomials $\mathcal{C}_{R_i}(r), P_v(r) \cdot R(r)$ form a collision for the ECRH for some index i. On the other hand, if $P_v(R)$ can be written as a product of degree 1 polynomials, it follows that it can be written as $\beta\mathcal{C}_X$ for some set X and $\mathcal{A}'$ could output appropriate proof for the claim $\cap_{i=1}^{t_v} R_i = X$, in the exact same manner as we demonstrated in proof of Lemma 5, which can only happen with negligible probability and this concludes the base case of the induction.

If the operation for v is a union, the claim immediately holds from our treatment for the union case above, for the tree $\mathcal{T}_v$ corresponding to the union operations defined in v over its children.

Likewise, if v is a set difference immediately holds from our treatment for the set difference case above.

general step Let us assume that the statement holds for all the children of node v, we show it also holds for v. Assuming there exists such node v, we can separate into two cases.

If the operation at v is an intersection, then let $Q_1(r), ..., Q_{t_v}(r)$ be the extracted polynomials corresponding to its children nodes. By assumption $Q_i(r) = \beta_i \mathcal{C}_{O_i}$ where O_i are the correctly computed sets up to that point according to q. Similar as for the case for level-1, if $P_v(r)$ contains a factor that is an irreducible polynomial of degree > 1, $\mathcal{A}'$ can find a collision in the ECRH. Therefore, with all but negligible probability, $P_v(r)$ can be written as $\beta \mathcal{C}_X$ for some efficiently computable set $X = \{x_1, ..., x_{|X|})\}$. Hence $\mathcal{A}'$ can output $\{O_1, ..., O_{t_v}, X, \Pi^* = (h_X^{\beta^{-1}}, W_i^{\beta/\beta_i}, F_i^{\beta_i/\beta};\ i = 1, ..., t_v)\}$. It is easy to verify that the above is a satisfying proof for the claim $X = \cap_{i=1}^{t_v} O_i$ which by Lemma 5 can happen with negligible probability.

If the operation at v is a union, then we argue as follows. Let $\mathcal{T}_v$ be the tree corresponding to the union operations defined in v over its children. Observe that the only difference between this case and the case analyzed previously in the proof of Lemma 5 is that the polynomials at the leafs of tree $\mathcal{T}_v$ are not characteristic polynomials necessarily. However, by assumption, they are polynomials of the form $\beta_i \mathcal{C}_{O_i}$ where O_i are the correctly computed sets up to that point according to q. $\mathcal{A}'$ can produce a satisfying proof for an incorrect set, in the *exact* same manner as described in the general step of our induction proof for Claim 1 above. Hence, with all but negligible probability, $P_v(r) \approx_a \mathcal{C}_O$, which concludes our induction proof.

A similar treatment can be used if v is a set difference. Again, the extracted polynomials $Q_1(r), Q_2(r)$ are, by the inductive assumption, of the form $\beta_i \mathcal{C}_{O_i}$ where O_i are the correctly computed sets up to that point according to q. Similar to our treatment for the intersection case, if $P_v(r)$ contains a factor that is an irreducible polynomial of degree > 1, $\mathcal{A}'$ can find a collision in the ECRH. Therefore, with all but negligible probability, $P_v(r)$ can be written as $\beta \mathcal{C}'_X$ for some efficiently computable set $X' = \{x_1, ..., x_{|X'|})\}$. Hence $\mathcal{A}'$ can output $\{O_1, O_2, X', \Pi^* = (h'^{\beta^{-1}}_X, W_1^{\beta/\beta_i}, F_1^{\beta_i/\beta}; \ i = 1, 2\}$. which is a satisfying proof for the claim $X' = S_1 \setminus S_2$ which by Lemma 5 can happen with negligible probability.

Therefore, the claim follows. ■

It follows by the way we defined these events that the overall abort probability of $\mathcal{A}'$ is (using a union bound) $\Pr[\text{abort}] \leq \Pr[E^c_V] + \Pr[E^c_1] + \Pr[E^c_2] + \Pr[E^c_3] = 1 - \epsilon + \nu'(\lambda)$ where $\nu(\lambda)'$ is the sum of the three negligible probabilities. Hence $\Pr[\neg\text{abort}] \geq 1 - 1 - \epsilon + \nu'(\lambda) = \epsilon - \nu'(\lambda)$. We showed above that $\Pr[E'] \geq (1 - \nu^*(\lambda)) \Pr[\neg\text{abort}] \geq \epsilon(1 - \nu(\lambda))$ (for an appropriately defined negligible function $\nu(\lambda)$) which is non-negligible. This contradicts the collision resistance of the ECRH h and the security of $\mathcal{AHSO}$ follows. □

Corollary 1. *If the server maintains a list of m fresh proofs $\pi_1, ..., \pi_m$ for the validity of values f_i,* **refresh** *has complexity $O(m^{2\epsilon} \log m)$, in order to update the m^ϵ proofs π_i affected by an update, and* **query** *has complexity $O(N \log^2 N \log\log N \log t + t)$.*

Corollary 2. *In a two-party setting, where only the source issues queries, proofs consist of $O(t)$ elements.*

For Corollary 1 the following modifications are made to the scheme:

- The server upon receiving $D, auth_D, d, pk$ computes and stores m proofs $\pi_1, .., \pi_m$ by running the algorithm *VerifyTree* for each value f_i corresponding to S_i. These values are computed in time $m^{1+\epsilon} \log m$.

- Upon receiving a query request, the server performs t lookups to find the corresponding proofs π_i (instead of computing them on-the-fly) and includes them in the proof.

- Upon receiving an update, modifying $f_i \to f_i^*$, let $\pi_1, ..., \pi_{m^\epsilon})$ be the proofs that corresponds to the value f_i and its $m^\epsilon - 1$ siblings in the accumulation tree. The server computes updated proofs $\pi_1^*, ..., \pi_{m^\epsilon}^*$ by running *QueryTree* m^ϵ, hence this takes overall time $m^{2\epsilon} \log m$.

Likewise for Corollary 2:

- Upon receiving query q, the server runs **query** skipping step (4).

- Upon receiving α, Π, the source computes $\prod_{i=1}^{\delta}(x_i + s)$ in time $O(\delta)$ using the secret key s. He then runs **verify** replacing steps (3),(4) with a single check of the equality $e(g^{\prod_{i=1}^{\delta}(x_i+s)}, g) = e(f_a, g)$.

3.4.4 Complexity analysis for the algorithms of the scheme

Recall that we are using the *access complexity* model and we are measuring primitive operations in $\mathbb{Z}_p^*$ ignoring a poly-logarithmic in λ cost for element representation and group operations.

Intersection

This is the most complicated argument in terms of asymptotic analysis and it will be useful for the consecutive ones, therefore we will provide an elaborate analysis. The algorithm *proveIntersection* consists of the following steps:

1. Compute values W_i for $i = 1, ..., t$.

2. Compute polynomials $q_i(r)$.

3. Compute values F_i.

For simplicity of presentation, we will assume without loss of generality that all t sets have cardinality n and we denote $N = tn$. From Lemma 3 step (1) can be done with $\sum_{i\in[t]} n \log n$ operations which can be bound by $O(N \log N)$.[9]

For the greatest common divisor computation, we will be making use of the *extended Euclidean algorithm* presented in [vzGG03] which, for two polynomials $a(r), b(r)$ of degree n runs in time $O(n \log^2 n \log\log n)$. The algorithms outputs three polynomials $u(r), v(r), g(r)$ s.t. $u(r)a(r) + v(r)b(r) = g(r)$ and $g(r)$ is the $gcd(a(r), b(r))$ and u, v are known as Bézout coefficients of a, b. Observe that $g(r)$ can be at most of degree n and by the analysis of the algorithm, $deg(u) < deg(b) - deg(g)$ and $deg(v) < deg(a) - deg(g)$. In our case, it is thus true that the degrees of polynomials u, v, g are all upper bounded by n.

The $gcd(P_1, ..., P_t)$ can be recursively computed as $gcd(gcd(P_1, ..., P_{t/2}), gcd(P_{t/2+1}, ..., P(t))$ and this can be applied repeatedly all the way to first computing the pairwise gcd of all consecutive pairs of polynomials and following that the gcd of each pair of $gcd's$ all the way to the top. In order to better analyze the complexity of step (2), let us introduce the following conceptual construction that captures exactly this recursive approach. Let $\mathcal{T}$ be a binary tree with polynomials $\mathcal{C}_{S_i \setminus I}$ at the t leafs. Each internal node is associated with one major polynomial which is the gcd of the major polynomials of its two children nodes, and two minor polynomials, which are the corresponding Bézout coefficients. The tree must be populated (all polynomials of internal nodes computed) as follows. For the nodes that are parents of leafs, compute the gcd of their children nodes and the corresponding Bézout coefficients. Following that, for each level of the tree

[9]A tighter bound would be $O(N \log n)$. However we do not wish to capitalize on the fact that we assumed all sets are of the same size, since this is an assumption for ease of notation. Hence we provide this more general bound.

all the way up to the root, the nodes are populated by computing the *gcd* of the *gcd*'s stored in their two children nodes. It follows that the root of $\mathcal{T}$ stores the $gcd(\mathcal{C}_{S_1\setminus I},...,\mathcal{C}_{S_t\setminus I})$.

Let us now analyze how long it takes to populate the nodes of $\mathcal{T}$. By the analysis of the extended Euclidean algorithm, it follows that each of the nodes that are parents of leafs can be populated in time $O(n\log^2 n\log\log n)$. Since the degrees of the *gcd* polynomials higher in $\mathcal{T}$ can only be lower, it follows that the same bound holds for all nodes. Since there exist $O(t)$ nodes, $\mathcal{T}$ can be populated in time $O(N\log^2 N\log\log N)$.

Following that, we need to compute polynomials $q_i(r)$. Observe that each such polynomial can be computed after populating $\mathcal{T}$ as the product of exactly $O(\log t)$ polynomials each of which can be at most of degree n. We start by proving the following.

Claim 3. *Having populated $\mathcal{T}$, all the polynomials $q_i(t)$ for $i = 1,...,t$ can be computed by $2t-2$ polynomial multiplications.*

Proof of Claim. We will prove the above by induction on the number of sets t. For $t=2$, having populated the tree, polynomials $q_1(r), q_2(r)$ are already stored at the root. Hence we need $2\cdot t-2=0$ multiplications. If this is true for $t=j$ we will show it is true for $2j$. Observe that for two sibling sets, the polynomials $q_i(r), q_{i+1}(r)$ can be written as $q_i = h(r)u(r)$ an $q_{i+1} = h(r)v(r)$ where u, v are the corresponding Bézout coefficients stored in their parent. The polynomials $h_k(r)$ for $k=1,...,j$ (each associated with one grand-parent node of the leafs in $\mathcal{T}$) can be computed with $2j-2$ multiplications by the assumption. Hence each polynomial $q_i(r)$ can be computed with one additional multiplication for a total of $2j$ additional multiplications. Thus the overall number of multiplications to compute $q_1(r),...,q_{2j}(r)$ is $4j-2=2t-2$, which concludes our proof of the claim. ■

Since each of $q_i(r)$ can be at most of degree $O(n \log t)$, an upper bound on the complexity of each of these multiplications is $O((n \log t) \log(n \log t))$, by using fast multiplication with FFT interpolation. By the above claim, there are $O(t)$ such multiplications, therefore, the overall complexity for the computation of the polynomials $q_i(r)$ is $O(N \log N \log t \log\log t)$. Finally, the output of this procedure is the polynomial coefficients of the q_i's hence the values F_i can be computed in time $O(N \log t)$ since each q_i has degree at most $n \log t$. Since $t \leq N$, from the above analysis the overall complexity of *proveIntersection* is $O(N \log^2 N \log\log N)$.

Algorithm *verifyIntersection* consists of $O(t)$ bilinear pairings. Finally, the size of the proof Π is $O(t)$ group elements (in practice $2t$ elements).

Union

We begin with the proof Π for a union of two sets A, B with cardinalities n_A, n_B (denote $N = n_A + n_B$). The intersection argument for $I = A \cap B$ can be computed in time $O(N \log^2 N \log\log N)$ from the above analysis. The value h_U can be computed in time $O(N \log N)$ from Lemma 3, hence the algorithm *proveUnion* for two sets runs is time $O(N \log^2 N \log\log N)$.

For the general case, let us denote with n_i the cardinality of each set S_i and let $N = \sum_{i \in [t]} n_i$. Finally, we denote with N_v the sum of the cardinalities of the sets of the children nodes of each node $v \in \mathcal{T}$. Each of the first level nodes is related to value N_i for $i = 1, ..., t/2$ s.t. $\sum_{i=1}^{t/2} N_i \leq N$). Hence computing the proofs for all first level nodes of $\mathcal{T}$ can be done in time $\sum_{i=1}^{t/2} N_i \log^2 N_i \log\log N_i$ which can be upper bound by $O(N \log^2 N \log\log N)$. Moreover, this bound is true for all levels of $\mathcal{T}$ since due to the commutativity of the union operation, no elements will be left out (in the worst case the sets are disjoint, hence $|U| = N$) and since we have exactly $\log t$ levels in the tree, the algorithm *proveUnion* in general runs in time $O(N \log^2 N \log\log N \log t)$.

Each proof for a pair of sets can be verified by checking $O(1)$ bilinear equalities and since there are exactly $t-1$ such arguments, the runtime of *verifyUnion* is $O(t)$. The proof for each node v consists of 8 group elements and there are $t-1$ such arguments, hence the size of the argument is $O(t)$ (in practice, $8(t-1)$ elements).

Set difference

Let Π be the proof for the set difference $S_1 \setminus S_2$ with cardinalities n_1, n_2 (denote $N = n_1 + n_2$). The intersection argument for $I = S_1 \cap S_2$ can be computed in time $O(N \log^2 N \log\log N)$ from the previous analysis. The value h_D can be computed in time $O(N \log N)$ from Lemma 3, hence the algorithm *proveDifference* runs is time $O(N \log^2 N \log\log N)$.

The overall runtime of *verifyDifference* is $O(1)$ as it consists of checking 4 bilinear equations. Likewise, the proof size is $O(1)$ and it consists of 6 group elements.

Hierarchical set operations

Observe that (similar to the generalized union case) the proof construction and verification consists of constructing (and verifying) a series of proofs as dictated by the structure of $\mathcal{T}$. Hence the complexity of the algorithms will be characterized by the complexity of the algorithms for the single operation case. As before we denote N_v for each node $v \in \mathcal{T}$ as the sum of the cardinalities of the sets of its children nodes and t_v as the number of its children nodes. Also let $N = \sum_{v \in \mathcal{T}} N_V$. The construction of the argument for each node can be made in time $O(N_v \log^2 N_v \log\log N_v \log t_v)$. If t is the length of the set operation formula corresponding to q, it follows that $t_v \leq t$ hence the above can be also bound as $O(N_v \log^2 N_v \log\log N_v \log t)$. Finally, the cost to compute Π is equal to the sum of computing all of the respective proofs, which can be written as $O(N \log^2 N \log\log N \log t)$. Also, each of the proofs π_i is computed in time $O(m^\epsilon \log m)$ and since there are t of them, the overall complexity for **query** is $O(N \log^2 N \log\log N \log t + tm^\epsilon \log m)$.

Each proof can be verified by checking $O(t_v)$ bilinear equalities. Since each node has a single parent it follows that the runtime of **verify** is $O(|\mathcal{T}|)$. However, $|\mathcal{T}| \leq 2t$ since all operations are defined over at least two sets, hence **verify** consists of $O(t)$ operations. Each atomic proof in Π consists of $O(t_v)$ group elements and therefore the total size of Π is $O(t + \delta)$.

3.5 Server-assisted updates

The standard ADS model presented in Chapter 2 requires that the data owner maintains a copy of D and $auth(D)$ in order to be able to perform updates. One extension to this is a scenario where not even the owner needs to store the data: All updates can be performed by the server (upon request by the owner) who sends to the owner a new digest d' and a proof that the update was executed honestly. Observe that this can always happen naively, if the owner downloads all of D and performs the update locally. However, we are interested in more efficient solutions. This closely resembles the two-party ADS model introduced in [Pap11], where a property is identified that is sufficient to turn a standard ADS into a two-party one (i.e., one that supports server-assisted updates). In practice, this model introduces two new algorithms **serverUpdate** and **verifyUpdate**, that replace the previous **update** and **refresh** process associated with an update. In this section, we provide appropriate correctness and security definitions for an ADS that supports server-assisted updates and show how our construction can be accordingly modified.

First, we define the two new algorithms that are necessary and replace **update** and **refresh** from the ADS definition. The other four algorithms of the scheme remain the same, as presented in Chapter 2.

1. $\{D_{h+1}, auth(D_{h+1}), d_{h+1}, \Pi(u), upd\} \leftarrow$ **serverUpdate**$(u, auth(D_h), d_h, pk)$: On input update u on data structure D_h, the authenticated data structure

$auth(D_h)$ and the digest d_h, it outputs the updated data structure D_{h+1} along with $auth(D_{h+1})$, and the new digest d_{h+1} with (possibly) some update-relevant information upd, and a proof $\Pi(u)$ that the update was executed correctly. This is executed by the server upon a request by the owner.

2. $\{\text{accept/reject}\} \leftarrow$ **verifyUpdate**$(u, \Pi(u), d_{h+1}, upd, d_h, sk, pk)$: On input update u, a proof $\Pi(u)$, claimed new digest d_{h+1} with (possibly) some update-relevant information upd, and the existing digest d_h it outputs either "accept" or "reject". If the answer is "accept", then is sets d_{h+1} as the new digest.

Let $\{D_{h+1}\} = applyUpdate(u, D_h)$ be a method for applying an update u at data structure D_h (this method is not part of the scheme but only introduced for ease of notation), and *check* a method for checking the validity of an answer, as defined in 2. By $applyUpdate((u_1, \ldots, u_t, D_h)$, we denote the data structure D_{h+t} that results from sequentially applying updates u_i for $1, \ldots, t$ to D_h.

Then an authenticated data structure scheme with server-assisted updates, should satisfy the following:

Correctness. We say that an ADS with server-aided updates is *correct* if, for all $\lambda \in \mathbb{N}$, for all (sk, pk) output by algorithm **genkey**, for all $(D_h, auth(D_h), d_h)$ output by one invocation of **setup** followed by polynomially-many invocations of **serverUpdate** where $h \geq 0$, for all queries q and for all $a(q), \Pi(q)$ output by **query**$(q, D_h, auth(D_h), pk)$, with all but negligible probability, whenever $check(q, a(q), D_h)$ accepts, so does **verify**$(q, a(q), \Pi(q), d_h, pk)$ and **verifyUpdate**$(u_i, \Pi(u_i), d_{h+i}, upd_i, d_{h_i}, sk, pk)$ outputs "accept" for $i = 1, \ldots, h$.

Security. Let $\lambda \in \mathbb{N}$ be a security parameter and $(sk, pk) \leftarrow genkey(1^\lambda)$ and $\mathcal{A}$ be a poly-size adversary that is only given pk and has access to the algorithms of the ADS via an oracle $O_{\lambda,ADS}$ that accepts queries in the following model: The adversary picks an initial state of the data structure D_0 and computes $D_0, auth(D_0), d_0$ through an oracle call to algorithm **setup**. Following this, he is given free oracle access to all algorithms of the ADS. We say that an ADS with server-aided is *secure* if for all large enough $\lambda \in \mathbb{N}$, for all poly-size adversaries $\mathcal{A}$ it holds that:

$$\Pr\left[\begin{array}{rl} & (q, a(q), \Pi(q), (u_i, upd_i, \Pi(u_i), d_i)_{i=1}^{t}) \leftarrow \mathcal{A}^{O_{\lambda,ADS}}(1^\lambda, pk) \text{ s.t.} \\ \text{accept} & \leftarrow \textbf{verify}(q, a(q), \Pi(q), d_t, pk) \quad \wedge \\ \text{reject} & \leftarrow check(q, a(q), applyUpdate(u_1, \ldots, u_t, D_0)) \quad \wedge \\ \text{accept} & \leftarrow \textbf{verifyUpdate}(u_i, \Pi(u_i), d_{i+1}, upd_i, d_i, sk, pk) \\ & \hfill \text{for } i = 0, \ldots, t-1 \end{array}\right] \leq \nu(\lambda),$$

where the probability is taken over the randomness of **genkey** and the coins of $\mathcal{A}$.

Next. we propose an ADS for nested set operations, with support for server-assisted updates. This construction builds upon our previous scheme. We model updates u as a set operation that needs to be evaluated over some of the existing sets. That is, an update is a combination of query q, as defined above, with an index j, that can be described as "perform query q and set $S_j = \alpha$", whereα is the output set of q.

The **setup**, **query**, and **verify** algorithms remain exactly the same. Moreover, we introduce the following two algorithms:

Algorithm$\{D_{h+1}, auth(D_{h+1}), d_{h+1}, \Pi(u), upd\} \leftarrow$ **serverUpdate** $(u, D_h auth(D_h), d_h, pk)$: Parse u as q, j and run **query**$(q, D_h, auth(D_h), pk)$ to receive α, Π. Run $QueryTree(pk, d_h, j, auth(D_j))$ to get proof of membership π_j for current value f_j. Set $upd = \{h(\alpha), f_j, \pi_j\}$. Update D_h to D_{h+1} by setting $S_j = \alpha$ and $f_j = g^{\prod_{x \in \alpha}(x+s)}$. Then update the $\mathcal{AT}$ as follows. Let v_0 be the corresponding node for the j-the set in $\mathcal{AT}$, with value $d(v_0)$. Set $d(v_0) = f_j^{j+s}$. Let $v_1, ..., v_{\lceil 1/\epsilon \rceil}$ the node path from v_0 to root r For $k = 1, ..., \lceil 1/\epsilon \rceil$ let N_{v_k} be the set of children of v_k, and set $d(v_k) = g^{\prod_{u \in N(v_k)}(\phi(d(u)+s)}$, i.e., after updating the value $d(v_{k-1})$, recompute the accumulation value and the corresponding value at the next level. Set $auth(D_{h+1})$ to the updated version of $\mathcal{AT}$, and $d_{h+1} = d(r)$ where r is the new computed root. Send $\{D_{h+1}, auth(D_{h+1}), d_{h+1}, \Pi(u), upd\}$ to the owner.

Algorithm {accept/reject} $\leftarrow$ **verifyUpdate**$(u\Pi(u), d_{h+1}, upd, d_h, sk, pk)$: Parse u as q, j and run **verify**$(q, \emptyset, \Pi, d_h, pk)$, omitting steps 3 and 4. If the output of **verify** is reject, output reject and halt. Check that for the values $f_\alpha, \tilde{f}_\alpha$ contained in upd, it holds that $e(f_\alpha, g^\alpha) = e(\tilde{f}_\alpha, g)$ and if the check fails, output reject and halt. Run $VerifyTree(pk, d_h, j, f_j, \pi_j)$ and, if it outputs reject, output reject and halt. Finally, let $d_1, \ldots, d_{\lceil 1/\epsilon \rceil - 1}$ be the corresponding node values from proof π_j and $d_{\lceil 1/\epsilon \rceil} = d_h$. Set $d'_0 = f_\alpha$ and $d_0 = f_j$. Then for $k = 1, ..., \lceil 1/\epsilon \rceil$, set $d'_k = d_k^{(\phi(d'_{k-1})+s)(\phi(d_{k-1}))+s)^{-1}}$. If $d'_k \neq d_{h+1}$ output reject and halt, otherwise output reject and halt.

The algorithm executed at the server essentially consists of three steps. First, the server evaluates the query q but instead of returning the result α, he computes its hash value h_α. Then he computes an accumulation tree proof for the old accumulation value of the previous set S_j and then replaces S_j with α and its accumulation value f_j with the accumulation value of α. Finally, he performs the necessary modifications to the accumulation tree $\mathcal{AT}$, by updating all the nodes in the path from the j-th leaf to the root, a process that results in computing the new root d_{h+1}. The only difference of this last step, compared with the **update** algorithm of our previous construction, is that, without access to sk, the server needs to re-compute all these node values from scratch.

The update verification process at the owner similarly consists of three steps. First, the owner checks that the returned hash value is truly the hash value of the honestly computed result for q. This consists of the same process as the one for **verify**, omitting the last two steps that validate that pre-image correctness of the hash value. Then, he checks the validity of the previous hash j-th accumulation value with respect to the old digest d_h. Finally, using the node values of the previous version of $\mathcal{AT}$ contained in π_j, he recomputes himself the values of all nodes from the j-th leaf upwards, using as a starting point the verified accumulation value for the new result. At each level k, using sk he efficiently removes the old value of the previous level d_{k-1}, and inserts the new value d'_{k-1}. At the end of this process, he checks that the value he computed for level $\lceil 1/\epsilon \rceil$. is equivalent to the claimed new digest d_{h+1} received by the server.

Now we can state and prove the following result.

Theorem 2. *The scheme* {**genkey**, **setup**, **query**, **verify**, **serverUpdate**, **verifyUpdate**} *is an ADS with support for server-assisted updates, for queries q from the class of hierarchical set operations formulas involving unions, intersections and set differences, of length polynomial in λ. It is* correct *and* secure *under the q-**SBDH** and the q-**PKE** assumptions.*

Proof: Correctness follows by close inspection of the algorithms above. We will prove security by reducing to the security of $\mathcal{AHSO}$, i.e., we will show how a successful adversary for this scheme can be used to break the security of our previous one. We begin with the key observation that all algorithms of our scheme can be efficiently executed with access to pk alone and, with the exception of **genkey**, they are all fully deterministic. In particular, for a given pk, D, there exists a single pair of authentication information and digest values $auth(D), d$.

Let now $(q, a(q), \Pi(q), (u_i, upd_i, \Pi(u_i), d_i^*)_{i=1,\ldots,t})$ be the tuple output by the adversary, and let D_i be the result of running $applyUpdate(u_i, D_{i-1})$ starting from D_0, and let d_i be the digest received by running **setup** on input D_i. Let us distinguish between two cases. Either there exists index $i \in [t]$ such that it holds that $d_i^* \neq d_i$, or not. In the latter case, all updates have been performed honestly, therefore it is easy to see that the tuple $(q, a(q), \Pi(q), t)$ can be used to break the security of $\mathcal{AHSO}$.

In the former case, let i' be the smallest such index. Then it must be that the result value $f^*_{\alpha_{i'}} \in \Pi(u_{i'})$ is different than $f_{\alpha_{i'}}$ where $\alpha_{i'}$ is the honestly computed accumulation value corresponding to u_i executed on $D_{i'-1}$. This holds because, the digest is computed in deterministic way from the set accumulation values and all values except for the one modified by the i'-th query are correctly computed at this point. Since the proof $Pi_(u_{i'})$ contains the pair of values $f^*_{\alpha_{i'}}, \tilde{f}^*_{\alpha_{i'}}$, from the properties of the ECRH there exists an extractor $\mathcal{E}$ that outputs a corresponding pre-image with overwhelming probability. By the analysis used in the proof of Theorem 1, with all but negligible probability, this pre-image is the characteristic polynomial of a set α^*. Since **verifyUpdate** accepts, it follows that the tuple $(q', \alpha^*, \Pi(u_{i'}), i')$, where q' is the query involved in $u_{i'}$, can again be used to break the security of $\mathcal{AHSO}$. $\square$

Complexity analysis of the algorithms. Updates at the server consist of running **query** which, as discussed previously, takes $O(N \log^2 N \log\log N \log t + tm^\epsilon \log m)$. Subsequently, the hash value of the result is computed, the complexity of which is subsumed by the above term. Then *QueryTree* takes $O(m^\epsilon \log m)$ from Lemma 4, whereas computing each $d(v_k)$ value with the public key takes $O(\epsilon m^\epsilon \log m)$, as it requires hashing a set of ϵm values. Finally $1/\epsilon$ such values are computed, therefore the overall time for **serverUpdate** is the same as **query**, i.e., $O(N \log^2 N \log\log N \log t + tm^\epsilon \log m)$. For **verifyUpdate**, we argue as follows. Running **verify**, omitting steps 3,4, takes $O(t)$ where t is the number of operations in u. Checking the validity of the final hash value and running *VerifyTree* takes $O(1)$ as they both involve a constant number of operations. Finally, computing the new root takes $1/\epsilon$ operations (the owner has sk and does not recompute the values from scratch), hence the total cost is $O(t)$.

Expressiveness of server-assisted updates. The way we described server-assisted updates above, each u is expressed as an arbitrary combination of set operations among existing sets. While this is a large class of operations, it does not cover all possible updates and, in particular, it does not cover the simple case of element insertion-to/deletion-from an existing set, i.e., the updates supported by $\mathcal{AHSO}$. However, this is not a real limitation of our construction; we simply chose this style of presentation for u to facilitate notation. In fact, with very small modifications, our construction can support server-assisted updates where u is expressed as $q, j, R_1, \ldots, R_t$ where q is a query and j is an index, same as before. Moreover $R_1, \ldots, R_t$ are (external) sets of elements that will be used (possibly in conjunction with some of the set S_i in D, as specified by q) that will participate in the evaluation of the query. Note that, this already includes element insertion/deletion as a special case. The only significant change in the construction is that for the hash values $h(R_i)$ there is no need to provide or verify accumulation tree proofs; instead, the owner computes these values himself, making verification much simpler.

3.6 Extensions and implementation decisions

Reducing the proof size. The size of proof Π can be reduced to being independent of the size of the final answer α. Observe that what makes the proof be of size $O(t+\delta)$ is the presence of coefficients $\mathbf{c}$. However, given α itself, coefficients $\mathbf{c} = (c_0, ..., c_{\delta-1})$ can be computed using an FFT algorithm in time $O(\delta \log \delta)$. An alternative to the above scheme would be to omit $\mathbf{c}$ from the proof and let the verifier upon input α compute the coefficients by himself to run the last step of **verify**. That would give a proof size of $O(t)$ and verification time of $O(t + \delta \log \delta)$. Since in most real world applications $\delta \gg t$, a proof that has size independent of δ is useful, especially if one considers that the additional overhead for verification is logarithmic only. Of course the communication bandwidth is still $O(t+\delta)$ because of the answer size, but it does not extend to the proof size.

A note on the public key size. A downside of our construction -and all other constructions that are provably secure under a q-type assumption- is the large public key size. More specifically, the public key pk is of size linear to the parameter q where q is an upper bound on the size of the sets that can be hashed. This holds not only for the original sets $S_1, ..., S_m$ but for any set that can result from hierarchical set operations among them thus a natural practical bound for q is $|D|$. While computing this public key cannot be avoided and it is necessary for proof computation at the *server*, a *client* that needs to verify the correctness of query q with corresponding answer α of size δ, only needs the first $max\{t, \delta\}$ elements of the public key. By deploying an appropriate authentication mechanism (digital signatures, Merkle trees, accumulation trees etc.) to validate the elements of pk, a scheme that relieves clients from the necessity to store a long public key can be constructed. Ideally the necessary public key elements should be transmitted alongside proof Π and cached or discarded at the behest of the client.

Symmetric vs. asymmetric pairings. Throughout the presentation of our scheme, we assumed implicitly that the pairing $e(\cdot, \cdot)$ is symmetric (i.e., Type-1 pairing). For example for the construction of the union argument for the operation $A \cup B$, the value f_B appears both in term $e(f_A, f_B$ and term $e(f_B, g)$ and we assumed that in both cases the same value is used as input for the pairing, as is the case if e is symmetric. However, many times asymmetric pairings are preferable for implementation purposes since they are much faster than asymmetric ones in terms of computation. This is not a serious problem for our scheme as there is an easy way to circumvent it.

A pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ is asymmetric if $\mathbb{G}_1 \neq \mathbb{G}_2$ but both are of prime order p and let g_1, g_2 be respective generators. Observe that $e(g_1^{P(s)}, g_2) = e(g_2, g_2^{P(s)})$ is an efficiently checkable equality that verifies that two hash values (their first parts)

$f^1 = g_1^{P(s)}, f^2 = g_2^{P(s)}$ have the same pre-image but are computed in $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively. Therefore, by including both values f_A^1, f_A^2 in the proof, the case of an asymmetric pairing can be accommodated. By verifying the above equality a prover can be sure that both values refer to the same characteristic polynomial and use either one of them selectively, as dictated by the argument verification algorithm. By using the naive approach of including the "dual" hash value of each element in the proof, we see that the proof size can at most double but maintains the same asymptotic behaviour, i.e., proofs have size $O(t + \delta)$ and the same holds for the runtime of the verification algorithm. In practice, a smarter approach can be taken where only necessary elements are added (the ones that participate in union arguments and, of these, half can be "routed" through $\mathbb{G}_1$ and the other half through $\mathbb{G}_2$). Another by-product of using an asymmetric pairing it that the public key size is doubled $(g_1, ..., g_1^{s^q}, ..., g_2, ..., g_2^{s^q})$ and likewise for the setup phase cost for the source. Note that no isomorphism between $\mathbb{G}_2$ and $\mathbb{G}_1$ is explicitly used in the above process, hence our construction can work both with Type-2 and Type-3 pairings.

3.7 Experimental evaluation

In this section we present an experimental evaluation of our construction. The scheme was written in C++, by building upon the bilinear accumulator implementation of [Tre13] and the DCLXVI [DCL16] library for bilinear pairings over a 256-bit curve with an asymmetric pairing, in order to implement our ECRH. For modular arithmetic we used the FLINT library [Fli16]. All experiments were run on a 64-bit machine with Intel Core i5 CPU 2.5GHz, running Linux. The code was compiled using g++ version 4.7.3 in C++11 mode. Our goal was to measure important quantities related to the execution of our scheme, with a focus on the verification time for the clients and the proof generation time for the server.

Experimental setup. We used a synthetic dataset consisting of 100 sets. Each set had 10,000 elements and each element was a 256-bit value. During the setup phase, the owner deployed a 1-level accumulation tree (using our established notation, $\epsilon = 1$). We tested two distinct types of queries, a single union over two sets, i.e., $X \cup Y$, and a more elaborate operation over four sets expressed as the formula $(X \cup Y) \cap (W \setminus Z)$, i.e., a union and a set difference followed by an intersection over their results. For our tests, we engineered the synthetic input sets so that they overlapped approximately on half of their elements, i.e., for the union case the final result was of size approximately $3n/2$ assuming that $|X| = |Y| = n$. Likewise, for the more elaborate operation the final result was of size approximately $n/4$. To measure the verification and proof construction overhead, we varied the input set size n between 2^7 and 2^{10}.

Verification time. In Figure 3·6 (left) we demonstrate the verification cost at the client for the two query types versus the size of the input sets. Observe that, in both cases, the verification overhead grows with the size of the sets. However, despite the fact that a single union is a conceptually simpler operation, its verification cost is considerably larger. This is due to the fact that the verification cost is strongly dominated by the computation of the hash value of the result (step 3 in the verification process). As explained above, the result size for the complex query is much smaller than that of the union. Most importantly, the overall verification cost is below 1.2 seconds for the union and 240 milliseconds for the complex query for inputs of size 1024 elements each. As a final comment, we stress that in our implementation we used the technique described in Section 3.6 to reduce the proof size. That is, the server did not send the coefficients of the characteristic polynomial of the result; instead the client computed the characteristic polynomial himself, using an FFT interpolation.

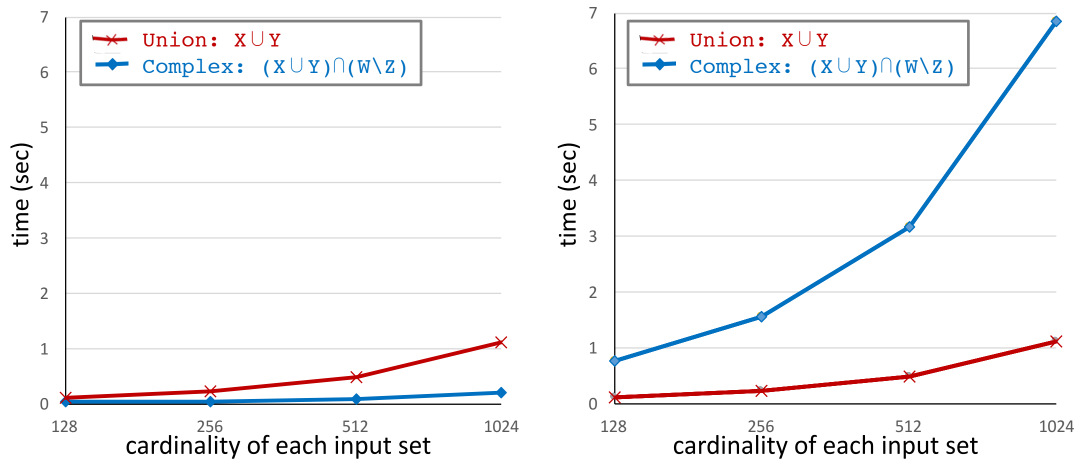


Figure 3·6: Overhead for verification at the client (left) and proof computation at the server (right) versus the cardinality of each input set for two types of queries.

Query time. Figure 3·6 (right) shows the server's overhead for proof generation for our two tested queries. The overall overhead again increases with the size of the input sets but, contrary to the verification case, proof generation takes much longer for the complex query than for the single union. This comes as no surprise since, according our analysis, this cost depends strongly on the sizes of the input and intermediate sets (unlike the verification overhead that only depends on the result size and the number of operations). For the union case, the proof construction took a little above 1 second for sets of 1024 elements each, whereas for the complex query the overhead was approximately 7 seconds for the same input sizes This cost is dominated by the proof construction for each atomic operation and a very small percentage (3-9%) comes from computing the accumulation tree proof. Finally, note the exponential growth along the x-axis; the proof generation overhead only increases quasi-linearly with the input sizes, as predicted from our asymptotic analysis.

Other quantities. While our main goal was to measure the verification and proof construction overheads, we discuss here some other measurements beginning with the setup time at owner. While this is a one-time cost it is still important to keep it reasonable. For our tested setting (with 100 sets of 10,000 elements each) setup took approximately 9.5 seconds. The reason this is so small is that the owner has access to the secret key, therefore each hash value component is computed with $n = 10,000$ modular multiplications and only a single exponentiation. Regarding the update cost, we only tested source updates, namely, insertions and deletions on a single set. For a batch update involving 100 operations (element insertions or deletions) the cost at the owner is barely above 10 milliseconds, once again capitulating on the fact that he has access to the secret key. Finally, regarding the proof size, in both cases it is only dependent on the number of operations (see also our comment above for the used techniques for proof size reduction). In particular, for the single union case the proof consists of 7 bilinear group elements, whereas for the complex query it consists of 21 such elements.

Discussion and comparison with alternative schemes. The above experiments demonstrate that our scheme can be run in practice and, in some cases, the involved overheads are far from prohibitive, particularly for verification at the client. On the other hand, if one tries to compare this cost with the time it takes to perform the computation itself if one downloads the original sets from the server (which is in the order of microseconds for the given set sizes), it is clear that there is long way to go in terms of making our schemes truly practical. Yet, our construction has the benefit of not having to process the sets locally at the client, therefore, for the case of very large input sets with a small final result the client can still benefit from our approach

In order to better quantify the performance overhead of our scheme, we compared it with a state of the art general-purpose VC scheme applied for the case of a single union over two sets, using the `libsnark` library [lib16], To implement this, we created an arithmetic circuit that upon input two sets and their (candidate) hash values (using a standard cryptographic hash function), it proceeds in two steps: First, it checks that the two sets are correct pre-images for the given hash values (by re-computing these hashes and checking that they are the same as the ones provided) and then returns the union of the two sets.[10] For the union computation we chose the naive "compare-all-pairs" approach. We stress that for the given input sizes (between 2^7 and 2^10 elements for each set) this approach yields the smallest circuit in terms of number of gates.

Regarding verification time, this general-purpose scheme behaves very similarly to our solution. In particular, it outperforms our construction by less than 5% for all tested input sizes. However, the big difference in performance occurs at the proof computation overhead. There, our scheme is anywhere between 2 and 3 orders of magnitude faster than the general-purpose approach. Specifically for sets of cardinality 1024, the circuit-based VC took approximately 17 minutes whereas our construction took a little more than 1 second. We need to stress that more than 85% of this overhead originates from computing the hashes of the input sets and testing them against the provided hash values. In our implementation, we used SHA-256 which is part of the `libsnark` library. However, by replacing it with a more "circuit-friendly" hash function (i.e., a function that can be represented as a circuit with fewer arithmetic gates), such as [Ajt96] we believe that this cost can be drastically reduced. Finally, these times were achieved using a pre-processing SNARK which means that for each input size we had to construct (and cryptographically pre-process) a different arithmetic circuit. On the other hand, our scheme has a single pre-processing phase after which any possible set operation for any input cardinality can be accommodated. Alternatively we could have used a SNARK without pre-processing (e.g., the construction of [BCTV14]), which, however, would impose an

[10]More formally, the circuit takes the two sets, the two (candidate) hash values and a (candidate) union and it checks whether the sets hash to the given values and whether their union is the same as the one provided, outputting 1 if that is the case and 0 otherwise.

even larger proof generation overhead.

Chapter 4

Verifiable Multi-dimensional Range Queries

4.1 Introduction

In this chapter, we target the case where the client issues a *multi-dimensional range query*. We model the owner's database as a table T that contains n tuples with m attribute values. A range query is defined over d out of the m attributes, which we refer to as dimensions. It is expressed as d pairs of values l_i, u_i, each along a certain dimension a_i. Its result includes all the tuples whose value on a_i is in range $[l_i, u_i]$ for *all* dimensions a_i specified in the query.

This query is fundamental in a vast variety of applications. For instance, it is a typical `SELECT`...`WHERE` query in conventional relational databases. Moreover, it is a frequent query in the emerging scientific databases (e.g., it is called `subarray` in SciDB [Bro10]). Relational and scientific database systems manage numerous types of data, such as corporate, stock, astronomical, medical, etc. With the advent of "big data", such systems are commonly deployed by third party servers in massively distributed architectures, in order to address the issue of scalability. Integrity assurance is a desirable property that serves both as a guarantee against a possibly malicious server, but also as a tool for error detection.

4.1.1 Prior work

The most basic authentication problem is *set membership*, i.e., whether an element belongs in a data collection. Well-known example schemes include Merkle trees [Mer89] and accumulation trees [PTT08]. At the opposite extreme, general-purpose VC can be used to verify any possible query on outsourced data, as discussed in Chapter 1. Although such protocols can address our problem, they incur an excessive proof cost overhead at the server, due to their generality. Therefore, there is a large variety of specialized constructions that have been proposed in the literature for the problem of authenticated multi-dimensional range queries.

Martel et al. [MND$^+$04] provide a generalization of Merkle trees, which captures the case of multi-dimensional range queries. Chen et al. [CMH$^+$08] proposes a solution that is similar to [MND$^+$04], based on attribute domain partition and access control. For the restricted case of 1-dimensional queries, Li et al. [LHKR06] propose a variant of the B$^+$-tree that incorporates hash values similarly to the Merkle tree, for processing queries in external storage. Yang et al. [YPPK09b] extend this idea to multiple dimensions, by transforming the R*-tree [BKSS90] into a Merkle R*-tree. There are also other cryptographically augmented data structures (e.g., [MNT06, CT09] based on signatures instead of hashes).

The existing literature suffers from the following critical problems. On the one hand, the schemes of [MND$^+$04, CMH$^+$08] that provide guaranteed (non-trivial) complexity bounds, scale *exponentially* with the number of dimensions d. On the other hand, the rest of the approaches rely on the heuristic R*-tree and fail to accommodate more than a limit of dimensions in practice (e.g., more than 8), as the performance and effectiveness of the index deteriorate with dimensionality. Most importantly, *all* methods require an *exponential* in m number of structures to support queries on every possible combination of dimensions in the database. This is because each structure is built on a *specific* set of dimensions, and different sets require separate structures.

Finally, there is a work by Xu [Xu10] that, contrary to [MND$^+$04, CMH$^+$08], scales quadratically with d. However, this scheme falls within a different model, as it necessitates multi-round interaction between server and client (as opposed to our non-interactive setting). Its security is based on non-falsifiable "knowledge-type" assumptions. Moreover, this scheme makes use of functional encryption [BSW11], considerably reducing its potential for implementation.

4.1.2 Overview of result

We introduce a new ADS for multi-dimensional range query processing. Our construction offers two novel powerful properties: (i) it is the first scheme where *all* costs (i.e., setup, storage, update, proof construction, verification, and proof size) grow only *linearly* with the number of dimensions, a huge improvement over the current literature, and (ii) it is the first to support an *exponential* in m number of range queries with *linear* in m setup cost and storage. One downside of our construction is that update cost also grows linearly (in the worst case) with the database size. To remedy this, we also present an update-efficient version of our construction, that significantly reduces this cost as can be seen in Figure 4·1.

In that sense, the main result of our construction is that it takes authenticated range query processing to *arbitrary* dimensions, both in terms of number and choice. Table 4·1 provides a comparison of the asymptotic complexities of both versions of our scheme against previous works (with non-trivial bounds).

4.1.3 Overview of techniques

The central idea of our solutions is the reduction of the multi-dimensional range query to *set operations* over appropriately defined sets in the database. In particu-

Scheme	Setup	Proof size	Proof construction	Verification	Update
Martel et al. [MND+04]	$O(\|T\| \log^{m-1} \|T\|)$	$O(\log^{d-1} \|T\|)$	$O(\log^{d-1} \|T\|)$	$O(\log^{d-1} \|T\|)$	$O(\log^{m-1} \|T\|)$
Chen et al. [CMH+08]	$O(\|T\| \log^{m} N)$	$O(\log^{d} N)$	$O(\log^{d} N)$	$O(\log^{d} N)$	$O(\log^{m} N)$
Section 4.4.2 with [Mer89]	$O(\|T\| \log n)$	$O(d \log n)$	$\tilde{O}(\sum_{i=1}^{d} \|R_i\|) + O(d \log n)$	$\tilde{O}(\|R\|) + O(d \log n)$	$O(\|T\|)$
Section 4.4.2 with [PTT08]	$O(\|T\| \log n)$	$O(d)$	$\tilde{O}(\sum_{i=1}^{d} \|R_i\|) + O(dn^{\epsilon} \log n)$	$\tilde{O}(\|R\|) + O(d)$	$O(\|T\|)$
Section 4.5.1 with [Mer89]	$O(\|T\| \log n)$	$O(d \log n)$	$\tilde{O}(\sum_{i=1}^{d} \|R_i\|) + O(d \log n)$	$\tilde{O}(\|R\|) + O(d \log n)$	$O(m\sqrt{n})$
Section 4.5.1 with [PTT08]	$O(\|T\| \log n)$	$O(d)$	$\tilde{O}(\sum_{i=1}^{d} \|R_i\|) + O(dn^{\epsilon} \log n)$	$\tilde{O}(\|R\|) + O(d)$	$O(m\sqrt{n})$

Figure 4·1: m: # attributes, n: # tuples, $|T|(= mn)$: database size, d: # dimensions, R_i: partial result at dimension a_i, R: query result, N: maximum domain size, $\epsilon \in (0, 1]$

lar, in a one-time setup stage, the owner builds a novel authenticated structure over every database attribute *separately*, and then binds all structures using an existing membership structure (e.g., [Mer89, PTT08]). Given a query involving *any* set of dimensions, the server decomposes it into its d 1-dimensional ranges, and processes them *individually* on the structure of each dimension, producing d proofs for the partial results $R_1, \ldots, R_d$. The main challenge is for these d proofs to (i) be *combinable* such that they verify the *intersection* of R_i, which is the final result R, and (ii) *be verifiable without the partial results*, so that the total proof size and verification cost are independent of their (potentially large) sizes. We address this challenge through an elaborate fusion of existing and novel *intersection*, *union*, and *set difference* protocols, based on bilinear accumulators.

This particular treatment of the problem, i.e., the authentication of an arbitrary d-dimensional range query via the combination of d separate 1-dimensional proofs, would not be feasible without the recent advances in set operation authentication (e.g., [PTT11, CPPT14]). We anticipate that future research will substantially improve the efficiency of the set operation sub-protocols. Motivated by this, as an additional important contribution, we identify and abstract the set operation sub-protocols needed as building blocks in our schemes, and formulate a general framework that can integrate any future improved machinery for set operation authentication.

We formally prove our construction secure under the q-Strong Bilinear Diffie-Hellman [BB08] assumption and the security of the underlying set membership schemes. We also provide an experimental evaluation, demonstrating the feasibility of our schemes.

4.2 Set membership and set operations authentication

Consider that a data owner outsources a set X to an untrusted third-party server. Clients issue queries about a single element $x \in X$. A set membership authentication protocol ($\mathcal{SMA}$) allows the server to prove to a client that x is indeed a member of X. An $\mathcal{SMA}$ is a collection of algorithms KeyGen, Setup, Prove, Verify and Update. The owner executes Keygen and Setup prior to outsourcing X. The former generates a secret and public key pair sk, pk, whereas the latter produces a digest δ that is a succinct cryptographic representation of X. The owner keeps sk and publishes pk and δ. Given a client query about $x \in X$, the server runs Prove to produce a proof of membership π. Given pk, δ, π and x, the client runs Verify to check the membership of x in X. An $\mathcal{SMA}$ is *secure*, if the probability that ($\mathtt{accept} \leftarrow \mathsf{Verify} \wedge x \notin X$) is negligible. In case the owner modifies X by inserting/deleting element, it executes Update to produce a new digest δ about the updates X, and notifies the server about the changes.

The most well-known $\mathcal{SMA}$ is the *Merkle tree* [Mer89], which is a binary tree where (i) each leaf node contains an element $x \in X$, and (ii) each non-leaf node stores the hash of the values of its children, using a CRHF H. During Setup, the owner builds a Merkle tree on X, signs the hash value in the root, publishes it as the digest δ, and sends the tree to the server. During Prove, the server accesses the tree to find x, and includes in proof π all sibling hash values along the path from the root to the leaf storing x. In Verify, the client recursively performs the hash operations to reconstruct the root hash value, and checks it against δ. For n elements, the proof construction and size, verification, and update time are all $O(\log n)$, whereas the setup is $O(n)$. An alternative $\mathcal{SMA}$ is the *accumulation tree* [PTT08, PTT11], which we discussed in detail in Section 3.4.1. The accumulation tree features two main differences to the Merkle tree: (i) the fanout of each non-leaf node is $n^{1/\epsilon}$, where $\epsilon \in (0,1]$ is a user-defined parameter, and (ii) each non-leaf node stores an *accumulation value* (discussed below) produced over the values of its children. This $\mathcal{SMA}$ offers $O(1)$ proof size, verification, and update time, and $O(n)$ setup cost. The downside is the proof construction overhead, which is now $O(n^\epsilon \log n)$, and the costly operations as opposed to the Merkle tree (exponentiations vs. hashes).

Going beyond set membership, we now discuss the case of authenticating set operations. Consider an owner of a collection of sets $\mathcal{X} = \{X_1, \ldots, X_m\}$, who outsources them to an untrusted server. Clients issue queries describing set operations over $\mathcal{X}$, consisting of *unions*, *intersections*, and *set differences*. Example queries include $X_1 \cap X_5$, $(X_2 \cup X_3) \cap X_1$, and $X_1 \setminus X_2$. A set operation authentication protocol ($\mathcal{SOA}$) enables the server to prove the integrity of the result. Similarly to $\mathcal{SMA}$, it is comprised of algorithms KeyGen, Setup, Prove, Verify Update, and its security is defined as the inability of the server to present a false answer with an accepting proof.

In the previous chapter we presented a $\mathcal{SOA}$ scheme that can support any combination of polynomially many hierarchical set operations. For the constructions of this chapter we will rely on the conceptually simpler scheme of [PTT11] that can support queries expressed as a *single* set operation (for instance $X_i \cap \ldots \cap X_j$, i.e., one intersection over an arbitrary number of sets), at the same asymptotic complexities. Specifically, for a query on collection $\mathcal{X}_Q \subseteq \mathcal{X}$ of d sets with result R, the proof has size $O(d)$, and is generated in time $\tilde{O}(\sum_{X \in \mathcal{X}_Q} |X|)$. Note that the proof construction incurs only a poly-logarithmic overhead compared to the result computation time, which is $\Omega(\sum_{X \in \mathcal{X}_Q} |X|)$. The verification overhead is $\tilde{O}(|R|) + O(d)$, whereas the setup cost is $O(\sum_{X \in \mathcal{X}} |X|)$. Although our construction from Chapter 3 subsumes [PTT11] in terms of functionality, its security relies on non-standard "knowledge-type" assumptions.

We next describe the intersection scheme of [PTT11], as we utilize it in our constructions. This scheme employs the *bilinear accumulator* primitive [Ngu05], which can be seen as a simpler variant of our ECRH construction (without the extractability property). Let X be a set with elements from $\mathbb{Z}_p$, and $s \leftarrow_R \mathbb{Z}_p^*$ a secret. Recall that the *accumulation value* of X is defined as: $acc(X) = g^{\prod_{x \in X}(x+s)}$. As discussed in 2, this value is a succinct, collision-resistant cryptographic representation of X under q-SBDH. It is also computable (from scratch) even without s, by having access to the public pairing parameters pub, as well as a public key $(g^s, ..., g^{s^q})$, where q is a user-defined parameter that is an upper of bound on the cardinality of X. In particular, we can write $\prod_{x \in X}(x+S) = P_X(S) = \sum_{i=0}^{|X|} c_i S^i$, where S is an undefined variable. The coefficients $c_0, ..., c_{|X|}$ can be computed in time $O(|X| \log |X|)$ using FFT interpolation. One can compute $acc(X) = g^{P_X(s)} = \prod_{i=0}^{|X|} (g^{s^i})^{c_i}$ using only the public information. Note that, with access to s, the bilinear accumulator can accommodate an insertion/deletion in X with $O(1)$ operations [Ngu05]. However, without s, the updated accumulation value must be computed from scratch.

In order to prove to a client with access to $acc(X_1)$, $acc(X_2)$ that a set I is the intersection $X_1 \cap X_2$, it suffices to prove that (i) $I \subseteq X_1$ and $I \subseteq X_2$, and (ii) $(X_1 \setminus I) \cap (X_2 \setminus I) = \emptyset$. Towards (i), the server must send *subset witnesses* W_1, W_2 to the client, where $W_i = acc(X_i \setminus I)$ for $i = 1, 2$. To verify (i), the client first computes $acc(I)$, and checks the following for $i = 1, 2$:

$$e(acc(I), W_i) \stackrel{?}{=} e(acc(X_i), g) \ .$$

For (ii), the server computes two *disjointness witnesses* F_1, F_2 as follows. Since $(X_1 \setminus I) \cap (X_2 \setminus I) = \emptyset$, $P_{X_1 \setminus I}(S) = \prod_{x \in X_1 \setminus I}(x+S)$ and $P_{X_2 \setminus I}(S) = \prod_{x \in X_2 \setminus I}(x+S)$ have greatest common divisor of degree zero. Hence, there exist polynomials $Q_1(S), Q_2(S)$ such that $Q_1(S) \cdot P_{X_1 \setminus I}(S) + Q_2(S) \cdot P_{X_2 \setminus I}(S) = 1$. These polynomials (also known as

Bézout coefficients) are efficiently computable by the Extended Euclidean algorithm. The server calculates the disjointness witnesses as $F_1 = g^{Q_1(s)}, F_2 = g^{Q_2(s)}$. To verify (ii), the client simply checks

$$e(W_1, F_1) \cdot e(W_2, F_2) \overset{?}{=} e(g, g)$$

This approach naturally generalizes for $d > 2$ sets X_i, with corresponding *intersection proof* $\pi_\cap = \{W_i, F_i\}_{i=1}^{d}$. In our security proofs, we use the following lemma from [PTT11]:

Lemma 6 ([PTT11])**.** *Let λ be a security parameter, and pub $\leftarrow \mathsf{GenBilinear}(1^\lambda)$. Under the q-**SBDH** assumption, no poly-size adversary can, on input pub and elements $(g, g^s, ..., g^{s^q}) \in \mathbb{G}$ for some s chosen at random from $\mathbb{Z}_p^*$, output sets $X_1, \ldots, X_d, I$ with elements in $\mathbb{Z}_p$, where $d = \mathsf{poly}(\lambda)$, and proof $\pi_\cap = \{W_i, F_i\}_{i=1}^{d}$, such that $e(acc(I), W_i) = e(acc(X_i), g)$, $\prod_i e(W_i, F_i) = e(g, g)$, and $I \neq \bigcap_i X_i$, for $i = 1, \ldots, d$, except with probability $\nu(\lambda)$.*

4.3 Problem formulation

In this section we describe our targeted setting, formulate our authentication protocol, and model its security.

Setting and query. Our setting involves three types of parties; an *owner*, a *server*, and a number of *clients*. The owner outsources to the server a dataset T that consists of n tuples, each having a set $A = \{a_1, \ldots, a_m\}$ of attributes. This dataset can either be perceived as a table in traditional relational databases. It could also be a multi-dimensional array in scientific databases (e.g., SciDB [Bro10]), where a subset of the attributes are the array dimensions (i.e., the array indices), and the rest are the array attributes (i.e., the array cell values). In addition, the server is responsible for maintaining the dataset, upon receiving tuple updates (modeled as insertion/deletion requests) from the owner.

Clients issue *multi-dimensional range queries* on T to the server, which return the tuples from T that satisfy certain range conditions over a set of attributes. More formally, a query Q is specified over *any subset* of d attributes $A_Q \subseteq A$, where $|A_Q| = d \leq m$, and encoded by the set of triplets $\{(i, l_i, u_i)\}_{a_i \in A_Q}$. The result of Q is a set $R \subseteq T$, denoted as $R(Q, T)$, that contains exactly those tuples $t \in T$ that satisfy $l_i \leq t.a_i \leq u_i$ for all $a_i \in A_Q$. This query corresponds to a `select`...`where` query in relational databases, and a `subarray` query in scientific databases. In our terminology, each $a_i \in A_Q$ represents a dimension in the multi-dimensional range query.

In our setting, we consider that the server is *untrusted*, and may present to the client a tampered result. Our goal is to construct a protocol for authenticated multi-dimensional range queries, which allows the client to verify the integrity of the received result.

Range-query authentication protocol. Let T_j denote the version of dataset T after j rounds of updates. An *authenticated multi-dimensional range query protocol* ($\mathcal{AMR}$) consists of the following algorithms:

1. $\mathsf{KeyGen}(1^\lambda)$: It outputs secret and public keys sk, pk.

2. $\mathsf{Setup}(T_0, sk, pk)$: It computes some authentication information $auth(T_0)$ and digest δ_0, given dataset T_0, sk and pk.

3. $\mathsf{Update}(upd, auth(T_j), \delta_j, sk, pk)$: On input update information upd on T_j, $auth(T_j)$, δ_j and sk, it outputs an updated dataset T_{j+1}, along with new $auth(T_{j+1})$, and δ_{j+1}.

4. $\mathsf{Prove}(Q, R, T_j, auth(T_j), pk)$: On input query Q on T_j with result R, and $auth(T_j)$, it returns R and proof π.

5. $\mathsf{Verify}(Q, R, \pi, \delta_j, pk)$: On input query Q, result R, proof π, digest δ_j and pk, it outputs either `accept` or `reject`.

In a pre-processing stage, the owner runs KeyGen and Setup. It publishes public key pk and digest δ_0, which is a succinct cryptographic representation of initial dataset T_0. Moreover, it sends $T_0, auth(T_0)$ to the server, where $auth(T_0)$ is some authentication information on T_0 that will be used by the server to construct proofs. The owner maintains its dataset by issuing Update when changes occur at the dataset. Specifically, an update is a tuple insertion or deletion, encoded by *upd*. An update on T_j produces a new version T_{j+1}, as well as new digest δ_{j+1} and $auth(T_{j+1})$. The owner sends to the server only the modified parts necessary for computing $T_{j+1}, auth(T_{j+1}), \delta_{j+1}$. The server responds to a query Q from the client by first computing the result R, and executes Prove that constructs the corresponding proof π. Finally, the client validates the integrity and freshness of R as an answer to Q on current T_j, by running Verify.

An $\mathcal{AMR}$ must satisfy the following two properties:

Correctness. A $\mathcal{AMR}$ is *correct* if, for all $\lambda \in \mathbb{N}$, $(sk, pk) \leftarrow \mathsf{KeyGen}(1^\lambda)$, all $(T_0, auth(T_0), \delta_0)$ output by one invocation of Setup followed by j' calls to Update on updates *upd*, where j' is $\mathsf{poly}(\lambda)$, for any Q, and π output by $\mathsf{Prove}(Q, T_j, auth(T_j), pk)$, $\mathsf{Verify}(Q, R(Q, T_j), \pi, \delta_j, pk)$ returns `accept` with probability 1, for all $j \leq j'$.

Security. Let $\lambda \in \mathbb{N}$ be a security parameter, key pair $(sk, pk) \leftarrow \mathsf{KeyGen}(1^\lambda)$, and $\mathcal{A}$ be a poly-size adversary that is only given pk and has access to the algorithms of the $\mathcal{AMR}$ via an oracle $O_{\lambda,\mathcal{AMR}}$ that accepts queries in the following model: The adversary picks an initial state of the dataset T_0 and receives $T_0, auth(T_0), \delta_0$ through oracle access to Setup. Then, for $i = 0, ..., j' - 1 = \mathsf{poly}(\lambda)$, $\mathcal{A}$ issues an update upd_i for T_i and receives $T_{i+1}, auth(T_{i+1})$ and δ_{i+1} through oracle access to Update. At any point during these update queries, $\mathcal{A}$ can make polynomially many oracle calls to algorithm Prove. Finally, $\mathcal{A}$ picks an index $0 \leq j \leq j'$, a query Q, a result R^* and a proof π^*. We say that a $\mathcal{AMR}$ is *secure* if for all large enough $\lambda \in \mathbb{N}$ and all poly-size adversaries $\mathcal{A}$, it holds that:

$$\Pr\left[\begin{array}{c}(Q, R^*, \pi^*, j) \leftarrow \mathcal{A}^{O_{\lambda,\mathcal{AMR}}}(1^\lambda, pk) \text{ s.t} \\ \mathtt{accept} \leftarrow \mathsf{Verify}(Q, R^*, \pi^*, \delta_j, pk) \\ \wedge R^* \neq R(Q, T_j)\end{array}\right] \leq \nu(\lambda),$$

where the probability is taken over the randomness of the algorithms and the coins of $\mathcal{A}$.

As an additional remark, note that the above protocol falls within the framework of authenticated data structures, as presented in Chapter 2, rephrased for the specific problem of range queries. We dropped the refresh algorithm as, in our construction, its mode of operation is rather trivial, i.e., it simply consists of replacing values, without any computational component.

4.4 Basic scheme

4.4.1 A general framework

We present our proposed framework, outline its benefits, and highlight the challenges behind a secure and efficient instantiation.

Framework. Recall that the query result is a set of tuples, each consisting of m attribute values, and satisfying certain range conditions. For the sake of simplicity, we henceforth define result R of query Q on dataset T as a set containing exactly the *hash value* $h_i = H(t_i)$ of the binary representation of each tuple $t_i \in T$ that satisfies the query conditions, under a CRHF $H : \{0,1\}^* \rightarrow \mathbb{Z}_p$. Our $\mathcal{AMR}$ constructions will focus on proving that R is the correct set of hash values corresponding to the tuples

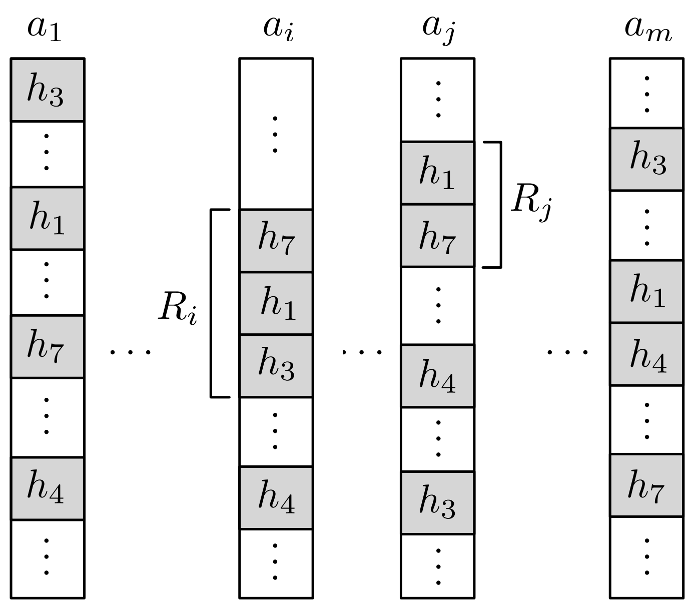


Figure 4·2: Illustrating the different tuple orders per attribute

satisfying the query. Then, given these hash values along with the full result tuples, the client can validate the integrity of each result tuple t_i by testing $H(t_i) \overset{?}{=} h_i$. Due to collision-resistance of H, the server cannot return a falsified t_i^* such that $H(t_i^*) = h_i$, instead of the correct pre-image t_i of h_i. In the following, when clear from the context, we use term "tuple" for a table tuple $t_i \in T$ and its hash value $h_i = H(t_i)$ interchangeably.

We illustrate the main idea of our framework using Figure 4·2. Let $h_1, \ldots, h_n$ correspond to the hash values of the tuples $t_1, \ldots, t_n$ of T, respectively. We maintain a copy of these values for every attribute a_i, and *sort* the copy of a_i according to the values of the tuples on a_i. For instance, in Figure 4·2, $h_3 = H(t_3)$ appears first in the ordering of a_1 because t_3 has the smallest value on attribute a_1 among the tuples in T.

A multi-dimensional range query Q is defined over an *arbitrary* set of dimensions (where, recall, each dimension is an attribute). Our framework "decomposes" a d-

dimensional range query into d separate 1-dimensional queries. More specifically, our framework boils down to two steps:

- Step 1: **(1-D proofs)** For each dimension a_i involved in Q, compute the set R_i of all hash values of tuples that satisfy the condition on a_i. Formally, $R_i = \{h_j = H(t_j) \mid l_i \leq t_j.a_i \leq u_i\}$. Also compute proof π_{R_i} for the integrity of R_i.

- Step 2: **(Combination)** Compute the result $R \overset{\text{def}}{=} \bigcap_i R_i$ and proof π for its integrity, given pairs (R_i, π_{R_i}) for every a_i involved in Q.

For example, suppose in Figure 4·2 that a 2-dimensional query Q is defined over a_i and a_j. Our two-step framework first dictates the computation of $R_i = \{h_1, h_3, h_7\}$ that corresponds to the 1-dimensional result along dimension a_i, and $R_j = \{h_1, h_7\}$ along dimension a_j, as well as proofs π_{R_i}, π_{R_j}. It next requires the computation of result $R = \{h_1, h_7\}$ and a proof π.

Benefits. Our view of multi-dimensional range queries as a collection of 1-dimensional range queries offers multiple advantages over existing approaches: (i) We aim to support range queries over *any combination* of attributes. Thus, there are $O(2^m)$ possible different attribute combinations that can be involved in a query, where m is the total number of attributes in T. In order to support all of them, existing solutions must build $O(2^m)$ *separate* authenticated structures. On the contrary, our framework requires m such structures (one per attribute) constructed *once*, which suffice to capture *all* $O(2^m)$ possible subsets of attributes. (ii) As discussed in Section 5.1, the performance of all existing constructions deteriorates drastically with d. In contrast, the separate handling of each dimension allows our framework to scale with d gracefully. (iii) To address scalability issues that arise from the advent of "big data", data management systems typically employ multi-core CPU hardware, as well as cloud infrastructures involving multiple nodes. In our framework, the 1-dimensional sub-queries can be distributed across multiple cores/nodes, and run in parallel. The combination step can then take place using well-known in-network aggregation techniques (e.g., in a MapReduce fashion [DG08]).

The challenge. There are several efficient solutions for the problem of 1-dimensional range queries (e.g., [MND$^+$04, LHKR06]), each of which can be used to instantiate Step 1 in our framework. The problem lies in Step 2, i.e., how to efficiently combine the separate proofs. In particular, for all known 1-dimensional solutions, Step 2 entails creating the proof π as the concatenation of all proofs π_{R_i} *and* the partial results R_i. This makes the proof size as large as the sum of the partial result cardinalities, which can be substantially larger than the final result R. In turn, this may lead to a prohibitive communication and verification cost for the client.

A fundamental requirement of our framework is the partial proofs π_{R_i} produced in Step 1 to be efficiently combinable to a short proof π in Step 2, whose size is independent of $\sum_i |R_i|$. More formally:

Efficiency 1. *A $\mathcal{AMR}$ following our framework is* efficient*, if it outputs proofs π of size $o(\sum_i |R_i|)$.*

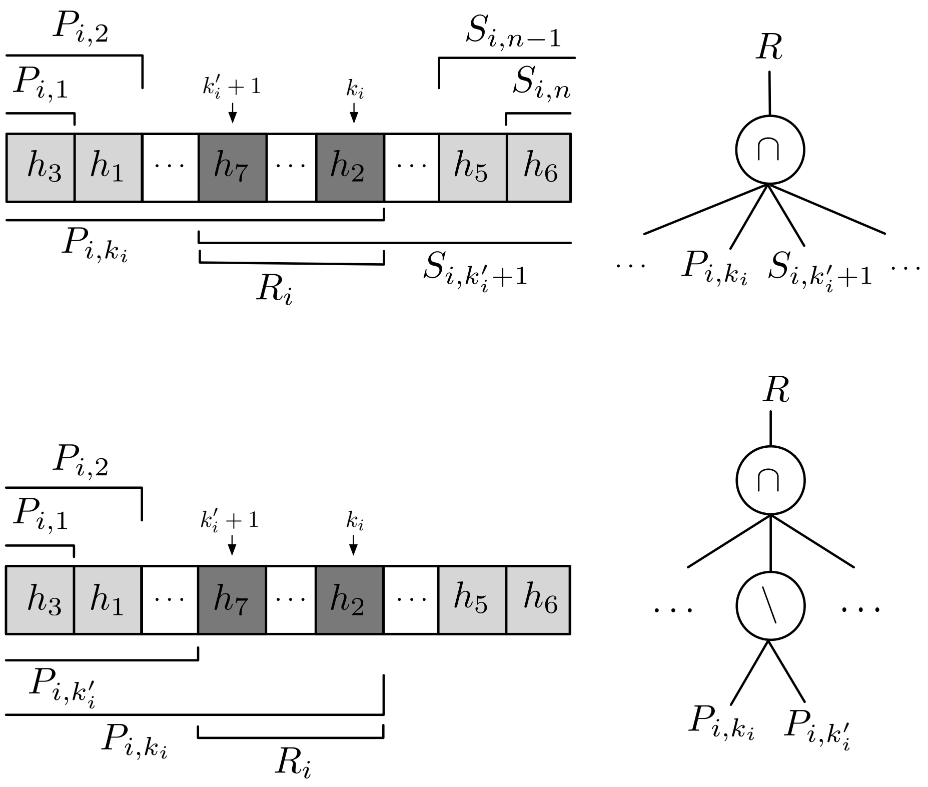
Based on our observation above, any existing 1-dimensional solution trivially conforms to our framework. However, *no* such solution satisfies the efficiency requirement. Essentially, the efficiency requirement motivates the design of $\mathcal{AMR}$'s with non-trivial proof combination techniques. What has prevented the research community from devising such $\mathcal{AMR}$'s is the combination of the lack of appropriate cryptographic tools, and the reduced need for range queries over arbitrarily many dimensions, and large quantities of data. However, the emergence of big data practices renders the problem timely and important, whereas the recent introduction of $\mathcal{SOA}$ techniques opens new directions towards efficient solutions.

4.4.2 Construction

We first outline the main idea of our scheme and elaborate on some important implementation decisions. Subsequently, we present the instantiation of our algorithms.

Main idea. Recall that Step 2 of our framework dictates that the result R is expressed as the *intersection* of *sets* R_i. We stress that $\mathcal{SOA}$ techniques appear to solve our targeted problem trivially as follows: The owner pre-computes a proof component $\pi_{R_i} = acc(R_i)$ for each R_i, where $acc(R_i)$ is the accumulation value of set R_i, and signs each π_{R_i}. According to our discussion in Section 4.2, given all R_i, π_{R_i}, the server computes and sends to the client a combined intersection proof $\pi_\cap$ for the integrity of R, along with all π_{R_i} and their corresponding signatures. Observe that this approach satisfies our efficiency requirement. However, there exist $O(n^2)$ possible R_i sets per dimension that can be involved in a query, which makes the pre-processing cost for the owner and the storage overhead for the server prohibitive. The main idea behind our scheme is to express *any possible* R_i as the result of an operation over a *fixed* number of "primitive sets", given the constraint that there are $O(n)$ such "primitive sets".

One possible way to derive R_i from "primitive sets" is illustrated in Figure 4·3 (top). Let us focus on a_i and the ordering of the hash values h_j (of tuples t_j) according to the $t_j.a_i$ values. We define the *prefix set* $P_{i,j}$ to consist of all hash values appearing in positions $1, \dots, j$ in the ordering. In the figure, $P_{i,1} = \{h_3\}$ and $P_{i,2} = \{h_3, h_1\}$. Similarly, we define *suffix set* $S_{i,j}$ to consist of all hash values appearing in positions $n-j+1, \dots, n$ in the ordering. In our example, $S_{i,1} = \{h_6\}$ and $S_{i,2} = \{h_5, h_6\}$. Now assume that $k'_i + 1, k_i$ are the two positions in this ordering corresponding to the first and last tuple satisfying the query on a_i. Observe that, in this case, $R_i = P_{i,k_i} \cap S_{i,k'_i+1}$, and, thus, there exist $2n$ "primitive sets" per dimension, i.e., all prefix and suffix sets. Let $\pi_{P_{i,k_i}} = acc(P_{i,k_i})$ and $\pi_{S_{i,k'_i+1}} = acc(S_{i,k'_i+1})$. Then,



$P_{i,2}$
$P_{i,1}$
$k_i'+1$
k_i
$S_{i,n-1}$
$S_{i,n}$
h_3 h_1 $\cdots$ h_7 $\cdots$ h_2 $\cdots$ h_5 h_6
P_{i,k_i}
$S_{i,k_i'+1}$
R_i
R
$\cap$
$\cdots$ P_{i,k_i} $S_{i,k_i'+1}$ $\cdots$
$P_{i,2}$
$P_{i,1}$
$k_i'+1$
k_i
h_3 h_1 $\cdots$ h_7 $\cdots$ h_2 $\cdots$ h_5 h_6
$P_{i,k_i'}$
P_{i,k_i}
R_i
R
$\cap$
$\cdots$ $\setminus$ $\cdots$
P_{i,k_i} $P_{i,k_i'}$

Figure 4·3: Set representation of R_i using two different techniques.

since $R = \bigcap_i R_i = \bigcap_i (P_{i,k_i} \cap S_{i,k_i'+1})$ can be computed with a single set intersection, we can utilize the $\mathcal{SOA}$ of [PTT11] to create a proof π (consisting of $\pi_\cap$ and signatures on every $\pi_{P_{i,k_i}}, \pi_{S_{i,k_i'+1}}$) for the integrity of R, while satisfying both efficiency and $O(n)$ pre-processing/storage.

Unfortunately, from the complexity analysis of [PTT11] in Section 4.2, it follows that $\pi_\cap$ requires $\tilde{O}(d \cdot n)$ time for each query at the server, which makes this approach impractical. The reason is that the $\pi_\cap$ construction overhead is dictated by the cardinality of the input sets, which is $|P_{i,k_i}| + |S_{i,k_i'+1}| \in \Omega(n)$, along each attribute.

Motivated by the above, we propose an alternative solution, which we demonstrate using Figure 4·3 (bottom). We define sets R_i through *set difference.* In particular, using the notation of the previous paragraph, it holds that $R_i = P_{i,k_i} \setminus P_{i,k'_i}$. Consequently, in this case the "primitive sets" are only the n prefix sets. Now $R = \bigcap_i R_i = \bigcap_i (P_{i,k_i} \setminus P_{i,k'_i})$ is no longer expressed as a single set operation. The only known $\mathcal{SOA}$ that can accommodate a circuit of set operations is [CPPT14]. Briefly stated, [CPPT14] allows the construction of π_{R_i} with $O(|R_i|)$, and $\pi_\cap$ with $O(\sum_i |R_i|)$, exponentiations. The downside is that its security relies on non-standard cryptographic assumptions. To circumvent this, we construct our own sub-protocol for producing combinable proofs of set difference, customized for the special case where the first participating set is a *strict superset* of the second. This particular constraint enables our sub-protocol to prove the validity of $R_i = P_{i,k_i} \setminus P_{i,k'_i}$ with $O(|R_i|)$ exponentiations, while being secure under a standard cryptographic assumption.

Our algorithms are comprised of a collection of set operation sub-protocols, bundled with a set membership scheme. For clarity of presentation, we will abstract the internal mechanics of these sub-protocols, and instead use the following conventions:

- By $\mathcal{SMA}$ we refer to either a Merkle tree [Mer89] or an accumulation tree [PTT08], along with all its algorithms.
- By ProveIntersection, VerifyIntersection, we refer to the corresponding algorithms of the $\mathcal{SOA}$ of [PTT11]. The former computes an intersection proof on its input sets, and the latter verifies this operation.
- By ProveSetDiff, VerifySetDiff, we refer to the corresponding algorithms of our set difference construction. The former generates a set difference proof, and the latter verifies this operation.

This presentation choice also highlights that our algorithms use elementary cryptographic tools as building blocks. Therefore, the overall performance of our scheme

Algorithm Setup(T, pk, sk)

```
1.  For $j = 1, ..., n$, compute $h_j = H(t_j)$
2.  For $i = 1, ..., m$
3.      Sort $h_j$ in ascending order along $t_j.a_i$
4.      For $j = 1, ..., n$
5.          Compute $\pi_{P_{i,j}} = acc(P_{i,j})$
6.          Let $v_{i,j}$ be the $j^{\text{th}}$ largest value on attribute $a_i$ in $T$
7.          Construct triplet $\tau_{P_{i,j}} = (v_{i,j}, v_{i,j+1}, \pi_{P_{i,j}})$
8.      Build $\mathcal{SMA}_i$ with digest $\delta_i$ over $\tau_{P_{i,1}}, ..., \tau_{P_{i,n}}$
9.  Build $\mathcal{SMA}$ with digest $\delta$ over $(1, \delta_1), ..., (m, \delta_m)$
10. Send $T, auth(T) = (\mathcal{SMA}, \mathcal{SMA}_1, ..., \mathcal{SMA}_m)$ to the server
11. Publish $pk, \delta$
```

is highly dependent on that of the underlying tools, leaving potential for great improvement as novel tool instantiations are introduced in the literature.

Key generation. It outputs key pair pk, sk, which are simply the public and secret keys of the underlying $\mathcal{SMA}$ and $\mathcal{SOA}$ schemes, generated by their corresponding key generation routines.

Setup. Figure 4·4 visualizes the detailed authentication structure produced by the setup algorithm, whose pseudo code is shown above. The owner computes the hash value $h = H(t)$ for every $t \in T$ (Line 1). It then produces the sorted orderings of the hash values along every attribute (Lines 2-3), and computes the prefix sets $P_{i,j}$ as explained in Figure 4·3. Next, it calculates prefix proof $\pi_{P_{i,j}}$ for each $P_{i,j}$ (Lines 4-5). For each $P_{i,j}$, it computes a triplet $\tau_{P_{i,j}} = (v_{i,j}, v_{v,j+1}, \pi_{P_{i,j}})$ in Lines 6-7. Values $v_{i,j}, v_{i,j+1}$ indicate the j^{th} and $(j+1)^{\text{th}}$ largest values on attribute a_i appearing in T. These values are necessary for guaranteeing the completeness of the result, and their purpose will become clear soon. Subsequently, it computes an $\mathcal{SMA}_i$ over the triples $\tau_{P_{i,j}}$ of every attribute a_i, producing digests $\delta_1, \ldots, \delta_m$ (Line 8). It then constructs a $\mathcal{SMA}$ over the (i, δ_i) pairs, and generates digest δ (Line 9). Finally, it sends the $m + 1$ $\mathcal{SMA}$ structures to the server along with T (Line 10), and publishes pk, δ

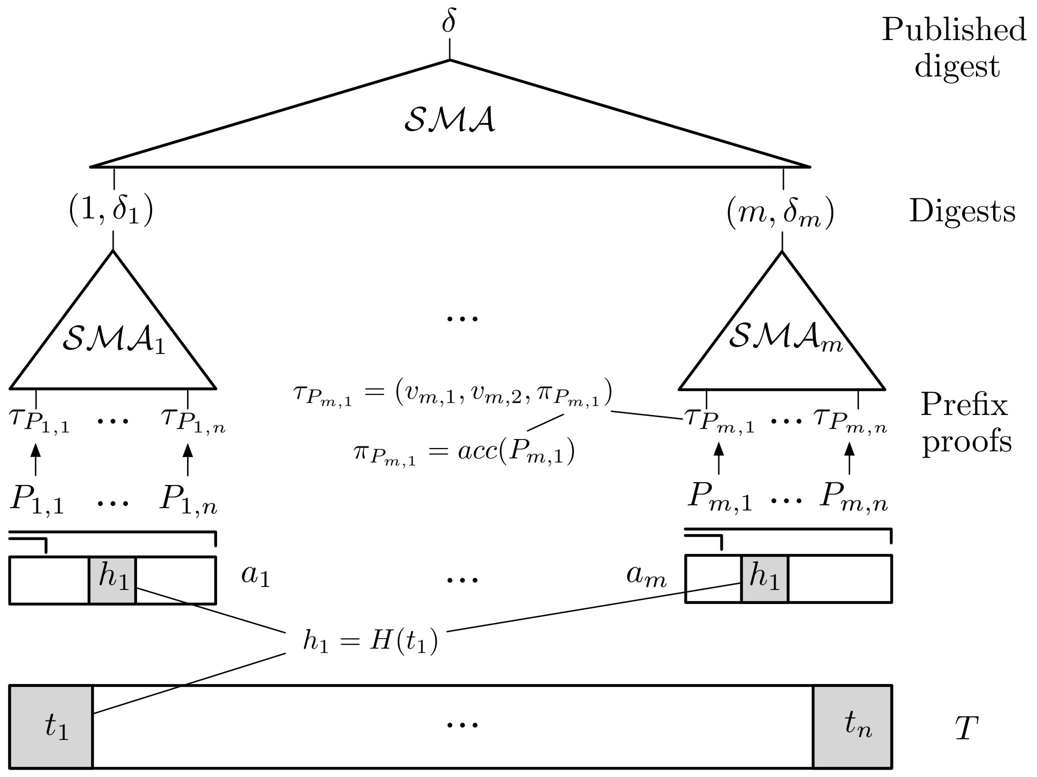


Figure 4·4: The authentication structure of our basic scheme

(Line 11).

Proof construction. For ease of presentation and without loss of generality, we assume that the requested query is upon the d first attributes of T and, hence, encode it as $Q = \{i, l_i, u_i)$ for $i = 1, ..., d$. We provide the pseudo code of this algorithm below.

Given R, the server first computes π_R (Line 1), and calculates set R_i for each attribute a_i. It identifies prefixes P_{i,k_i}, P_{i,k'_i} such that $R_i = P_{i,k_i} \setminus P_{i,k'_i}$ (as in Figure 4·3), and locates the corresponding triplets $\tau_{P_{i,k_i}}, \tau_{P_{i,k'_i}}$ (Lines 4-5). Subsequently, it constructs the $\mathcal{SMA}_i$ proofs for $\tau_{P_{i,k_i}}, \tau_{P_{i,k'_i}}$ (Line 6) and $\mathcal{SMA}$ proofs for (i, δ_i), for $i = 1, \ldots, d$ (Line 7). It then invokes subroutines ProveSetDiff and ProveIntersection as defined in our main idea paragraph, and produces proofs $\pi_{R_1}, \ldots, \pi_{R_d}$ and $\pi_\cap$

$$\begin{tikzcd}
 & R & & \text{Level 5} \\
 & \pi_R \arrow[u] & & \text{Level 4} \\
\pi_{R_1} \arrow[ur, "\pi_\cap"] & \cdots & \pi_{R_d} \arrow[ul, "\pi_\cap"'] & \text{Level 3} \\
(\tau_{P_{1,k_1}}, \tau_{P_{1,k'_1}}) \arrow[u, "{(1, l_1, u_1)}"'] & \cdots & (\tau_{P_{d,k_d}}, \tau_{P_{d,k'_d}}) \arrow[u, "{(d, l_d, u_d)}"] & \text{Level 2} \\
(1, \delta_1) \arrow[u, "\mathcal{SMA}_1 \text{ proof}"] & \cdots & (d, \delta_d) \arrow[u, "\mathcal{SMA}_d \text{ proof}"'] & \text{Level 1} \\
 & \delta \arrow[ul, "\mathcal{SMA} \text{ proof } 1"] \arrow[ur, "\mathcal{SMA} \text{ proof } d"'] & & \text{Level 0}
\end{tikzcd}$$

Figure 4·5: Authentication flow

(Lines 8-9), respectively. Finally, it puts together all proof components into a single proof π (Line 10), and sends it to the client along with result R (Line 11). We thoroughly describe the functionality of every proof component in π in the next paragraph.

Verification. We visualize the intuition in Figure 4·5, which depicts the authentication flow among the various proof components of the final proof π sent to the client. Specifically, if a component authenticates another, we draw an arrow from the former to the latter. Arrow labels represent information serving as "glue" between the components. The goal is to verify result R (top of the figure), but the only trusted information (in addition to *pk*) is δ (bottom of the figure). Verification proceeds bottom-up from level 0 to 5, maintaining the invariant that, at level ℓ, the server must have computed the components therein truthfully with respect to T and Q.

At level 0, δ is signed/published by the owner and, thus, it is trusted. At level 1, given the d $\mathcal{SMA}$ proofs and δ, we verify the integrity of components (i, δ_i). Likewise, at level 2, given the $2d$ $\mathcal{SMA}_i$ proofs along with (i, δ), we verify the integrity of $2d$ triplets $(\tau_{P_{i,k_i}}, \tau_{P_{i,k'_i}})$. Here, we reach a critical point in the verification process. We must prove that these particular $(\tau_{P_{i,k_i}}, \tau_{P_{i,k'_i}})$ correspond to the triplets for sets P_{i,k_i}, P_{i,k'_i}, such that $P_{i,k_i} \setminus P_{i,k'_i} = R_i$, where R_i is the truthful result of Q on dimension a_i. To do this, we parse Q as $(i, l_i, u_i)_{i=1}^{d}$ and check $v_{i,k'_i} < l_i \leq v_{i,k'_i+1}$ and $v_{i,k_i} \leq u_i < v_{i,k_i+1}$, where $v_{i,k_i}, v_{i,k_i+1}, v_{i,k'_i}, v_{i,k'_i+1}$ are included in $\tau_{P_{i,k_i}}, \tau_{P_{i,k'_i}}$. This guarantees that P_{i,k_i} is the smallest prefix set that contains the entire R_i, and P_{i,k'_i} is the largest prefix set that does not intersect R_i. Therefore, we verify that $(\tau_{P_{i,k_i}}, \tau_{P_{i,k'_i}})$ indeed correspond to the correct P_{i,k_i}, P_{i,k'_i}. Next, we retrieve $\pi_{P_{i,k_i}}, \pi_{P_{i,k'_i}}$ from $(\tau_{P_{i,k_i}}, \tau_{P_{i,k'_i}})$, respectively, and run routine VerifySetDiff to validate the truthfulness of π_{R_i} as the accumulation value of set R_i at level 3. Next, at level 4 we verify that π_R is the accumulation value of $\bigcap_i R_i$ with VerifyIntersection, using π_R, all π_{R_i}, and $\pi_\cap$. Observe that, at this point we know that π_R corresponds to the accumulation of the correct result of Q on T. At the last level 5, we verify that R is indeed this correct result by checking if $acc(R) = \pi_R$. We summarize this verification process in the pseudocode below.

Updates. We focus on an insertion of a single tuple t (the case of deletions is similar). The process is easier to follow by revisiting Figure 4·4. The owner first computes $h = H(t)$. It then inserts h in the appropriate position in the ordering of each attribute a_i, and properly updates all the prefix sets it affects. Note that, if h is placed in position j for attribute a_i, the owner must change sets $P_{i,j'}$ for all $j' \geq j$, and modify their corresponding proofs $\pi_{P_{i,j'}}$ in $\tau_{P_{i,j'}}$. Furthermore, it must create a new $\tau_{P_{i,j}}$, and alter $v_{i,j}$ in $\tau_{P_{i,j-1}}$ (where it appears as the second element). Finally, it must propagate the changes of all τ triplets in all $\mathcal{SMA}_i$ and $\mathcal{SMA}$. Admittedly,

```
Algorithm $\mathsf{Prove}(Q, R, pk, auth(T))$

1. Compute $\pi_R = acc(R)$
2. For $i = 1, ..., d$
3.     Compute $R_i$
4.     Identify $P_{i,k}$ and locate $\tau_{P_{i,k}} = (v_{i,k_i}, v_{i,k_i+1}, \pi_{P_{i,k_i}})$
5.     Identify $P_{i,k'}$ and locate $\tau_{P_{i,k'}} = (v_{i,k'_i}, v_{i,k'_i+1}, \pi_{P_{i,k'_i}})$
6.     Compute $\mathcal{SMA}_i$ proofs for $\tau_{P_{i,k}}, \tau_{P_{i,k'}}$
7.     Compute $\mathcal{SMA}$ proof for $(i, \delta_i)$
8.     $\pi_{R_i} = \mathsf{ProveSetDiff}(R_i, pk)$
9. $\pi_\cap = \mathsf{ProveIntersection}(R, R_1, ..., R_d, \pi_R, \pi_{R_1}, ..., \pi_{R_d}, pk)$
10. Set $\pi = (\pi_R, \pi_\cap, (\pi_{R_i}, \tau_{P_{i,k}}, \tau_{P_{i,k'}}, \delta_i)_{i=1}^d$, all $\mathcal{SMA}$ proofs)
11. Send $\pi, R$ to the client
```

```
Algorithm $\mathsf{Verify}(Q, R, \pi, pk, \delta)$

1. Parse $Q$ as $(i, l_i, u_i)_{i=1}^d$
2. For $i = 1, ..., d$
3.     Verify $\delta_i$ with respect to $\delta$ with $\mathcal{SMA}$ proof
4.     Verify $\tau_{P_{i,k}}, \tau_{P_{i,k'}}$ with respect to $\delta_i$ with $\mathcal{SMA}_i$ proofs
5.     Verify $v_{i,k'_i} < l_i \leq v_{i,k'_i+1}$ and $v_{i,k_i} \leq u_i < v_{i,k_i+1}$
6.     Run $\mathsf{VerifySetDiff}(\pi_{P_{i,k}}, \pi_{P_{i,k'}}, \pi_{R_i}, pk)$
7. Run $\mathsf{VerifyIntersection}(\pi_R, \pi_{R_1}, ..., \pi_{R_d}, \pi_\cap, pk)$
8. Compute $acc(R)$ and verify $acc(R) = \pi_R$
9. If verification in Lines 3-8 fails, return reject, else return accept
```

the update process in this basic scheme can be quite expensive; in fact, it can be as costly as re-running the setup stage. Due to this, we define algorithm Update for this construction, as a simple call to Setup. In Section 4.5, we introduce a solution that supports efficient updates, while maintaining all other asymptotic costs.

A set difference sub-protocol. Before we state our main result regarding our basic scheme, we present a sub-protocol for proving the correctness of a set difference operation between two sets X_1, X_2, *under the constraint that the first is a proper superset of the second.* This constraint renders our sub-protocol conceptually simple and very efficient. It consists of two routines ProveSetDiff and VerifySetDiff. The former takes as input set $X_1 \setminus X_2$ and outputs a proof for its validity as the set difference of X_1, X_2. The latter receives succinct representations $\pi_{X_1}, \pi_{X_2}, \pi_{X_1 \setminus X_2}$ of $X_1, X_2, X_1 \setminus X_2$, respectively, and returns `accept` if $X_1 \setminus X_2$ is the set difference of X_1, X_2, and `reject` otherwise. Below is the pseudo codes of the two routines.

Note that these routines are meaningful only as part of a more elaborate $\mathcal{SOA}$ scheme (e.g., [PTT11, CPPT14]), which utilizes bilinear accumulators as well, and relies on the same public key pk. More specifically, the caller $\mathcal{SOA}$ is enforced with the computation of input $X_1 \setminus X_2$ to $\mathsf{ProveSetDiff}$. Therefore, this routine simply returns $\pi_\setminus$ as the accumulation value of $X_1 \setminus X_2$ in time $\tilde{O}(|X_1 \setminus X_2|)$. In addition, the $\mathcal{SOA}$ must first check that inputs π_{X_1}, π_{X_2} of $\mathsf{VerifySetDiff}$ are the accumulation values of X_1, X_2, such that X_1 is a proper superset of X_2, prior to calling the routine. In this case, the cost of $\mathsf{VerifySetDiff}$ is $O(1)$ pairings.

For example, in our scheme in Section 4.4.2, $\mathsf{ProveSetDiff}$ is called in algorithm Prove for each set R_i, *after* R_i has been computed. Moreover, $\mathsf{VerifySetDiff}$ is invoked in Verify using as inputs the *already verified* accumulation values of prefix sets P_{i,k_i}, P_{i,k'_i} that, by definition, satisfy the constraint $P_{i,k_i} \supset P_{i,k'_i}$. The following lemma is useful in our proofs.

Lemma 7. *Let λ be a security parameter, $pub \leftarrow \mathsf{GenBilinear}(1^\lambda)$, and elements $(g, g^s, ..., g^{s^q}) \in \mathbb{G}$, computed for some s chosen at random from $\mathbb{Z}_p^*$. Let X_1, X_2 be sets with elements in $\mathbb{Z}_p$, such that $X_1 \supset X_2$. For an element $y \in \mathbb{G}$, it holds that $y = acc(X_1 \setminus X_2)$, iff $e(acc(X_2), y) = e(acc(X_1), g)$.*

Proof: ($\Leftarrow$) Let $X_1 = \{x_1, ..., x_{l'}\}$ and $X_2 = \{x_1, ..., x_l\}$ for $l, l' \in \mathbb{N}$ with $l < l'$. If $e(acc(X_2), y) = e(acc(X_1), g)$, then we have $e(g^{\prod_{i=1}^{l}(x_i+s)}, y) = e(g^{\prod_{i=1}^{l'}(x_i+s)}, g)$. Hence, it holds $e(y, g) = e(g^{\prod_{i=l+1}^{l'}(x_i+s)}, g) \overset{\text{def}}{=} e(acc(X_1 \setminus X_2), g)$ which implies that $y = acc(X_1 \setminus X_2)$, since $e(g, g)$ is a generator of $\mathbb{G}_T$.

($\Rightarrow$) If $y = acc(X_1 \setminus X_2) = g^{\prod_{i=l+1}^{l'}(x_i+s)}$, then it holds that $e(g^{\prod_{i=1}^{l}(x_i+s)}, y) = e(g^{\prod_{i=1}^{l}(x_i+s)}, g^{\prod_{i=l+1}^{l'}(x_i+s)}) = e(g^{\prod_{i=1}^{l'}(x_i+s)}, g) = e(acc(X_1), g)$. □

```
Algorithm ProveSetDiff($X_1 \setminus X_2, pk$)
1. Return $\pi_\setminus = acc(X_1 \setminus X_2)$

Algorithm VerifySetDiff($\pi_{X_1}, \pi_{X_2}, \pi_\setminus, pk$)
1. If $e(\pi_{X_2}, \pi_\setminus) = e(\pi_{X_1}, g)$, return accept, else return reject
```

We can now state the following result.

Theorem 3. *The scheme* $\{\mathsf{KeyGen}, \mathsf{Setup}, \mathsf{Update}, \mathsf{Prove}, \mathsf{Verify}\}$ *is a* correct*,* efficient*, and* secure $\mathcal{AMR}$ *under the q-**SBDH** assumption.*

Proof: The correctness of our scheme results from the semantics of the proof generation and verification and by close inspection of the algorithms. Moreover, since the proof size is either $O(d)$ or $O(d \log n)$, our construction satisfies the efficiency requirement.

Let us assume there exists poly-size adversary $\mathcal{A}$ that wins the $\mathcal{AMR}$ security game with non-negligible probability. Also, let Q, R^*, π^*, j be the cheating tuple output by $\mathcal{A}$ and let T denote the data structure's state at index j, and $auth(T), \delta$ the corresponding authentication information and digest . In the following, we annotate by $*$ any element of π^*. Moreover, if an event is denoted by $\mathcal{E}$, then its complement is denoted by $\mathcal{E}'$. Consider the following events:

$\mathcal{E}_1$**:** $\mathcal{A}$ wins the $\mathcal{AMR}$ game.

$\mathcal{E}_2$**:** π^* contains a tuple τ^* or $(i, \delta_i)^* \notin auth(T)$.

Recall that $auth(T)$ consists of $m+1$ $\mathcal{SMA}$ structures; each of the m first is built over n tuples τ containing sequential values and prefix accumulations for attribute a_i, and the last is built over m pairs of the form (i, δ_i), i.e., containing the attribute index and corresponding digest. Note that, in the $\mathcal{AMR}$ game the values $auth(T), d$ are computed correctly by the challenger for T.

By the law of total probability, we have:

$$\begin{aligned}\Pr[\mathcal{E}_1] &= \Pr[\mathcal{E}_2]\Pr[\mathcal{E}_1|\mathcal{E}_2] + \Pr[\mathcal{E}_2']\Pr[\mathcal{E}_1|\mathcal{E}_2'] \\ &\leq \Pr[\mathcal{E}_1|\mathcal{E}_2]] + \Pr[\mathcal{E}_1|\mathcal{E}_2']\ .\end{aligned}$$

Intuitively, the first term in the right hand of the above relation corresponds to an adversary that wins by breaking the security of the underlying $\mathcal{SMA}$ and the second term with breaking the q-SBDH.

Claim 4. $\Pr[\mathcal{E}_1|\mathcal{E}_2]$ *is negligible in* λ*.*

Proof: Let us assume it is non-negligible in λ. Without loss of generality, we will assume that the non-existing tuple is of the form τ^*, i.e., it should fall under some of the first m $\mathcal{SMA}$ structures, e.g., $\mathcal{SMA}_i$. Since $\mathcal{A}$ wins, it follows that the $\mathcal{AMR}$ verification succeeds however $\tau^* \notin \mathcal{SMA}_i$. We now distinguish between the chosen $\mathcal{SMA}$ instantiation:

- **Merkle tree.** We will construct adversary $\mathcal{A}'$ that finds a collision in the CRHF H used to implement the Merkle tree as follows. $\mathcal{A}'$ runs $\mathsf{BilGen}(1^\lambda)$ to compute bilinear parameters pub, chooses $s \leftarrow_R \mathbb{Z}_p^*$ and $q \in poly(\lambda)$ and computes values $g^s, \ldots, g^{s^q}$. Finally, he runs $\mathcal{A}$ on input $(pub, g^s, \ldots, g^{s^q})$. He then proceeds to provide oracle access for all the $\mathcal{AMR}$ algorithms. After the setup and each update call from $\mathcal{A}$, database T_η for $\eta = 0, \ldots, j$ is produced and $\mathcal{A}'$ stores all triplets $(T_\eta, auth(T_\eta), \delta_\eta)$. When $\mathcal{A}$ outputs his challenge tuple for index j, $\mathcal{A}'$ parses π^*, checking for each tuple whether it appears in $auth(T_j)$. If any of them does not appear in $auth(T_j)$, there must exist triplet τ in the corresponding $\mathcal{SMA} \in auth(T_j)$ such that $\tau \neq \tau^*$ and $H(\tau) = H(\tau^*)$ (for the challenge sample key of H). This holds since the verification process for τ^* under a Merkle tree in $auth(T_j)$ succeeds, yet τ^* is not in the tree. By assumption this will happen with non-negligible probability, hence $\mathcal{A}'$ breaks the collision resistance of H, and the claim follows.

- **Accumulation tree.** The reduction proceeds in the same manner as in the previous case. The difference is that $\mathcal{A}'$ is now playing against an accumulation tree challenger, he receives as input a public key that coincides perfectly with the $\mathcal{AMR}$ game and he does not need to issue any queries to his challenger before he sees the challenge tuple by $\mathcal{A}$, since everything can be computed with access to the public key only (this follows from the properties of the bilinear accumulator used to build the tree). After $\mathcal{A}$ issues his challenge, $\mathcal{A}'$ constructs the tree $\mathcal{SMA}_i$ by issuing consecutive update queries to his challenger. Finally, he outputs τ^* and the part of π^* that corresponds to proving (the false statement) that $\tau^* \in \mathcal{SMA}_i$. By Lemma 4 this can only happen with negligible probability, which contradicts our original assumption, and the claim follows.

□

Now we prove that the second term of the inequality, namely $\Pr[\mathcal{E}_1|\mathcal{E}_2']$, is negligible, by contradiction. Assume that $\Pr[\mathcal{E}_1|\mathcal{E}_2']$ is non-negligible. Since $\mathcal{E}_2$ does not happen, all triplets τ^* and pairs $(1, \delta_1), \ldots, (m, \delta_m)$ in π^* appear in $auth(T)$.

This immediately implies that the two values $v_{i,l}^*, v_{i,l+1}^*$ in each triplet are consecutive along their dimension and each digest matches its corresponding dimension. By construction, along each dimension there exist *exactly two distinct* τ^*, τ'^* for which verification of Q succeeds; one corresponds to the lower bound of the 1-dimensional range of the query (l_i) and one for the upper (u_i). Furthermore, if a triplet correctly formed for $\mathcal{SMA}_i^*$ of attribute a_i, is used as part of the proof of an $\mathcal{SMA}_j^*$ corresponding to $a_j \neq a_i$, then it can be used to break the $\mathcal{SMA}$ security as shown in the proof of Claim 4, which can only happen with negligible probability.

From the above, it follows that, for all i, i', the triplet $\tau^*_{i,i'} \in \pi^*$ in dimension a_i contains the accumulation value of the correctly computed prefix set $P_{i,i'}$ with all but negligible probability. Assuming this holds, by Lemma 7 and because verification succeeds, it follows that $\pi^*_{R_i} \in \pi^*$ is the accumulation value of the correctly computed set R_i for query Q on T. Therefore, the values $W^*_i, F^*_i \in \pi^*_\cap$, along with sets R_i and cheating answer $R^* \neq R(Q,T)$ as output by $\mathcal{A}$, contradict Lemma 6, breaking the q-SBDH assumption. Therefore, the probability $\Pr[\mathcal{E}_1|\mathcal{E}'_2]$ must be negligible.

Since $\Pr[\mathcal{E}_1|\mathcal{E}_2] + \Pr[\mathcal{E}_1|\mathcal{E}'_2]$ is negligible, $\Pr[\mathcal{E}_1]$ must be negligible as well, contradicting our original assumption that there exists poly-size adversary $\mathcal{A}$ that breaks our scheme with non-negligible probability. $\square$

4.5 Update-efficient scheme

Here we present the necessary modifications to make our scheme handle updates in an efficient way. As a building block for this, we also include a set union authentication sub-protocol, for the case of union of disjoint sets.

4.5.1 Construction

Similar to our solution above, the update-efficient scheme views the query result as a combination of "primitive set" operations. It then allows the server to compute a small set of proof elements, which can be aggregated by the client in a bottom-up fashion (similar to Figure 4·5). It adopts the same idea of computing proofs for the partial R_i results along each dimension a_i, and then combining them through a set intersection protocol into a single proof that verifies the final result R. It also adopts the idea of performing set difference operations over prefix sets. The primary difference with the basic scheme is that we now organize the hash values in the ordering of each dimension into *buckets*, and compute prefix sets over both the buckets, as well as the hashes in each bucket. As we shall see, this twist *isolates* the effect of an update, thus, reducing the update cost complexity. It also mandates small modifications of the overall authentication structure, proof generation, and verification processes, and creates the need for a new *set union sub-protocol*. In the following, we describe the main ideas behind the construction.

Figure 4·6 depicts the authentication structure created by the owner during the setup stage, focusing on attribute a_i. As before, the owner sorts the hash values of the n tuples of T in ascending order of the a_i values of the tuples. It then creates b buckets, enumerated as $B_{i,1}, \ldots, B_{i,b}$ (bottom left in the figure). For clarity of presentation, we assume that the partitioning of hashes into buckets is publicly known (e.g., each bucket may correspond to a specific range of the domain of a_i), and that each bucket has n/b hashes.

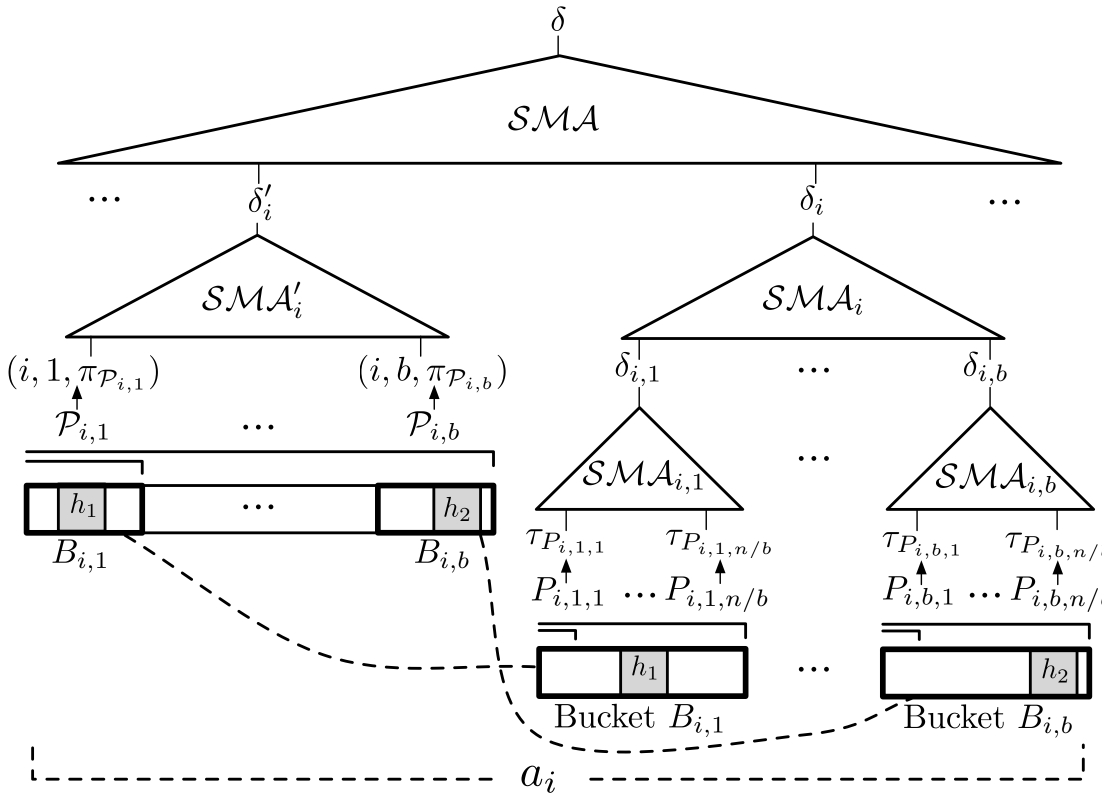


Figure 4·6: The authentication structure for our dynamic scheme

We define as $\mathcal{P}_{i,j}$ the prefix set over buckets $B_{i,1}, \ldots, B_{i,j}$, i.e., the set of hashes included in $B_{i,1}, \ldots, B_{i,j}$ (we use calligraphic $\mathcal{P}$ for bucket prefixes to distinguish them from hash prefixes denoted by P). The owner computes a proof $\pi_{\mathcal{P}_{i,j}} = acc(\mathcal{P}_{i,j})$ for every $\mathcal{P}_{i,j}$. In addition, for every bucket $B_{i,j}$, it computes prefixes $P_{i,j,l}$ for the hashes therein (bottom right in the figure), as well as proofs $\pi_{P_{i,j,l}} = acc(P_{i,j,l})$. Subsequently, the owner creates a triplet $(i, j, \pi_{\mathcal{P}_{i,j}})$ for every $\mathcal{P}_{i,j}$, as well as tuple $\tau_{P_{i,j,l}}$ for every $P_{i,j,l}$. Note that $\tau_{P_{i,j,l}}$ is similar to the case of the basic scheme (i.e., it encompasses $\pi_{P_{i,j,l}}$ along with two a_i values), but now also incorporates the index j of the bucket. The owner feeds $(i, j, \pi_{\mathcal{P}_{i,j}})$ to the leaf level of $\mathcal{SMA}'_i$ with digest δ'_i. It also feeds $\tau_{P_{i,j,l}}$ to $\mathcal{SMA}_{i,j}$ with digest $\delta_{i,j}$. It then superimposes another $\mathcal{SMA}_i$ over digests $\delta_{i,j}$ which has digest δ_i. Finally, it builds $\mathcal{SMA}$ over all δ'_i, δ_i with final digest δ that is published. The various $\mathcal{SMA}$'s will later allow the server to construct proofs validating that $\pi_{\mathcal{P}_{i,j}}, \pi_{P_{i,j,l}}$ were indeed computed by the owner specifically for bucket $B_{i,j}$; this is conceptually similar to their usage in the basic scheme.

We explain the proof construction and verification process using Figure 4·7, focusing on R_i. In our example, R_i *fully covers* buckets $B_{i,\kappa'+1}, \ldots, B_{i,\kappa}$, and *partially covers* buckets $B_{i,\kappa'}$ and $B_{i,k+1}$. Observe that we can decompose R_i into three sets, let ①, ②, ③ (so that we alleviate our notation and allow an easy reference to the figure), such that $R_i = ① \cup ② \cup ③$ and ①, ②, ③ are *pairwise disjoint*. Observe also that $① = P_{i,\kappa',k} \setminus P_{i,\kappa',k'}$, $② = \mathcal{P}_{i,\kappa} \setminus \mathcal{P}_{i,\kappa'}$, and $③ = P_{i,\kappa'+1,k}$. Therefore, the server builds the proof π by including proof components $\pi_{P_{i,\kappa',k}}$, $\pi_{P_{i,\kappa',k'}}$, $\pi_{\mathcal{P}_{i,\kappa}}$, $\pi_{\mathcal{P}_{i,\kappa'}}$, $\pi_{P_{i,\kappa'+1,k}}$. Moreover, it adds $(i, j, \pi_{\mathcal{P}_{i,j}}), \tau_{P_{i,j,l}}$, their proper proofs from $\mathcal{SMA}'_i, \mathcal{SMA}_{i,j}, \mathcal{SMA}_i, \mathcal{SMA}$, as well as $\pi_{①} = acc(①)$, $\pi_{②} = acc(②)$, $\pi_{③} = acc(③)$. With all the above, the client can verify that $\pi_{①}$, $\pi_{②}$, $\pi_{③}$ are the truthful proofs for sets ①, ②, ③.

The client next needs to combine $\pi_{①}$, $\pi_{②}$, $\pi_{③}$ in order to verify that proof

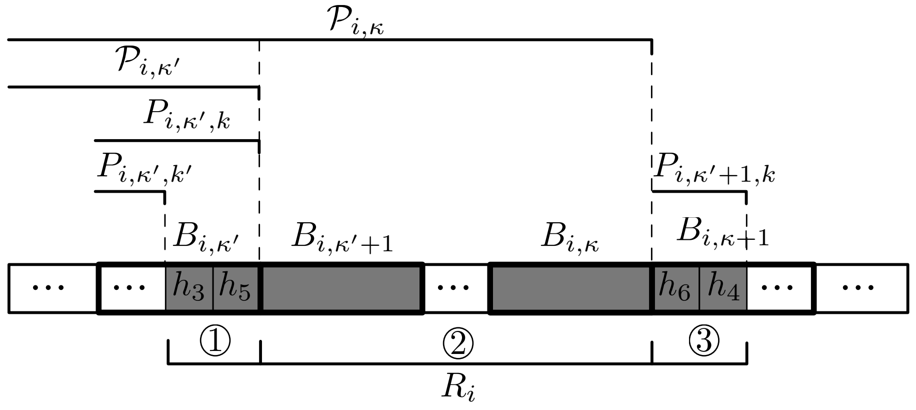


Figure 4·7: Representation of R_i through sets

$\pi_{R_i} = acc(R_i)$, also included in the final π by the server, indeed corresponds to the R_i that is the *union* of ①, ②, ③. After that point, the client can proceed to prove the final result R in an identical way to the basic scheme. For this particular task, we utilize our own customized *set union sub-protocol*, presented below. This sub-protocol is motivated by similar reasons that motivated our set difference sub-protocol previously; we need it to be executed in time $\tilde{O}(|R_i|)$, and be secure under standard cryptographic assumptions. What enables us to do this is the extra constraint that the participant sets must be a priori proven *pairwise disjoint*. At a high level, its ProveUnion routine outputs a proof $\pi_\cup$ on input sets ①, ②, ③, which later facilitates the VerifyUnion routine invoked on $\pi_①$, $\pi_②$, $\pi_③$, π_{R_i}.

Consider that tuple t is inserted in bucket $B_{i,j}$ (deletions are handled similarly). This insertion affects all b bucket prefixes in the worst case, and all n/b hash prefixes in $B_{i,j}$. It is important to observe that t does not affect any hash prefix of any other bucket; in that sense, the buckets isolate the effect of the update within their boundaries. Setting $b = \sqrt{n}$, the owner must update $O(\sqrt{n} \cdot m)$ prefixes in overall, each with a single exponentiation. Moreover, it should propagate the changes of the corresponding proofs inside the $\mathcal{SMA}$ structures, whose cost is asymptotically the same as in the case of the basic scheme. Therefore, the total update time in

```
**Algorithm** ProveUnion($X_1, X_2, X_3, pk$)
1.  Output $\pi_\cup = acc(X_1 \cup X_2)$

**Algorithm** VerifyUnion($\pi_{X_1}, \pi_{X_2}, \pi_{X_3}, \pi_X, \pi_\cup, pk$)
1.  Verify $e(\pi_{X_1}, \pi_{X_2}) = e(\pi_\cup, g)$
2.  Verify $e(\pi_\cup, \pi_{X_3}) = e(\pi_X, g)$
3.  **If** verification in Lines 1-2 fails, **return** reject, **else return** accept
```

this construction reduces from $O(n \cdot m)$ to $O(\sqrt{n} \cdot m)$. Interestingly, the asymptotic complexities of all other algorithms and the proof size remain unaffected. However, the absolute costs slightly increase due to the extra bucket prefixes, as confirmed by our experiments in Section 4.6. The proof of security for this construction follows the exact same process as that of Theorem 3. The only difference is that instead of set difference along each dimension, the operation performed is a union of disjoint sets, therefore the security of the construction needs to be reduced on breaking the soundness for such an operations. Next, we present the necessary sub-protocol for this and prove its security.

A union sub-protocol. We present a sub-protocol for proving the correctness of a union operation among a number of sets X_i *under the constraint that they are pairwise disjoint*. We focus on the case of three input sets, as this is the way it is utilized in Section 4.5. The sub-protocol consists of two routines, ProveUnion and VerifyUnion. The former receives sets X_1, X_2, X_3, and outputs a proof $\pi_\cup$ for the integrity of the union operation $X = X_1 \cup X_2 \cup X_3$. The latter receives succinct descriptions $\pi_{X_1}, \pi_{X_2}, \pi_{X_3}, \pi_X$ of X_1, X_2, X_3, X, respectively, as well as a proof $\pi_\cup$, and returns `accept` if X is the union of the three sets, and `reject` otherwise. We provide the pseudo codes of the two routines below.

Similar to the set difference sub-protocol, these routines are meaningful only as part of a $\mathcal{SOA}$ scheme based on bilinear accumulators. ProveUnion runs in time $\tilde{O}(|X_1| + |X_2| + |X_3|)$. For VerifyUnion, it is the responsibility of the caller to check that π_{X_1}, π_{X_2}, π_{X_3} are the accumulation values of pairwise disjoint X_1, X_2, and X_3, prior to calling the routine. Its cost is $O(1)$ pairings. The following lemma is useful in our proofs. It states the claim for the union of two disjoint sets but, it can trivially be generalized for sets X_i for $i \in [k]$, as long as $X_i \cap X_j = \emptyset$ for $i, j \in [k]$ and $i \neq j$.

Lemma 8. *Let λ be a security parameter, $pub \leftarrow \mathsf{GenBilinear}(1^\lambda)$, and elements $(g, g^s, ..., g^{s^q}) \in \mathbb{G}$, computed for some s chosen at random from $\mathbb{Z}_p^*$. Let X_1, X_2 be sets with elements in $\mathbb{Z}_p$, such that $X_1 \cap X_2 = \emptyset$. For an element $y \in \mathbb{G}$, it holds that $y = acc(X_1 \cup X_2)$, iff $e(y, g) = e(acc(X_1), acc(X_2))$.*

Proof: ($\Leftarrow$) Let $X_1 = \{x_1, ..., x_{l'}\}$ and $X_2 = \{x_1, ..., x_l\}$ for $l, l' \in \mathbb{N}$ with $X_1 \cap X_2 = \emptyset$. If $e(y, g) = e(acc(X_1), acc(X_2))$, then we have $e(y, g) = e(g^{\prod_{i=1}^{l'}(x_i+s)}, g^{\prod_{i=1}^{l}(x_i+s)})$. Hence, it holds $e(y, g) = e(g^{\prod_{i=l+1}^{l'}(x_i+s) \cdot \prod_{i=1}^{l}(x_i+s)}, g) \overset{\text{def}}{=} e(acc(X_1 \cup X_2), g)$ which implies that $y = acc(X_1 \cup X_2)$, since $e(g, g)$ is a generator of $\mathbb{G}_T$.

($\Rightarrow$) If $y = acc(X_1 \cup X_2) = g^{\prod_{i=1}^{l'}(x_i+s) \cdot \prod_{i=1}^{l}(x_i+s)}$, then it holds that $e(y, g) = e(g^{\prod_{i=1}^{l}(x_i+s) \cdot \prod_{i=1}^{l'}(x_i+s)}, g) = e(g^{\prod_{i=1}^{l}(x_i+s)}, g^{\prod_{i=1}^{l'}(x_i+s)}) = e(acc(X_1), acc(X_2))$. □

4.6 Performance evaluation

We performed our experiments on a 64-bit machine with Intel Core i5 CPU 2.5GHz, running Linux. We implemented both versions of our scheme in C++, using the following libraries: DCLXVI [DCL16] for fast bilinear pairing computations, Flint [Fli16] for modular arithmetic, and Crypto++ [Cry16] for SHA-256 hash operations. DCLXVI employs a 256-bit BN elliptic curve and an asymmetric optimal ate pairing, offering bit-level security of 128 bits. We represent elements of $\mathbb{G}_1$ with 768 bits using Jacobi coefficients, which yield faster operations. Elements in $\mathbb{G}_2$ are roughly twice as large as those of $\mathbb{G}_1$. We chose an asymmetric pairing for efficiency reasons, but we note that this choice does not introduce any redundancy to our schemes as presented with symmetric pairings. We instantiate all $\mathcal{SMA}$'s with

Operation		**Cost**
Exp. in $\mathbb{G}_1$ / $\mathbb{G}_2$		0.55 / $0.94\ ms$
Mult. in $\mathbb{Z}_p$ / $\mathbb{G}_T$		$7\ \mu s$ / $0.09\ ms$
SHA-256 / Bilinear pairing		$5\ \mu s$ / $1.41\ ms$
Quicksort in $\mathbb{Z}_p$	(100/1000/10000 elems.)	0.1 / 0.9 / $4.6\ ms$
Acc. in $\mathbb{G}_1$	||	25.3 / 236 / $2,628\ ms$
Acc. in $\mathbb{G}_2$	||	32.6 / 338 / $3,471\ ms$
Polynom. Mult in $\mathbb{Z}_p[r]$	(100/1000/10000 coeffs.)	0.4 / 7.3 / $92.9\ ms$
XGCD in $\mathbb{Z}_p[r]$	||	8.4 / 599 / $108,093\ ms$

Table 4.1: Costs of primitive operations

Merkle trees [Mer89] and bilinear accumulator trees [PTT08]. Table 4.1 summarizes all primitive costs involved in our schemes.

We test four possible configurations: (i) our scheme with Merkle trees (Basic-Mer), (ii) our basic scheme with accumulator trees (Basic-Acc), (iii) our update-efficient scheme with Merkle trees (UpdEff-Mer), and (iv) our update-efficient scheme with accumulator trees (UpdEff-Acc). For each configuration, we assess the performance at the client, owner and server, varying several parameters. We run each experiment 10 times and report the average costs. Note that the performance of all schemes does not depend on the data distribution, but rather on the table schema and result selectivities. As such, we used synthetic datasets in our evaluation.

Client. Figure 4·8 depicts the verification cost at the client. This overhead is mainly affected by the result size $|R|$ and the number of query dimensions d. Figure 4·8 (left) shows the CPU time (in ms) as a function of $|R|$, fixing $d = 32$, $n = 10^6$ and $m = 64$. The verification cost increases with $|R|$ in all schemes. Basic-Mer is the fastest for $|R| \leq 1,000$. This is because the Merkle-based schemes are faster than the accumulator-based ones, as they entail hash operations for the $\mathcal{SMA}$ proofs, which are much cheaper than the pairings needed in accumulation trees. Moreover, the overhead in the update-efficient schemes is slightly larger than that in their basic counterparts, due to the extra proof verifications of the bucket prefixes and the taller

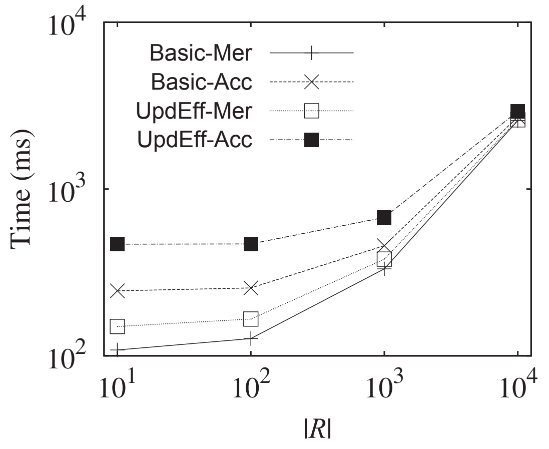


Figure 4·8: Verification overhead at client

$\mathcal{SMA}$ hierarchy. Nevertheless, observe that, for $|R| = 10,000$ the performance of all schemes converges. The reason is that the computation of $\pi_R = acc(R)$ that is common to all techniques becomes the dominant factor, which effectively hides the costs of the $\mathcal{SMA}$ proofs and all set operation verifications.

Figure 4·8 (right) illustrates the CPU time versus d, when $|R| = 1,000$, $n = 10^6$ and $m = 64$. The performance of the schemes is qualitatively similar to Figure 4·8 (left) for the same reasons. Once again, all costs increase linearly with d because the verification overhead of the set differences and intersections is also linear in d. However, the effect of d on the total CPU time is not as significant as that of $|R|$, since the common accumulation cost for R emerges as the dominant cost when $|R| = 1,000$. In both figures, the verification time for all constructions is between 20 ms and 3.36 seconds.

Table 4.2 includes the proof sizes for the four schemes when varying d. We make three observations. First, all sizes increase with d, since the proof includes components for every dimension. Second, the basic schemes have smaller proofs than their counterparts, again because of the extra bucket prefix proofs and taller $\mathcal{SMA}$ hierarchy. Third, although Basic-Acc outperforms Basic-Mer , this is not true

d	**2**	**4**	**8**	**16**	**32**
Basic-Mer	4.5	9.1	18.1	36.3	72.5
Basic-Acc	3.6	7.2	14.3	28.8	57.5
UpdEff-Mer	9.2	18.4	36.9	73.8	147.5
UpdEff-Acc	9.6	19.2	38.4	76.8	153.5

Table 4.2: Proof size in KB ($n = 10^6, m = 64$)

for UpdEff-Acc and UpdEff-Mer. This is because, although accumulators provide asymptotically smaller proofs than Merkle trees, this does not hold in practice for the database sizes we tested. In overall, the proofs for all schemes are quite succinct, ranging from 4.5 to 153.5 KBs, which are independent of the result size that could easily be in the order of MBs.

Owner. Figure 4·9 assesses the performance of the owner for the setup stage (which includes the key generation), and updates. In this set of experiments, we focus only on the Merkle-based schemes that have a clear performance advantage over the accumulator-based, as evident also from our evaluation for the client above. Figure 4·9 (left) plots the pre-processing cost when varying n and fixing $m = 64$. Naturally, the overhead increases linearly with n in both schemes. This overhead is dominated by the computation of $\pi_{P_{i,j}}$ for all i, j, which completely hides the sorting and hashing costs (see also Table 4.1). As expected, UpdEff-Mer is more than twice as slow as Basic-Mer. Although the pre-processing time can reach up to three hours for $n = 10^6$, recall that this is a one-time cost for the owner.

Figure 4·9 (right) evaluates the update time as a function of the number of updates performed in a single batch operation, where $n = 10^5$ and $m = 64$. Note that we report the respective *worst case* in both schemes. For Basic-Mer, the CPU time is practically unaltered and, in fact, is as bad as the setup overhead. On the contrary, UpdEff-Mer is greatly benefited by the bucket isolation and becomes up to more than two orders of magnitude more efficient than Basic-Mer. As the number of updates in the batch increase, the performance gap between the two schemes closes, since the

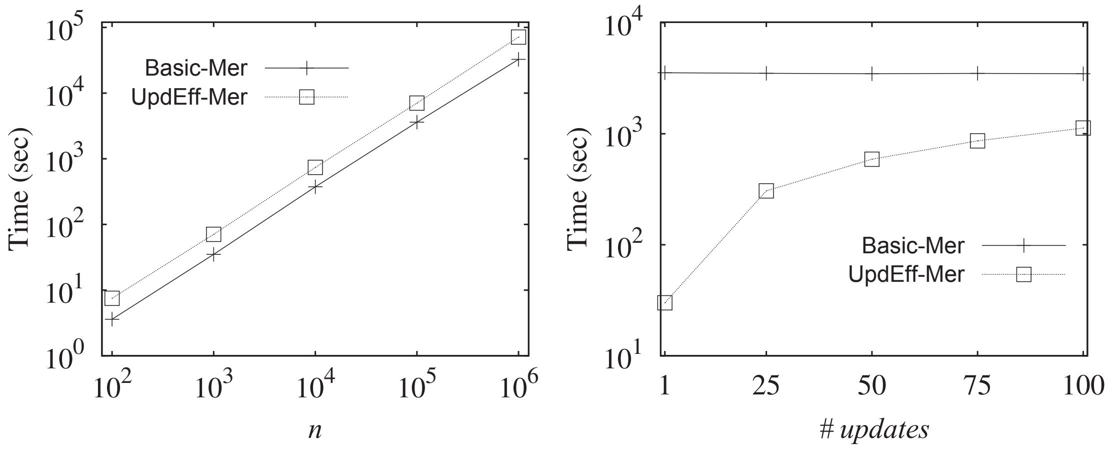


Figure 4·9: Setup (left) and update (right) overhead at owner.

updates in the batch are likely to affect more buckets. For the tested settings, the update time ranges between 30 seconds and one hour.

Server. Figure 4·10 reports the proof generation time at the server. As explained in our complexity analysis, the dominant factor here is $\sum_{i=1}^{d} |R_i|$. Therefore, due to the lack of real-world data and query workloads, it suffices to vary $|R_i|$ and fix it across all dimensions, rather than varying d and setting an arbitrary partial result size per dimension. Figure 4·10 depicts the CPU time at the server, when varying $|R_i|$ and setting $n = 10^5$, $m = 64$, $d = 32$ and $|R| = 0.1 \cdot |R_i|$ (i.e., 10% of a 1-dimensional result). At every point of the curve, we also provide the percentages of the three dominant computational costs, namely the construction of W_i, F_i (for the intersection proof) and π_{R_i}. The performances of two schemes differ marginally. This is because the generation of the extra set union proof of UpdEff-Mer incurs negligible cost compared to the large burden of computing the three types of elements mentioned above. The most interesting observation is that, for $|R_i| = 10$, the cost for W_i is 51% and for F_i is 15%, whereas for $|R_i| = 10,000$, the two costs become 7% and 87%, respectively. This is because computing F_i requires running the Extended

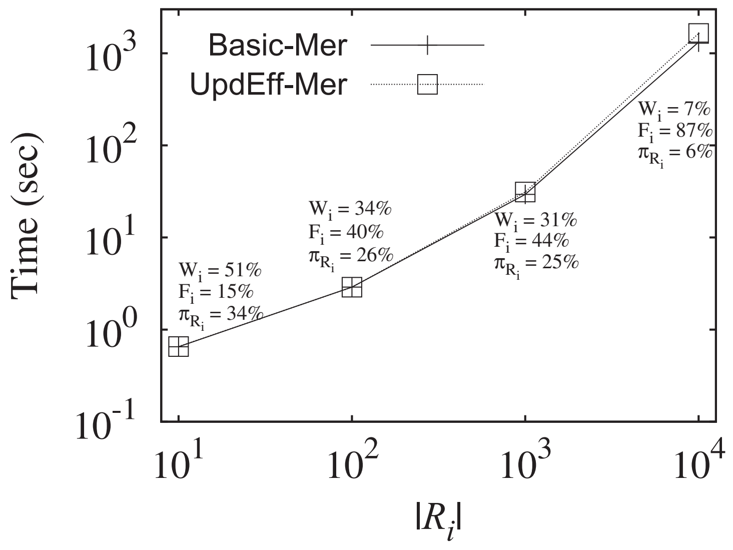


Figure 4·10: Proof construction cost at server

Euclidean (XGCD) algorithm, whose time increases drastically with the degree of the polynomials (as shown in Table 4.1), dictated by $|R_i|$. The server's total overhead ranges between 650 ms and 25 minutes.

Comparison with general-purpose VC. Regarding support of range queries with general-purpose VC schemes, [ZKP15] measures the costs for evaluating and providing a proof for a 10-dimensional query over a table with 10 attributes and just 1000 tuples, using the library of [BCTV14]. Even for this tiny dataset (much smaller than the ones used in our previous experiments), the server needs $\approx$ 329 seconds, for computing a single proof. It should be noted that this cost is fixed, i.e., independent of the size of partial or final result. For comparison, for the same query, our scheme takes anywhere from some milliseconds to a few seconds (depending, as discussed above, on $|R_i|$). For example, for R_i set to 100, the time for proof construction is approximately 1.1 second, i.e., more than 300x smaller. On the other hand, the proof size produced with the library of [BCTV14] is fixed to 288 bytes which is smaller than what is achieved with our approach. However, we believe that an increase of proof size to a few KB is justified (and a few KB is certainly not a prohibitive quantity) given the significant computational gain at the server.

Summary and future improvements. Our experimental evaluation confirms the feasibility of our schemes. Specifically, it demonstrates that the verification cost at the client in all schemes is in the order of a few seconds in the worst case, even for moderate result sizes, whereas the proof size is up to a few hundred of KBs. At the owner, we illustrated the benefits of our update-efficient scheme over the basic one in terms of updates, which come at the cost of a more expensive setup and client verification. Finally, the server is the most impacted party in our constructions. The proof generation cost takes from several ms to several minutes, for small and moderate partial result sizes and dimensionality.

Nevertheless, it is important to stress that the defining costs at the server account for exponentiations and modular polynomial arithmetic. These operations are at the core of numerous applications and, thus, there is huge potential for improvement in the near future. In addition, there are works (e.g., [Eme11]) that have substantially boosted such operations with modern hardware, which we did not possess in our experimentation. Being instantiations of a general framework, our schemes feature the attractive property that they are easily upgradeable with future advances in such cryptographic tools.

Chapter 5

Verifiable Pattern Matching Queries

5.1 Introduction

In this chapter we design protocols for verifiable processing of *pattern matching* queries. The problem setting involves an outsourced textual database, a query containing a text *pattern*, and an answer regarding the presence or absence of the pattern in the database. In its most basic form, the database consists of a single text $\mathcal{T}$ from an alphabet Σ, where a query for pattern p, expressed as a string of characters, results in answer "match at i", if p occurs in $\mathcal{T}$ at position i, or in "mismatch" otherwise. More elaborate models for pattern matching involve queries expressed as regular expressions over Σ or returning multiple occurrences of p, and databases allowing search over multiple texts or other (semi-)structured data (e.g., XML data). This core data-processing problem has numerous applications in a wide range of topics including intrusion detection [KS94], spam filtering [CB06], web search engines [RH02], computational biology [AQD+02] and natural language processing [FAA+94].

5.1.1 Prior work

Previous works on authenticated pattern matching include the schemes by Martel et al. [MND+04] for text pattern matching, and by Devanbu et al. [DGK+04] and Bertino et al. [BCF+04] for XML search. In essence, these works adopt the same general framework: First, by hierarchically applying a cryptographic hash function (e.g., SHA-2) over the underlying database, a short secure description or *digest* of the data is computed. Then, the answer to a query is related to this digest via a proof that provides step-by-step reconstruction and verification of the *entire answer computation*. This approach typically leads to large proofs and, in turn, high verification costs, proportional to the number of computational steps used to produce the answer. In fact, for XML search, this approach offers no guarantees for the worst-case cost of verification, since certain problem instances require that the proof includes almost the entire XML document, or a very large part of it, in order to ensure that no portions of the true (honest) answer were omitted from the returned (possibly adversely altered) answer.

On the other hand, recent work on built systems for general-purpose VC (e.g., [PHGR13, BFR$^+$13b, BCG$^+$13]) allows verification of general classes of computation. Here also, verification is based on cryptographic step-by-step processing of the entire computation, expressed by circuits or RAM-based representations. Although special encoding techniques allow for constant-size proofs and low verification costs, this approach cannot yet provide practical solutions for pattern matching, as circuit-based schemes inherently require complex encodings of all database searches, and RAM-based schemes result in very high proof generation costs, that can range in the order of hours for even medium database sizes (for example, see [ZPK14]). Indeed, costly proof generation comprises the main bottleneck in all existing such implementations.

5.1.2 Overview of result

We wish to design schemes for authenticated pattern matching that have the following two properties: (i) the validation of the correctness of the answer is based on a proof that is *succinct*, having size independent of the database size and the query description, and (ii) this proof can be *quickly generated and verified*. We emphasize that our requirement to support pattern matching verification with easy-to-compute constant-size proofs is in practice a highly desired property. First, it contributes

	space	**setup**	**proof size**	**query time**	**verification time**
[MND+04]	n	n	$m \log \Sigma$	$m \log \Sigma + \kappa$	$m \log \Sigma + \kappa$
Our construction	n	n	1	$m + \kappa$	$m \log m + \kappa$
[BCF+04, DGK+04] (any path)	n	n	n, d	n, d	n, d
Our construction (exact path)	n	n	1	m	$m \log m + s$

Figure 5·1: Asymptotic complexities of our scheme for text pattern matching and XML exact path queries: n is the size of the document, m the pattern length, κ the number of occurrences, Σ the alphabet size, s the answer size, and d the number of valid paths.

to *high scalability in query-intensive applications* in settings where the server that provides querying service for outsourced databases receives incoming requests by several clients at high rates; then obviously, faster proof generation and transmission of constant-size proofs result in faster response times and higher throughputs. But it also promotes *storage efficiency in data-intensive applications* in settings where the proof for a (mis)match of any pattern query over a database must be persistently retained in storage for a long or even unlimited time duration; then, minimal-size proofs result in the minimum possible storage requirements, a very useful feature in big-data environments.

An example of a data-intensive application where pattern matching proofs might be permanently stored, is the problem of *securing the chain of custody* in forensic and intrusion detection systems used by enterprises today. Such systems often apply big-data security analytics (e.g., [YOO$^+$13]) over terabytes of log or network-traffic data (collected from host machines, firewalls, proxy servers, etc.) for analysis and detection of impending attacks. Since any data-analytics tool is only as useful as the quality (and integrity) of its data, recent works (e.g., [YNR12, BHJT14]) focus on the integrity of the data consumed by such tools, so that any produced security alert carries a cryptographic proof of correctness. To support a verifiable chain of custody,[11] these proofs must be retained for long periods of time. As big-data security analytics grow in sophistication, authenticated pattern matching queries will be crucial for effective analytics (e.g., to match collected log data against known high-risk signatures of attacks), hence storing only constant-size associated proofs will be important in the fast-moving area of information-based security.

We present the first authentication schemes for pattern matching over textual

[11]Informally, any security alert—carrying important forensic value—can be publicly and with non-repudiation verified—thus, carrying also legal value when brought as evidence to court months, or even years, after the fact.

databases that achieve the desired properties explained above. Our schemes employ a novel authenticated version of the suffix tree data structure [GT02] that can provide *precomputed (thus, fast to retrieve), constant-size* proofs for any basic-form pattern matching query, at no asymptotic increase of storage.

Our pattern matching scheme has the following attractive properties:

- The size of the proof is $O(1)$; specifically, it always contains at most 10 bilinear group elements.
- The time to generate the proof that a query pattern of size m is found in κ occurrences is $O(m + \kappa)$, and *very short* in practice, as it involves *no* cryptographic operations but only assembling of *precomputed* parts—e.g., it takes less than 90μs to respond to a query of size 100 characters: 80μs to simply find the (mis)match and less than 10μs to assemble the proof.

We extend our scheme to also support regular expressions with a constant number of wildcards. Moreover, we apply our scheme for the authentication of *pattern matching queries over collections of text documents* (returning the indices of documents with positive occurrences), and *exact path queries* over XML documents. By design, these schemes also achieve an asymptotically optimal communication overhead (that is, the asymptotic communication cost is the same as simply transmitting the answer itself): On top of the requested answer, the server provides only a constant number of bits (modulo the security parameter)—e.g., for XML search and 128-bit security level, proofs can be made as small as $\sim$178 bytes. Unlike existing hash-based authentication schemes [BCF$^+$04, DGK$^+$04, MND$^+$04], our authentication schemes support fully parallelizable setup: They can be constructed in $O(\log n)$ parallel time on $O(n/\log n)$ processors in the EREW model, thus maintaining the benefits of known parallel algorithms for (non-authenticated) suffix trees [JáJ92, MASK11]. While the use of precomputed proofs best matches static text databases, we also present efficient fully or semi-dynamic extensions of our schemes.

5.1.3 Overview of techniques

Our construction is again an ADS and we follow the framework of [TT10]: Our scheme first defines and encodes *answer-specific* relations that are sufficient for *certifying* (unconditionally) that an answer is correct and, then, cryptographically authenticates these relations using optimal-size proofs. We achieve this by employing in a novel way the bilinear accumulator over a special encoding of the database with respect to a suffix tree, used to find the pattern (mis)match. The encoding effectively takes advantage of the suffix tree where patterns in the database share common prefixes, which in turn can be succinctly represented by an accumulator. For the XML query application, we use the same approach, this time over a trie defined over all possible paths in the document, and we link each path with the respective XML query answer (i.e., all reachable XML elements).

Comparison with related works. Table 5·1 summarizes our work as compared to [BCF$^+$04, DGK$^+$04, MND$^+$04].[12] In [MND$^+$04] a general technique is applied to the suffix tree, that authenticates every step of the search algorithm, thus obtaining proof size proportional to the length of the pattern, which is not optimal. Moreover, due to the use of sequential hashing, this solution is inherently not parallelizable. The authors of [DGK$^+$04] authenticate XPath queries over a simplified version of XML documents by relying on the existence of a document type definition (DTD) and applying cryptographic hashing over a trie of all possible semantically distinct path queries in the XML document. An alternative approach is taken in [BCF$^+$04], where

[12]We note that our ADS schemes operate with any accumulator, not just the accumulator. In fact, using the RSA accumulator [CL02a] reduces verification cost to $O(m)$. However, a recent experimental comparison demonstrates that the bilinear accumulator is more efficient in practice [Tre13].

similar XML queries are authenticated by applying cryptographic hashing over the XML tree structure. As discussed above, both these approaches suffer from very bad worst-case performance, e.g., yielding verification proofs/costs that are proportional to the size of the XML tree. However, these works are designed to support *general* path queries, not only *exact*, as our work does. Recently, the authors of [FHV13] presented a protocol for verifiable pattern matching that achieves security and secrecy in a very strong model, hiding the text even from the responding server. While that work offers security in a much more general model than ours, it has the downside that the owner that outsourced the text is actively involved in query responding and that it makes use of heavy cryptographic primitives, the practicality of which remains to be determined. There is a large number of ADS schemes in the database and cryptography literature for various classes of queries (e.g., [Mer89, CPPT14, LHKR10]). Also related to our problem is keyword search authentication, which has been achieved efficiently, e.g., in [PM08, YPPK09a, PTT11]. As previously discussed, verifiable computation systems such as [PHGR13, BFR$^+$13b] can be used for the verification of pattern matching; although optimized to provide constant-size proofs these constructions remain far from practical. Finally, parallel algorithms in the context of verifiable computation have only recently been considered. In [TRMP12, SBV$^+$13] parallel algorithms are devised for constructing a proof for arithmetic-circuit computations.

5.2 Pattern matching queries

The problem of pattern matching involves determining whether a pattern appears within a given text. In its basic form, assuming an alphabet Σ of size $|\Sigma| = \sigma$, a n-character text $\mathcal{T} \in \Sigma^n$ and a pattern $p \in \Sigma^m$ of length m, the problem is expressed as "*is there position* $1 < i \leq n-m+1$ *such that* $\mathcal{T}[i+j] = p[j]$ *for* $j = 0, \ldots m-1$*?*", where $\mathcal{T}[i]$ is the character at the i-th position in the text, and likewise for pattern p. If there exists such i, the answer is "match at i", otherwise "mismatch".

Answering pattern matching queries is an arduous task, if done naively. For instance, to check the occurrence of p in T, one could sequentially test if p occurs at any position i (i.e., if it is a prefix of some suffix of $\mathcal{T}$), for all positions in $\mathcal{T}$. Such a successful test would imply a match but would require $O(n)$ work. However, with some preprocessing of $O(n)$ work, one can organize patterns in a *suffix tree* [Wei73], reducing the complexity of pattern matching query from $O(n)$ to $O(m)$. A suffix tree is a data structure storing *all* the suffixes of $\mathcal{T}$ in a way such that any repeating patterns (common prefixes) of these suffixes are stored once and in a hierarchical way, so that every leaf of the suffix tree corresponds to a suffix of the text $\mathcal{T}$. This allows for (reduced) $O(m)$ search time while maintaining (linear) $O(n)$ space usage.[13]We provide next a more detailed description of the suffix tree data structure, represented as a directed tree $G = (V, E, \mathcal{T}, \Sigma)$. We refer to the example of Figure 5·2 depicting the suffix tree for the word minimize.

Each leaf of G corresponds to a distinct suffix of $\mathcal{T}$, thus G has exactly n leaves. We denote with $S[i]$ the i-th suffix of T, that is, $S[i] = \mathcal{T}[i] \ldots \mathcal{T}[n]$, for $i = 1, \ldots, n$. Internal tree nodes store common prefixes of these n suffixes $S[1], S[2], \ldots, S[n]$, where the leaves themselves store any "remainder" non-overlapping prefixes of T's suffixes. If leaf v_i corresponds to suffix $S[i]$, then $S[i]$ is formed by the *concatenation* of the contents of all nodes in the root-to-leaf path defined by v_i, where the root conventionally stores the empty string. For instance, in Figure 5·2, $S[4] =$ imize and $S[6] =$ ize, respectively associated with the paths defined by the second and fourth most left leaves, labelled by mize and ze (having as common parent the node labelled by i).

[13]This can be easily achieved by storing pointers to the text at the nodes, instead of entire prefixes.

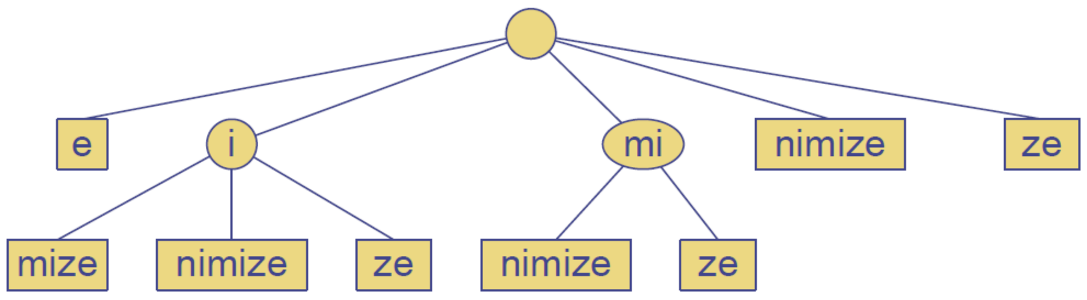


Figure 5·2: Suffix tree for minimize storing suffixes minimize, inimize, nimize, imize, mize, ize, ze, e as eight overlapping paths.

Additionally, every node $v \in V$ stores the following information that will be useful in the case of the mismatch: **(a)** the *range* $r_v = (s_v, e_v)$ of v, which corresponds to the start (s_v) and end (e_v) position of the string stored in v in the text (we pick an arbitrary range if v is associated with multiple ranges); **(b)** the *depth* d_v of v, which corresponds to the number of characters from the root to v, i.e., the number of characters that are contained in the path in G that consists of the ancestors of v; **(c)** the *sequel* $\mathcal{C}_v$ of v, which corresponds to the set of initial characters of the children of v. For example in Figure 5·2, for the node v labelled mi, it is $s_v = 1$ and $e_v = 2$ (or $s_v = 5$ and $e_v = 6$), $d_v = 0$, $\mathcal{C}_v = \{n, z\}$.

Traversing a suffix tree. This data structure allows for efficient searches of any given pattern p. Since all matching patterns must be a prefix of some suffix, the search algorithm beings from the root and traverses down the tree incrementally matching pattern p with the node labels, until it reaches some node v where either a mismatch or a complete match is found. We model this search on suffix tree $G = (V, E, \mathcal{T}, \Sigma)$ by algorithm $(v, k, t) \leftarrow \mathsf{suffixQuery}(p, G)$, returning: (1) the *matching node* v, i.e., the node of G at which the algorithm stopped with a match or mismatch, (2) the *matching index* k, $s_v \leq k \leq e_v$, i.e., the index (with reference to the specific range (s_v, e_v)) where the last matching character occurs (for successful matching

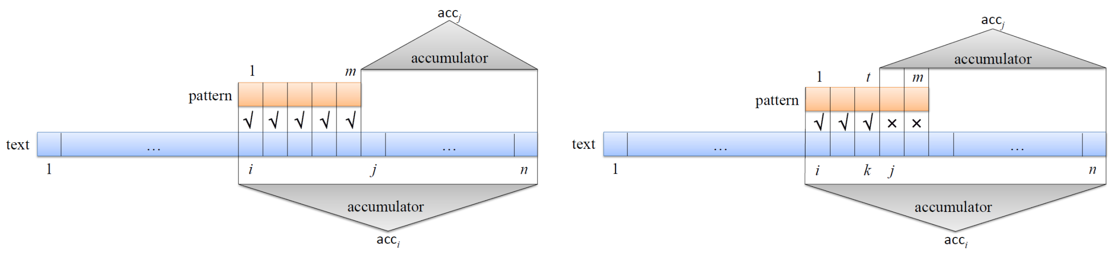


Figure 5·3: (Left) Pattern matching in our scheme for pattern p ($|p| = m$), using suffixes $S[i]$ and $S[j]$, where $S[i] = pS[j]$. (Right) Pattern mismatch in our scheme, using suffixes $S[i]$ and $S[j]$, where $S[i] = p_1p_2 \dots p_tS[j]$ and $t < m$.

searches, k coincides with the index of the last character of p within v), and (3) the *prefix size* $t \leq m$, i.e., the length of maximum matching prefix of p (m in case of a match). Figure 5·3 shows the relation of variables k and t for both cases.

Characterization of pattern matching queries. We provide here two important lemmas that *characterize* the correct execution of algorithm suffixQuery, by providing necessary and sufficient conditions for checking the consistency of a match or mismatch of p in T with the output (v, k, t) produced by suffixQuery. In the next section, we will base the security of our construction on proving, in a *cryptographic* manner, that these conditions hold for a given query-answer pair. Namely, the structure of these relations allows us to generate *succinct* and efficiently verifiable proofs. In the following we denote with xy the concatenation of two strings x, y (order is important).

Lemma 9 (Pattern match)**.** *There is a* match *of p in $\mathcal{T}$ if and only if there exist two suffixes of $\mathcal{T}$, $S[i]$ and $S[j]$, with $i \leq j$, such that $S[i] = pS[j]$.*

Proof: ($\Rightarrow$) Suppose there is a match of p it $\mathcal{T}$. Let i be the index where the match starts and j be the index where the match ends. Then for the suffixes $S[i]$ and $S[j]$ it holds $S[i] = pS[j]$. ($\Leftarrow$) Suppose there exist two suffixes $S[i]$ and $S[j]$ of a text $\mathcal{T}$ for which it holds $S[i] = pS[j]$. Then p has to be part of the text $\mathcal{T}$ and therefore there is match of p in $\mathcal{T}$. $\square$ $\square$

Lemma 10 (Pattern mismatch)**.** *There is a* mismatch *of p in $\mathcal{T}$ if and only if there exist a node $v \in G$, an integer $k \in [s_v, e_v]$ and an integer $t < m$ such that $S[s_v - d_v] = p_1p_2 \ldots p_tS[k+1]$ and $p_{t+1} \neq \mathcal{T}[k+1]$, if $k < e_v$, or $p_{t+1} \notin c_v$ if $k = e_v$.*

Proof: ($\Rightarrow$) Suppose there is a mismatch of p in $\mathcal{T}$. The desired triplet (v, k, t) is the one output by the algorithm $(v, k, t) \leftarrow \mathsf{suffixQuery}(p, G)$, where $v \in G$ is the node of the suffix tree where the mismatch is returned, $k \in [s_v, e_v]$ is the matching index and $t < m$ is the prefix size. Since t is the prefix size there is a match of $p_1p_2 \ldots p_t$ in $\mathcal{T}$. Therefore, by Lemma 9 there exist i and j with $i \leq j$ such that $S[i] = p_1p_2 \ldots p_tS[j]$. By the properties of the suffix tree, it is $i = s_v - d_v$ and $j = k + 1$ (note that since $t = d_v + k - s_v + 1 \geq 0$ it is always $i \leq j$, as required by Lemma 9). We distinguish the following cases:

- $k < e_v$. The mismatch is happening within the node v. All patterns in $\mathcal{T}$ starting with $p_1p_2 \ldots p_t$ traverse that node up to index $k < e_v$. Therefore it has to be $p_{t+1} \neq \mathcal{T}[k + 1]$;

- $k = e_v$. The last matching character is the last character of node v. All patterns in $\mathcal{T}$ starting with $p_1p_2 \ldots p_t$ traverse the whole node (i.e., up to index $k = e_v$) and continue with any of its children. Since there is a mismatch after that, it is the case that p_{t+1} should not belong to the sequel of v, which implies $p_{t+1} \notin c_v$.

($\Leftarrow$) By contradiction. Suppose there exist a node $v \in G$, an integer $k \in [s_v, e_v)$ and an integer $t < m$ such that $S[s_v - d_v] = p_1p_2 \ldots p_tS[k + 1]$ and $p_{t+1} \neq \mathcal{T}[k + 1]$ and however, there is a *match* of p in $\mathcal{T}$. Since $S[s_v - d_v] = p_1p_2 \ldots p_tS[k+1]$, all patterns in $\mathcal{T}$ starting with $p_1p_2 \ldots p_t$ traverse that node v to index $k < e_v$. Therefore, for a match to exist, it must be $p_{t+1} = \mathcal{T}[k + 1]$. This is contradiction. For the second case, the same argument holds. The contradiction is reached by deriving the false argument $p_{t+1} \in c_v$. This completes the proof. $\square$ $\square$

With reference to Figure 5·2, the match of the pattern $p = \mathsf{inim}$, can be shown by employing the suffixes $S[2] = \mathsf{inimize}$ and $S[6] = \mathsf{ize}$. Note that indeed $S[2] = pS[6]$. This is a match (as in Lemma 9). More interestingly, observe the case of mismatch for the string $p = \mathsf{minia}$. For this input, algorithm $\mathsf{suffixQuery}$ returns the node v labelled by nimize where the mismatch happens, matching index $k = 4$ and prefix size $t = 4$. For node v, we have $s_v = 3$ and $e_v = 8$ and also $d_v = 2$. To demonstrate the mismatch, it suffices to employ suffixes $S[s_v - d_v] = S[1] = \mathsf{minimize}$ and $S[k+1] = S[5] = \mathsf{mize}$ as well as symbols p_{t+1} and $\mathcal{T}[k+1]$. The concatenation of the prefix mini of p (of size $t = 4$) with the suffix $S[5]$ is $S[1]$, and also $p_{t+1} = \mathsf{a} \neq \mathcal{T}[k+1] = \mathsf{m}$. This is a mismatch (as the first case considered in Lemma 10, since $k < e_v$). Finally, to demonstrate the mismatch of the string $p = \mathsf{mia}$ we proceed as follows. Algorithm $\mathsf{suffixQuery}$ returns the node v labelled by mi where the mismatch happens, matching index $k = 6$ and prefix size $t = 2$. For node v, we have $s_v = 5$ and $e_v = 6$ (alternatively we can also have $s_v = 1$ and $e_v = 2$) and also $d_v = 0$. It suffices to employ suffixes $S[s_v - d_v] = S[5] = \mathsf{mize}$ and $S[k+1] = S[7] = \mathsf{ze}$ as well as symbol p_{t+1} and sequel (set) c_v. Note that indeed the concatenation of the prefix mi of p (of size $t = 2$) with the suffix $S[7]$ is $S[5]$, and that also $p_{t+1} = \mathsf{a} \notin c_v = \{\mathsf{n}, \mathsf{z}\}$. This is a mismatch (as in Lemma 10, since $k = e_v$).

5.3 Main construction

We now present our main ADS scheme for verifying answers to pattern matching queries. Our construction is based on building a suffix tree over the outsourced text and proving in a secure way the conditions specified in Lemmas 9 and 10 for the cases of match and mismatch respectively. The main cryptogaphic tool employed is the bilinear accumulator, which will be used to authenticate the contents of a suffix tree in a structured way, allowing the server to prove the existence of appropriate suffixes in the text and values in the tree that satisfy the conditions in the two lemmas. Moreover, due to the properties of the bilinear accumulator, the produced proofs will be independent of the size of the text and the pattern, consisting only of a *constant* number of bilinear group elements.

Key generation. The algorithm **genkey** proceeds as follows. The text owner first runs $\mathsf{GenBilinear}(1^\lambda)$ to compute bilinear parameters $pub = (p, \mathbb{G}, \mathbb{G}_T, e, g)$. He then picks a random $s \in \mathbb{Z}_p$ and computes $\mathbf{g} = [g, g^s, \ldots, g^{s^\ell}]$. Finally, the key pair is defined as $\mathsf{sk} = s$, $\mathsf{pk} = (pub, \mathbf{g})$.

Setup. The setup process is described in pseudo-code in Algorithm 1 and we provide a detailed explanation of each step here. The owner first computes a *suffix accumulation* for each suffix in the text with a linear pass. This value encodes information about the text contents of the suffix, its starting position and its leading character. In particular, $\mathsf{acc}(S[i])$ is denoted as $\mathsf{acc}_i := \mathsf{acc}(\mathcal{X}_{i1} \cup \mathcal{X}_{i2} \cup \mathcal{X}_{i3})$, where **(a)** $\mathcal{X}_{i1}$ is the set of *position-character* pairs in suffix $S[i]$, i.e., $\mathcal{X}_{i1} = \{(\mathsf{pos}, i, \mathcal{T}[i]), \ldots, (\mathsf{pos}, n, \mathcal{T}[n])\}$; **(b)** $\mathcal{X}_{i2}$ is the *first character* of $S[i]$, i.e., $\mathcal{X}_{i2} = \{(\mathsf{first}, \mathcal{T}[i])\}$; and **(c)** $\mathcal{X}_{i3}$ is the *index* of $S[i]$, i.e., $\mathcal{X}_{i3} = \{(\mathsf{index}, i)\}$. Also, for each suffix $S[i]$ he computes a *suffix structure accumulation* $t_i = \mathsf{acc}(\mathcal{X}_{i1})$, i.e. it contains only the position-character pairs in the suffix and its use will be discussed when we explain the verification process of our scheme. Structure accumulations are a very important part for the security of our construction. Observe that the suffix structure accumulation t_i encompasses only a *subset* of the information encompassed in acc_i. The security of the bilinear accumulator makes proving a false subset relation impossible, hence no efficient adversary can link t_i with acc_j for $j \neq i$.

Following this, the owner builds a suffix tree $G = (V, E, \mathcal{T}, \Sigma)$ over the text and computes a *node accumulation* for each $v \in G$. This value encodes all the information regarding this node in G, i.e., the range of $\mathcal{T}$ it encompasses, its depth in the tree and the leading characters of all its children nodes (taken in consecutive pairs). More formally, for a node v with values $(s_v, e_v), d_v, c_v$, its accumulation is defined as $\mathsf{acc}_v :=$ $\mathsf{acc}(\mathcal{Y}_{v1} \cup \mathcal{Y}_{v2} \cup \mathcal{Y}_{v3})$, where: **(a)** $\mathcal{Y}_{v1}$ is the *range* of v , i.e., $\mathcal{Y}_{v1} = \{(\mathsf{range}, s_v, e_v)\}$; **(b)** $\mathcal{Y}_{v2}$ is the *depth* of v, i.e., $\mathcal{Y}_{v2} = \{(\mathsf{depth}, d_v)\}$; and **(c)** $\mathcal{Y}_{v3}$ is the *sequel* of v defined as the set of consecutive pairs $\mathcal{Y}_{v3} = \{(\mathsf{sequel}, c_i, c_{i+1}) | i = 1, \ldots, \ell - 1\}$ where $\mathcal{C}_v) = \{c_1, c_2, \ldots, c_\ell\}$ is the alphabetic ordering of the first characters of v's children. He also computes a *node structure accumulator* $t_v = \mathsf{acc}(\mathcal{Y}_{v3})$ (similar to what we explained for suffixes). Finally, for each sequel c_i, c_{i+1}, compute a subset witness $\mathsf{W}_{\mathcal{P},\mathcal{Y}}$ (as defined in Section 2) where $\mathcal{P} = \{(\mathsf{sequel}, c_j, c_{j+1})\}$ and $\mathcal{Y} = \mathcal{Y}_{v1} \cup \mathcal{Y}_{v2} \cup \mathcal{Y}_{v3}$. This will serve to prove that the given sequel of characters are leading characters of consecutive children of v.

Note that, the keywords pos, first, index, range, depth and sequel are used as *descriptors* of the value that is accumulated. Without loss of generality one can view the elements of sets $\mathcal{X}_{ij}$, $\mathcal{Y}_{ij}$, $j \in \{1, 2, 3\}$ as distinct λ-bit strings (each less than the group's prime order $p \in O(2^\lambda)$), simply by applying an appropriate deterministic encoding scheme $\mathbf{r}(\cdot)$. Therefore, when we accumulate the elements of these sets, we are in fact accumulating their *numerical* representation under encoding $\mathbf{r}$. This allows us to represent all accumulated values as distinct elements of $\mathbb{Z}_p$, achieving the necessary domain homogeneity required by the bilinear accumulator.

At the end of this procedure, each suffix $S[i]$ has its suffix accumulation acc_i and its suffix structure accumulation t_i. Also, each node $v \in G$ is associated with its node accumulation acc_v, its node structure accumulation t_v and one subset witness $\mathsf{W}_{\mathcal{P},\mathcal{C}}$ for each consecutive pair of its children. We denote with $\mathcal{V}, \mathcal{S}$ the sets of node and suffix accumulations acc_v and acc_i, respectively. As a final step, the owner builds two accumulation trees $\mathcal{AT}_\mathcal{V}, \mathcal{AT}_\mathcal{S}$, using the bilinear accumulator described by the key pair. Let d_V, d_S be their respective digests. He sends to the server the text

Algorithm 1: setup$(\mathcal{T}, pk, sk)$

```
1.  For suffix $i = 1, \ldots, n$
2.      Compute suffix structure accumulation $t_i$
3.      Compute suffix accumulation $\mathsf{acc}_i$
4.  Build suffix tree $G = (V, E, \mathcal{T}, \Sigma)$
5.  For each node $v \in G$
6.      Compute node structure accumulation $t_v$
7.      Compute node accumulation $\mathsf{acc}_v$
8.      For each consecutive pair of children of $v$
9.          Compute subset witness $\mathsf{W}_{\mathcal{P},\mathcal{C}}$
10. Build accumulation trees $\mathcal{AT}_{\mathcal{V}}, \mathcal{AT}_{\mathcal{S}}$
11. Send $\mathcal{T}, auth(\mathcal{T})$ to the server and publish $\mathsf{pk}, d_{\mathcal{V}}, d_{\mathcal{S}}$
```

$\mathcal{T}$, as well as authentication information $auth(\mathcal{T})$ consisting of the suffix tree G, the two accumulation trees $\mathcal{AT}_{\mathcal{V}}, \mathcal{AT}_{\mathcal{S}}$ and all values $\mathsf{acc}_i, \mathsf{acc}_v, t_i, t_v, \mathsf{W}_{\mathcal{P},\mathcal{C}}$, and publishes $\mathsf{pk}, d_{\mathcal{V}}, d_{\mathcal{S}}$.

Proof generation. We next describe proof generation for pattern matching queries, i.e., matches and mismatches. The process varies greatly for the two cases as can be seen in Algorithm 2 below. The role of each proof component will become evident when we discuss the verification process in the next paragraph. In both cases, let (v, k, t) be the matching node in G, the matching index and the prefix size returned by algorithm $(v, k, t) \leftarrow \mathsf{suffixQuery}(p, G)$ (as described in Section 5.2 and Figure 5·3).

Proving a match. In this case the answer is $\alpha(q) =$ "$\mathsf{match\ at}\ i$". Let $p = p_1p_2\ldots p_m$ be the queried pattern. The server computes $i = s_v - d_v$ and $j = i + m$. By Lemma 9, suffixes $S[i]$ and $S[j]$ are such that $S[i] = pS[j]$ and $i \leq j$. The corresponding indexes are easily computable by traversing the suffix tree for p. The server returns i, j along with characters $\mathcal{T}[i], \mathcal{T}[j]$ as well as suffix structure and suffix accumulations $\mathsf{acc}_i, \mathsf{acc}_j, t_i, t_j$. Finally, using accumulation tree $\mathcal{AT}_{\mathcal{S}}$, he computes proofs π_i, π_j for validating that $\mathsf{acc}_i, \mathsf{acc}_j \in \mathcal{S}$.

Proving a mismatch. In this case the answer to the query q is $\alpha(q) =$ "$\mathsf{mismatch}$".

Algorithm 2: query$(q, \mathcal{T}, auth(\mathcal{T}), \mathsf{pk})$

```
1.  Call $\mathsf{suffixQuery}(p, G)$ to receive $(v, k, t)$
2.  Set $i = s_v - d_v$
3.  If $t = m$ Then
4.      Set $a(q) =$ "$\mathsf{match\ at}\ i$" and $j = i + m$
5.      Lookup $\mathsf{acc}_i, \mathsf{acc}_j, t_i, t_j$ in $auth(\mathcal{T})$
6.      Compute $\mathcal{AT}_{\mathcal{S}}$ proofs $\pi_i, \pi_j$ for $\mathsf{acc}_i, \mathsf{acc}_j$
7.      Set $\Pi(q) = (j, \mathcal{T}[i], \mathcal{T}[j], \mathsf{acc}_i, \mathsf{acc}_j, t_i, t_j, \pi_i, \pi_j)$
8.  Else
9.      Set $a(q) =$ "$\mathsf{mismatch}$" and $j = i + k + 1$
10.     Lookup $\mathsf{acc}_v, \mathsf{acc}_i, \mathsf{acc}_j, t_v, t_i, t_j$ in $auth(\mathcal{T})$
11.     Compute $\mathcal{AT}_{\mathcal{S}}$ proofs $\pi_i, \pi_j$ for $\mathsf{acc}_i, \mathsf{acc}_j$
12.     Compute $\mathcal{AT}_{\mathcal{V}}$ proof $\pi_v$ for $\mathsf{acc}_v$
13.     Set $\mathsf{aux} = (s_v, e_v, d_v, i, j, k, t)$
14.     Set $\Pi(q) = (\mathsf{aux}, \mathcal{T}[i], \mathcal{T}[j], \mathsf{acc}_v, \mathsf{acc}_i, \mathsf{acc}_j, t_v, t_i, t_j, \pi_v, \pi_i, \pi_j)$
15.     If $k = e_v$ Then
16.         Traverse the sequels of $v$ to find pair $c, c'$ s.t. $c < p_{t+1} < c'$
17.         Let $\mathcal{P} = \{(\mathsf{sequel}, c, c')\}$
18.         Lookup subset witness $\mathsf{W}_{\mathcal{P},\mathcal{C}}$
19.         Set $\Pi(q) \cup \{\mathcal{P}, \mathsf{W}_{\mathcal{P},\mathcal{C}}\}$
20. Output $a(q), \Pi(q)$
```

Let (s_v, e_v), d_v and c_v be the range, depth and sequel of v. The server computes $i = s_v - d_v$ and $j = i + k + 1$ and returns $s_v, e_v, d_v, k, t, i, j, \mathcal{T}[i], \mathcal{T}[j]$ along with accumulations $\mathsf{acc}_v, \mathsf{acc}_i, \mathsf{acc}_j$ with proofs π_v, π_i, π_j and structure accumulation values t_v, t_i, t_j. Finally, if $k = e_v$ he also returns $\mathsf{W}_{\mathcal{P},\mathcal{C}}$ where $\mathcal{P}$ contains sequel c, c' such that $c < p_{t+1} < c'$.

Verification. Here we describe the verification algorithm of our scheme. Below we provide the pseudo-code in Algorithm 3 and an intuitive explanation for the role of each component of the proof. In both cases, the verification serves to check the conditions stated in Lemmas 9, 10, which suffices to validate that the answer is correct.

Verifying a match. Recall that, by Lemma 9, it suffices to validate that there exist suffixes $S[i], S[j]$ in the text, such that $S[j] = pS[i]$. First the client verifies that $\mathsf{acc}_i, \mathsf{acc}_j \in \Pi(q)$ are indeed the suffix accumulations of two suffixes of $\mathcal{T}$ using proofs π_i, π_j (Line 1). Then, it checks that the corresponding structure accumulations are indeed t_i, t_j (Lines 2-4). It remains to check that the "difference" between them is p (Lines 6-7), by first computing the pattern accumulation value $g^{\mathbf{p}}$) for $\mathbf{p} = \prod_{l=1}^{m}(s + \mathbf{r}(\mathsf{pos}, i + l - 1, p_l))$. A careful observation shows that this is indeed the "missing" value between the honestly computed structure accumulations t_i, t_j. This can be cryptographically checked by a single bilinear equality testing $e(t_i, g) = e(t_j, g^{\mathbf{p}})$. This last step can be viewed as an accumulator-based alternative to chain-hashing using a collision-resistant hash function. It follows from the above that, if all these checks succeed, the conditions of Lemma 9 are met.

Verifying a mismatch. The case of a mismatch is initially similar to that of a match, however, it eventually gets more complicated. The client begins by verifying the same relations as for the case of a match for two indices i, j (Lines 1-4). In this case, these positions correspond to two suffixes $S[i], S[j]$ such that $S[j] = p'S[i]$, where p' is a prefix of p, i.e., their difference is a beginning part of the pattern (Lines 9-10). Unfortunately, this is not enough to validate the integrity of the answer. For example, a cheating adversary can locate the occurrence of such a prefix of p in the text, and falsely report its position, ignoring that the entire p appears in $\mathcal{T}$ as well. We, therefore, need to prove that p' is the maximal prefix of p appearing in the text and here is where the properties of the suffix tree become useful. In particular, if two characters appear consecutively within the same node of G, it must be that every occurrence of the first one in $\mathcal{T}$ is followed by the second one. Hence, if the server can prove that the part of $\mathcal{T}$ corresponding to the final part of p' as well as the consequent character, both fall within the same node and said consequent character is not the one dictated by p, it must be that p' truly is the maximal prefix of $p \in \mathcal{T}$.

Algorithm 3: verify$(q, \alpha(q), \Pi(q), d, \mathsf{pk})$

```
1.  Verify $\mathsf{acc}_i, \mathsf{acc}_v$ with respect to $d_{\mathcal{S}}$, with $\pi_i, \pi_j$
2.  Compute $g^{\mathbf{x}}$ for $\mathbf{x} = (s + \mathbf{r}(\mathsf{first}, \mathcal{T}[i]))(s + \mathbf{r}(\mathsf{index}, i))$
3.  Compute $g^{\mathbf{y}}$ for $\mathbf{y} = (s + \mathbf{r}(\mathsf{first}, \mathcal{T}[j]))(s + \mathbf{r}(\mathsf{index}, j))$
4.  Verify that $e(t_i, g^{\mathbf{x}}) = e(\mathsf{acc}_i, g)$ and $e(t_j, g^{\mathbf{y}}) = e(\mathsf{acc}_j, g)$
5.  If $\alpha(q) =$ "match at $i$" Then
6.      Compute $g^{\mathbf{p}}$ for $\mathbf{p} = \prod_{l=1}^{m}(s + \mathbf{r}(\mathsf{pos}, i + l - 1, p_l))$
7.      Verify that $e(t_i, g) = e(t_j, g^{\mathbf{p}})$
8.  Else
9.      Compute $g^{\mathbf{p}}$ for $\mathbf{p} = \prod_{l=1}^{t}(s + \mathbf{r}(\mathsf{pos}, i + l - 1, p_l))$
10.     Verify that $e(t_i, g^{\mathbf{p}}) = e(t_j, g)$
11.     Verify that $i = s_v - d_v$ and $s_v \leq k \leq e_v$ and $j = i + k + 1$
12.     Verify $\mathsf{acc}_v$, with respect to $d_{\mathcal{V}}$, with $\pi_v$
13.     Compute $g^{\mathbf{z}}$ for $\mathbf{z} = (s + \mathbf{r}(\mathsf{range}, s_v, e_v))(s + \mathbf{r}(\mathsf{depth}, d_v))$
14.     Verify that $e(t_v, g^{\mathbf{z}}) = e(\mathsf{acc}_v, g)$
15.     If $k < e_v$ Then verify that $p_{t+1} \neq \mathcal{T}[j]$
16.     Else
17.         Verify that $c < p_{t+1} < c'$ (alphabetically)
18.         Compute $g^{\mathbf{w}}$ for $\mathbf{w} = s + \mathbf{r}(\mathsf{sequel}, c, c')$
19.         Verify that $e(\mathcal{W}_{\mathcal{P},\mathcal{C}}, g^{\mathbf{w}}) = e(\mathsf{acc}_v, g)$
20. If any check fails Then output reject, Else accept
```

This is done by checking the relation between the node accumulation and the node structure accumulation of the returned node v (Lines 11-15).

This however does not cover the case where the consequent character, after p', falls within a child node of v (i.e., the part of $\mathcal{T}$ corresponding to p', ends at the end of the range of v). To accommodate for this case, the server needs to prove that the next character in p, does not appear as the leading character of any of v's children. Since all these characters have been alphabetically ordered and accumulated in consecutive pairs, it suffices to return the corresponding pair $\mathcal{P}$ that "covers" this consequent character. The validity of this pair is guaranteed by providing the related pre-computed witness, the relation of which to node v is tested by checking a bilinear equality (Lines 17-19).

We can now state and prove our main result.

Theorem 4. *The scheme* $\{$**genkey**,**setup**,**query**,**verify**$\}$ *is a* static *ADS for the class of pattern matching queries q over a text $\mathcal{T}$ of size polynomial in λ. It is* correct *and* secure *under the q-**SBDH** assumption.*

Proof: The correctness of our scheme follows from close inspection of the algorithms. Assume now that there exists a poly-size adversary $\mathcal{A}$ that outputs a winning tuple for the ADS game. We will show that under the q-SBDH assumption this can only happen with negligible probability.

Let $\mathcal{T}, auth(\mathcal{T}), d_{\mathcal{V}}, d_{\mathcal{S}}, q, \alpha^*(q), \Pi(q)$ be the tuple output by $\mathcal{A}$ at the end of the ADS game. We argue separately for the match and mismatch case specified by $\alpha * (q)$:

Match In this case, $\mathcal{A}$ falsely claims that pattern p, specified by query appears in $\mathcal{T}$ at position i. In reality however, p does not appear in $\mathcal{T}$ at that position. Note that this covers both the case where p does not appear in $\mathcal{T}$ at all, and the case where it appears at $i' \neq i$. By Lemma 4, the values $\mathsf{acc}_i, \mathsf{acc}_j$ included in $\Pi(q)$ are actual accumulation values for the suffixes at positions i, j in $\mathcal{T}$, with all but negligible probability. Assume that the value $\mathcal{T} * [i]$ in $\Pi(q)$ is different than the character $\mathcal{T}[i]$ from the actual text. Let $S_i = \mathcal{X}_{i1} \cup \mathcal{X}_{i2} \cup \mathcal{X}_{i3}$ be the accumulated set for acc_i, efficiently computable from $\mathcal{T}$. Then for the set $R = \{\mathbf{r}(\mathsf{first}, \mathcal{T}^*[i]), \mathbf{r}(\mathsf{index}, i)\}$ it is true that $R \not\subseteq S_i$, yet $e(t_i, g^{(s+\mathbf{r}(\mathsf{first},\mathcal{T}^*[i]))(s+\mathbf{r}(\mathsf{index},i))}) = e(t_i, g^{R(s)}) = e(\mathsf{acc}_i, g)$. However, from Lemma 1, this can only happen with negligible probability, therefore it must be that $\mathcal{T}^*[i] = \mathcal{T}[i]$, and similarly for $\mathcal{T}^*[j] = \mathcal{T}[j]$. Likewise, from the fact that the above equation holds, as verification succeeds, it follows that indeed the term t_i from $\Pi(q)$ can be written as $t_i = \mathsf{acc}(\mathcal{X}_{i1})$, where $\mathcal{X}_{i1} = \{(\mathsf{pos}, i, \mathcal{T}[i]), \ldots, (\mathsf{pos}, n, \mathcal{T}[n])\}$ is the correct set for the i-the suffix in $\mathcal{T}$, and likewise, $t_j = \mathsf{acc}(\mathcal{X}_{j1})$.

Observe that, by construction $\mathcal{X}_{i1} \subset \mathcal{X}_{j1}$. Therefore, by Lemma 7, it must be that the term $g^{\mathbf{p}}$ is the accumulation of their set difference. Equivalently, $S[i] = pS[j]$, which from Lemma 9 means that p appears in $\mathcal{T}$ at i, which contradicts our original assumption. From the above analysis, under the q-SBDH, this case can occur only with negligible probability.

Mismatch In this case, $\mathcal{A}$ falsely claims that pattern p, specified by query q, does not appear in $\mathcal{T}$. By Lemma 4, the values $\mathsf{acc}_v, \mathsf{acc}_i, \mathsf{acc}_j$ included in $\Pi(q)$ are actual accumulation values for node v and for the suffixes at positions i, j in $\mathcal{T}$, with all but negligible probability. Using the same analysis as for the match case above, it follows that $\mathcal{T}^*[i] = \mathcal{T}[i]$ and likewise for $\mathcal{T}^*[j]$, with all but negligible probability.

Now, we need to separately inspect two sub-cases, namely whether the mismatch occurred at the end of a node or not. For the former case, using the same analysis as for the match case above, it follows that s_v, e_v, d_v are the correct values for node v with all but negligible probability. Moreover, it must also be that $S[i] = p'S[j]$ where p' is the length t prefix of the queried pattern p, i.e., p' appears in $\mathcal{T}$ at position i. Since $s_v \leq k \leq e_v$, it follows that all occurrences of p' are followed by $\mathcal{T}^*[j]$. Finally, since verification succeeds, $p_{t+1} \neq \mathcal{T}^*[j] = \mathcal{T}[j]$. From Lemma 10, this means that p does not appear in $\mathcal{T}$, contradicting our original assumption. From the above analysis, under the q-SBDH, this case can occur only with negligible probability.

For the latter case, from Lemma 1 the sequel $c, c' \in \Pi(q)$ is a valid sequel of first characters of children of v (i.e., $\mathbf{r}(\mathsf{sequel}, c, c') \in \mathcal{Y}_{v3}$), with all but negligible probability. Therefore, p' is never followed by p_{t+1} in $\mathcal{T}$, and from Lemma 10, this means that p does not appear in $\mathcal{T}$, contradicting our original assumption. From our analysis, under the q-SBDH, this case can occur only with negligible probability.

□

Complexity analysis. The running time of algorithm **setup** is $O(n)$. This follows immediately from the following: (i) the construction of G takes $O(n)$ and the produced tree contains $O(n)$ nodes, (ii) all suffix and suffix structure accumulations can be computed with a single pass over $\mathcal{T}$, (iii) node and node structure accumulation values can be computed in time linear to the number of the node's children (using sk); since each node has a unique parent node, all node accumulations are also computable in time $O(n)$, and (iv) an accumulation tree over n elements can be constructed in time $O(n)$. The running time of algorithm **query** is $O(m)$, because all proof components in $\Pi(q)$ are pre-computed (including $\mathcal{AT}$ proofs if the accumulation trees are of height 1), hence the only costly component is the suffix tree traversal which takes $O(m)$. For algorithm **verify** the runtime is $O(m \log m)$. This holds because verification of $\mathcal{AT}$ proofs can be done with $O(1)$ operations, accumulating a set of m elements, with pk alone, takes $O(m \log m)$ operations and only a constant number of checks is made. The proof consists of a *constant* number of bilinear group elements (at most ten, corresponding to the case of a mismatch at the end of a node). Finally the overall storage space for $auth(\mathcal{T})$ is $O(n)$.

Handling wildcards. Our construction can be easily extended to support pattern matching queries expressed as limited regular expressions. In particular, it can accommodate queries with patterns containing a constant number of "wildcards" (e.g., $*$ or ?). To achieve this we proceed as follows. Partition p into segments associated with simple patterns, with the wildcards falling between them. Proceed to run proof generation and verification for each segment individually. For the mismatch case, it suffices for the server to demonstrate that just one of these segments does not appear in $\mathcal{T}$. For the match case, the server proves existence for all segments and the clients verifies each one separately. He then checks that the positions of occurrence (expressed as the i, j indices of each segment) are "consistent", i.e., they fall in the correct order within the text (or they have the specified distance in case there is a corresponding restriction in the query specification).

5.4 Applications

In this section we discuss two practical applications of our construction. We first show how our scheme can be used to accommodate pattern matching queries over a collection of documents and then explain how our bilinear accumulator authentication technique can be modified to support a class of queries over semi-structured data, namely XML documents. Finally, we discuss how our construction can be extended to efficiently handle modifications in the dataset.

5.4.1 Search on collection of text documents

We generalize our main construction to handle queries over multiple documents. By adding some modifications in the suffix tree authentication mechanism, we build a scheme that supports queries of the form "return all documents that contain pattern p". This enhancement yields a construction that is closer to real-world applications involving querying a corpus of textual documents.

Let $\mathcal{T}_1, \ldots, \mathcal{T}_\tau$ be a collection of τ documents, with content from the same alphabet Σ. Without loss of generality, assume each of them has length n, and let N be the sum of the lengths of all $\mathcal{T}_i$, i.e., $N = \tau n$. We assume a data structure that upon input a query q, expressed as string pattern p from Σ, returns the index set $\mathcal{I} := \{i | p$ appears in $\mathcal{T}_i\}$, i.e., the indices of all documents that contain the pattern. Using our construction as a starting point, one straight forward solution for authenticating this data structure is to handle each $\mathcal{T}_i$ separately, building and authenticating a corresponding suffix tree. Consequently, in order to prove the integrity of his answer, the server replies with τ separate proofs of our main construction (one for each document) which are separately verified by the client. This approach is clearly not efficient since τ can be very large in practice; shorter proofs are not possible, since a server can cheat simply by omitting the answer for some documents and the client has no way to capture this unless he receives a proof for all of them.

Main idea. We handle all documents as a single document $\mathcal{T} = \mathcal{T}_1 * \mathcal{T}_2 * \ldots * \mathcal{T}_\tau$ expressed as their concatenation, where $*$ is a special character $\notin \Sigma$ marking the end of a document. We define extended alphabet $\Sigma^* = \Sigma \cup \{*\}$ and build a single authenticated suffix tree $G = (V, E, \mathcal{T}, \Sigma^*)$ as in our main construction. Observe now that the query can be reduced to answering a single pattern matching query for p in $\mathcal{T}$, asking for *all* its occurrences (as opposed to our main scheme where we were interested with a single occurrence). This can be easily achieved with the following observation about suffix trees: for a pattern p for which $\mathsf{suffixTree}$ outputs node $v \in G$, the number of occurrences of $p \in \mathcal{T}$, is the number of children of G. For example, in Figure 5·2, the pattern i appears three times in the text, and the pattern mi appears twice.

Construction overview. The above relation can be incorporated in our main construction, by encoding in each node v not a single range (s_v, e_v) but the indices of all these ranges (s_u^v, e_u^v), one for each child node u of v. In fact, the information will consist of triples (i, s_u^v, e_u^v) where i is the index of the document $\mathcal{T}_i$ within which s_v falls. This can performed in time $O(n)$ in three steps. Initially, the owner sets up an efficient dictionary structure with key-value pairs formed by document indices and corresponding starting positions. Then he sets up a suffix tree G for $\mathcal{T}$ and with a post-order traversal computes all ranges for each node (with lookups to the dictionary). He finally runs the setup for our main construction with the modified node information explained above.

Regarding proof generation and verification, we distinguish between the two cases.

If p does not appear in $\mathcal{T}$, then the proof is same as in our basic scheme and the same holds for verification[14]. For the case of positive response, the server must return a proof that consists of three parts: (i) a match proof exactly as in our main construction, with boundaries i, j; (ii) a node accumulation acc_v (with its accumulation tree proof and structure accumulation) for the node v corresponding to p and all its ranges; (iii) the indices of all documents where p appears. With access to all this information, the client verifies that p indeed appears in $\mathcal{T}$, it corresponds to v (because there must exist one range of v that covers position j) and that the returned indices correspond exactly to *all* documents containing p. Observe that the special character $*$ makes it impossible for an adversary to cheat by finding two consecutive documents, the first of which ends with a prefix of p and the second of which begins with the corresponding suffix (as long as $\{*\} \notin p$).

5.4.2 Search on XML documents

We now turn our attention to queries over XML documents. We consider the standard tree-based representation of XML data: An XML document X consists of n elements $e_1, \ldots, e_n$ organized in a hierarchical structure, via sub-element relations, which, in turn, imposes a well-defined representation of X as a tree $\mathcal{X}_T$ having elements as nodes and sub-element relations expressed by edges. Each element has a *label* that distinguishes its data type, *attributes* and corresponding *attribute values* and actual *textual content* (which can be viewed as an additional attribute).[15] We also assume that each element of $\mathcal{X}_T$ is associated with a unique numerical identifier stored as an element attribute. Figure 5·4 provides one such simplified tree-based representation.

[14]The node accumulations must include separately a single range for v (randomly chosen in the case of multiple occurrences) and the collection of all ranges described above.

[15]We do not consider reference attributes relating elements to arbitrary nodes in $\mathcal{X}_T$) or processing instructions.

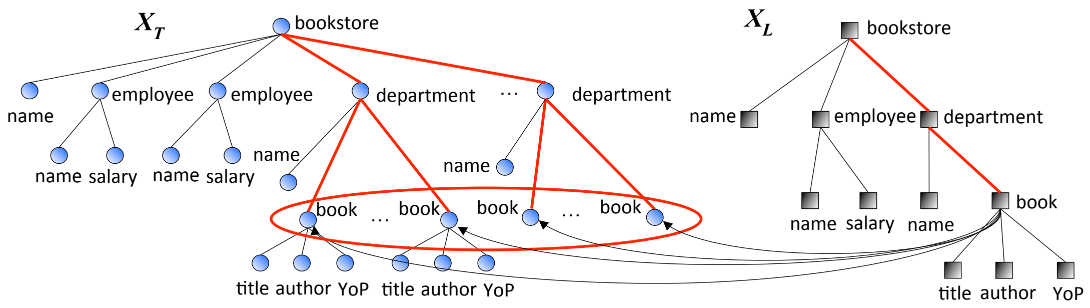


Figure 5·4: (Left) Tree $\mathcal{X}_T$ containing all the elements of XML document X. Element attributes can be included as a different type of node, directly below the corresponding element. (Right) Trie $\mathcal{X}_L$ containing all the distinct label paths that appear at X. Observe how each node has pointers to all the corresponding element nodes in $\mathcal{X}_T$.

Each node e in $\mathcal{X}_T$ is defined (or reachable) by a single *label path* that is the concatenation of the labels of e's ancestor nodes in the order that they must be traversed starting from the root of $\mathcal{X}_T$ in order to reach e. In general, many elements may share the same label path. We abstract away the details of the (often elaborate) querying process of an XML document by considering generic *path queries* that return a subset of the elements of $\mathcal{X}_T$ (in fact, a forest of subtrees in $\mathcal{X}_T$). A *path query* is generally a regular expression over the alphabet $\mathcal{L}$ of valid labels returning all nodes reachable by those label paths conforming to the query, along with the subtrees in $\mathcal{X}_T$ rooted at these nodes. An *exact path query* is related to a label path L of length m, i.e., $L \in \mathcal{L}^m$, returning the subtrees reachable in $\mathcal{X}_T$ by L. This abstraction fully captures the basic notion of path query as identified in various XML query languages, e.g., XPath, XML-QL. As an example a query of the form \bookstore\department\book will return all the books that appear in $\mathcal{X}_T$ as shown in Figure 5·4 *with* the corresponding subtree of each book element (i.e., nodes title, author, YoP).

Main idea. Similar to the case of text pattern matching, our goal is to identify the relations among the elements of XML document that are sufficient to succinctly certify the correctness of exact-path XML queries. Our main approach is to *decouple locating the queried elements* from *validating their contents*. We achieve this through a direct reduction to our (authenticated) suffix tree construction from the previous section: Given an XML document X in its tree-like representation $\mathcal{X}_T$, we construct a *trie* $\mathcal{X}_L$ that stores *all the distinct label paths that appear in* $\mathcal{X}_T$. Compared to our main scheme, $\mathcal{X}_L$ can be viewed as an *uncompressed suffix tree (trie)* with the alphabet being the element label space $\mathcal{L}$ associated with X and the "text" over which it is defined being all label paths in $\mathcal{X}_T$. Each node in $\mathcal{X}_L$ is associated with a valid label path according to $\mathcal{X}_T$ and also with the set of elements in $\mathcal{X}_T$ that are reachable by this label path, through back pointers. For example, the query \bookstore\department\book in Figure 5·4 will reach one node in $\mathcal{X}_L$ which points back to the elements reachable by the queried path. We define, encode and authenticate three types of certification relations (corresponding to edges in Figure 5·4):

1. *Subtree contents:* This relation maps nodes in $\mathcal{X}_T$ with the elements (and their attributes) in $\mathcal{X}_T$ that belong in the subtrees in $\mathcal{X}_T$ defined by these nodes.

2. *Label paths*: This relation maps nodes in $\mathcal{X}_L$ with their corresponding label paths. Here, we make direct use of our results from Section 5.3; however, since we no longer have a tree defined over all possible suffixes of a text, suffix accumulations are no longer relevant (instead, we use node accumulations).

3. *Element mappings:* This relation maps nodes in $\mathcal{X}_L$ with the corresponding elements in $\mathcal{X}_T$ that are reachable by the same label path (associated with these nodes).

We next describe how to cryptographically encode the above relations by carefully computing accumulations or hash values over sets of data objects related to the nodes in $\mathcal{X}_L$ and $\mathcal{X}_T$.

Notation. We denote by e_{id} the identifier, by $A_e = \{(a_i, \beta_i) | i = 1, \ldots, |A_e|\}$ the attribute values, by $lb(e)$ the label, and by $C_e = \{c_i | i = 1, \ldots, |C_e|\}$ the children of element e in $\mathcal{X}_T$. Also, for node $v \in \mathcal{X}_L$, we denote by L_v, $lb(v)$, C_v, and E_v its label path, label, children set, and respectively the set of elements e_i in $\mathcal{X}_T$ that are reachable by L_v. Finally, let d be the height of $\mathcal{X}_T$.

Subtree labels. Subtree contents in $\mathcal{X}_T$ are encoded using a special type of node-specific values. If h is a cryptographic collision-resistant hash function, then for any $e \in \mathcal{X}_T$ we let h_e denote the *hash content* of e $h_e = h(e_{id} \| (a_1, \beta_1) \| \ldots \| (a_{|A_e|}, \beta_{|A_e|}))$. Then, for $e \in \mathcal{X}_T$ we define two different ways for recursively computing node-specific *subtree labels* $\mathsf{sl}(e)$ that *aggregate the hash labels of all the descendant nodes of* e *in* $\mathcal{X}_T$:

1) Hash based: If e is leaf in $\mathcal{X}_T$ then $\mathsf{sl}_1(e) = h_e$, otherwise $\mathsf{sl}_1(e) = h(h_e \| \mathsf{sl}_1(c_1) \ldots \| \mathsf{sl}_1(c_{|C_e|}))$;

2) Accumulation based: $\mathsf{sl}_2 = \mathsf{acc}(\mathcal{Z}_e)$ where if e is leaf then $\mathcal{Z}_e = h_e$, otherwise $\mathcal{Z}_e = \{h_e, \mathcal{Z}_{c_1}, \ldots, \mathcal{Z}_{c_{|C_e|}}\}$;

Node accumulations. Label paths and element mappings in $\mathcal{X}_L$ are encoded using node accumulations. We associate with $v \in \mathcal{X}_L$ three sets of data objects: **(a)** The *label path* L_v of v; let $\mathcal{Y}_{v1} = \{(\mathsf{label}, i, l_i) : i = 1, \ldots, |L_v|\}$; **(b)** The *label sequels* of v is $lb(c_1), \ldots, lb(c_{|C_v|})$, the sequence of the alphabetically ordered labels of v's children; let $\mathcal{Y}_{v2} = \{(\mathsf{sequel}, lb(c_i), lb(c_{i+1})) : i = 1, \ldots, |C_v| - 1\}$; and **(c)** The *XML elements hash* is the hash value of the set E_v of elements of $\mathcal{X}_T$ that correspond to L_v; let $\mathcal{Y}_{v3} = \{(\mathsf{hash}, h(\mathsf{sl}(e_1), \ldots, \mathsf{sl}(e_{|E_v|})\}$ (alternatively, this hash can be computed as the accumulation of values $\mathsf{sl}(e_i)$ using a bilinear accumulator). Then the *node accumulation* for node $v \in \mathcal{X}_L$ is defined as $\mathsf{acc}(\mathcal{Y}_{v1} \cup \mathcal{Y}_{v2} \cup \mathcal{Y}_{v3})$.

Construction overview. Here we discuss the operation of the algorithms of our scheme. The **genkey** algorithm is exactly the same as the one for our main construction (plus generating a collision resistant hash function if sl_1 is chosen for subtree labels). For **setup**, the owner first builds $\mathcal{X}_L$, the trie containing all distinct paths appearing in the $\mathcal{X}_T$. He then computes subtree labels $\mathsf{sl}(e)$ for each element $e \in \mathcal{X}_T$ in a bottom up way starting from the leaves and node accumulations acc_v for each node $v \in \mathcal{X}_L$. He also computes for v, two structure accumulations t_v, s_v, the first of which contains only information regarding the labels of its children nodes and the second contains all the node information except for the label path L_v. Moreover, he computes a subset witness for each consecutive pair of children of v (ordered alphabetically based on their label), *exactly* as in the main scheme. He finally builds a single accumulation tree over the set $\mathcal{V}$ of node accumulations acc_v and sends all components to the server.

With respect to **query** we again distinguish the two cases. If the queried path L does not appear in $\mathcal{X}_T$, proof generation is identical to the mismatch case of our main construction. The server simply needs to prove the existence of a prefix of L, and that none of the children of the node v in $\mathcal{X}_L$ corresponding to this prefix has the necessary next label. This is achieved by providing the length of the prefix, the corresponding acc_v (with its accumulation proof), the structure accumulation s_v, and the corresponding pair of children labels with its subset witness. The client, first checks the validity of acc_v, then verifies it corresponds to the given prefix of L using the structure accumulation and, finally checks whether the next label in L is covered by the given label pair, as well as the fact that it is a well-formed pair using the given witness. Observe that, in contrast to our main construction, since $\mathcal{X}_L$ is uncompressed, the mismatch will always happen "at the end" of a node.

If L appears in $\mathcal{X}_T$, the answer consists of all elements e_i in the document that have label paths corresponding to L as well as the subtrees of $\mathcal{X}_T$ that have e_i as roots. Note that, since the result consists of a forest of subtrees, their structure (i.e., the parent-children relations of elements) is also explicitly part of the answer. Proof generation proceeds as follows. If v is the node in $\mathcal{X}_L$ that corresponds to L, the server only needs to provide acc_v (with its accumulation proof) and the structure accumulation t_v. The client first validates that acc_v is a correct node accumulation and then checks that it corresponds to L and all provided elements e_i using the structure accumulation t_v. To achieve the latter, he first computes subset label $\mathsf{sl}(e_i)$ for each element in the answer E_v and their hash value $\eta = h(\mathsf{sl}(e_1), \ldots, \mathsf{sl}(e_{|E_v|}))$. He then computes $g^{\mathbf{x}}$ for $\mathbf{x} = (s + \mathbf{r}(\mathsf{hash}, \eta)) \prod_{i=1}^{|L_v|} (s + \mathbf{r}(\mathsf{label}, i, l_i))$ and finally checks whether $e(g^{\mathbf{x}}, t_v) = e(\mathsf{acc}_v, g)$. This simultaneously validates that v corresponds to L and that *all* elements of $\mathcal{X}_T$ (including subtrees) have been returned. For the latter, observe that sl is a secure cryptographic representation, hence no elements may be omitted.

5.4.3 Dynamic datasets

So far we have only dealt with the case of static datasets, where the data owner outsources the data once, with no further changes. However, in many cases the owner may wish to update the dataset by inserting or removing data. When this occurs, the owner can of course run the entire setup process again, but here we investigate more efficient updates for the two applications presented above.

Collection of text documents. For our scheme we build a single suffix tree on the collection, hence our update efficiency will crucially depend on this data structure's behavior. In practice, a single modification in any of the documents may change the suffix tree entirely and the best we can do for updates is to re-run setup, in time $O(n\tau)$. One way to accommodate updates more efficiently is the following. We first split the documents in $\sqrt{\tau}$ groups, each with $\sqrt{\tau}$ documents, and then run our scheme separately for each group. A given query now decomposes into a separate query for each group. In this setting, an update –in the form of a document insertion or removal– will only cause the re-computation of one of the suffix trees (and the corresponding ADS) in time $O(n\sqrt{\tau})$ instead of $O(n\tau)$. On the other hand, this increases the cost for proof generation/verification and size by a multiplicative $\sqrt{\tau}$ factor, but in settings with frequent updates, this trade-off may be favorable.

XML documents. In this setting, we discuss updates in the form of element insertion or removal from the document, that do not change the structure of the label trie $\mathcal{X}_L$ (i.e. they do not introduce a new label path in the document). Otherwise, we face the same difficulties as in the previous application. We focus on leaf element insertions; in order to insert more than one element (building a new subtree in $\mathcal{X}_T$) the process is repeated accordingly. Updates of this form can be efficiently handled as follows: First, the new element's subtree label is computed and the subtree labels of all its ancestors in $\mathcal{X}_T$ are re-computed. Second, the node accumulation value of the corresponding node $v \in \mathcal{X}_L$ is updated by inserting the subtree label of the new element in the *XML elements hash*. Then, the second structure accumulation and children witnesses for v are updated, and the accumulation tree is updated accordingly.

Let us now calculate the efficiency of the above process. Computing the subtree labels takes $O(d)$ operations and recomputing the node and structure accumulations and children witnesses requires $O(|C_v|)$ exponentiations (assuming the *XML elements hash* is computed with a bilinear accumulator). We stress that $|C_v|$ is the number of distinct labels the siblings of the inserted element have, and not the number of its XML element siblings; for all practical purposes $|C_v|$ can be viewed as a constant. Finally, by the properties of the accumulation tree, the last step can be run in time $O(|\mathcal{X}_L|^\epsilon)$, where $\epsilon \in (0, 1]$ is a chosen parameter. The same holds for the case of element removal. Hence the overall update cost is $O(d + |C_v| + |\mathcal{X}_L|^\epsilon)$, which is much less than the setup cost.

5.5 Parallel algorithms

In this section we show how to derive parallel implementations for the setup algorithms of our schemes. We consider parallelism in the *exclusive-read-exclusive-write* (EREW) model [JáJ92].

Parallel bilinear accumulator setup. Given the trapdoor information s, the accumulation $\mathsf{acc}(\mathcal{X})$ of a set $\mathcal{X}$ can be computed in $O(\log n)$ parallel time using $O(n/\log n)$ processors in the EREW. This can be achieved using an algorithm for summing n terms in parallel (where sum is replaced with multiplication) [JáJ92] .

Parallel suffix products. It is easy to see that suffix accumulations can be computed with the parallel prefix sums algorithm in $O(\log n)$ parallel time using $O(n/\log n)$ processors.

Parallel path prefixes. We now show how to compute *prefix accumulations* on the paths of a tree. For a node v of a rooted tree T of size n, let x_v denote the element stored at v and $\mathsf{path}(v)$ denote the path of T between v and and the root, including the root and v. Let prefix accumulations of tree T, be computed as $\mathsf{accP}_v(T) = g^{\prod_{u \in \mathsf{path}(v)}(s+x_u)}$, for $v \in T$. These prefix accumulations can be computed in $O(\log n)$ parallel time, using $O(n/\log n)$ processors in the EREW model by computing a *suffix accumulation over the Euler tour of* T (using the previous approach), that is appropriately refined to accumulate $(s+x_v)$ modulo p in the exponent when the tour encounters the left side of v and $(s+x_v)^{-1}$ modulo p when the tour encounters the right side of v.

Parallel subtree products. Our authenticated XML tree in Section 5.4 also uses *subtree labels* on a tree T of size n, storing element x_v at node v. If these labels are accumulation based, they can be modeled as $\mathsf{accS}_v(T) = g^{\prod_{w \in \mathsf{subtree}(v)}(s+x_w)}$ for

$v \in T$, where $\mathsf{subtree}(v)$ is the set of nodes contained in the subtree rooted on node v (including v). Note that in order to compute $\mathsf{accS}_v(T)$ for all $v \in T$, it suffices to compute the products $\prod_{w \in \mathsf{subtree}(v)}(s + x_w)$ for all $v \in T$. Such a parallel algorithm running in $O(\log n)$ parallel time using $O(n/\log n)$ processors was originally presented as an application of a method named tree contraction [MR91].

Parallel accumulation trees. Accumulation trees on a set of n elements, originally presented in [PTT15], can also be constructed in parallel. First, partition the elements of the set in $O(n/\log n)$ buckets of size $O(\log n)$ and then compute the accumulations of the buckets with $O(n/\log n)$ processors in $O(\log n)$ parallel time. Next, for a fixed ϵ, build the accumulation tree on top of the $B = n/\log n$ buckets. Specifically, the accumulations ($O(B^{1-\epsilon})$ in total) of all internal nodes (of degree $O(B^{\epsilon})$) at a specific level can be computed independently from one another. Therefore, by the parallel accumulation setup algorithm (presented in the beginning of this section), the accumulation tree can be computed in $O(\log B^{\epsilon}) = O(\log n)$ parallel time using $O(B^{1-\epsilon}B^{\epsilon}/\log B) = O(n/\log n)$ processors, similarly with a Merkle tree.

We conclude that the setup algorithm of our scheme for authenticated pattern matching can be run in $O(\log n)$ parallel time using $O(n/\log n)$ processors, for both presented applications.

5.6 Performance evaluation

In this section, we present an experimental evaluation of our two authenticated pattern matching applications from Section 5.4. All scheme components were written in C++, by building on a core bilinear accumulator implementation [Tre13] developed by Edward Tremel, as well as using library DCLXVI [DCL16] for bilinear pairings, library FLINT [Fli16] for modular arithmetic, Crypto++ [Cry16] for implementing SHA-2, and the pugiXML [Pug16] XML parser. The code was compiled using `g++`

XML document	**size (MB)**	**# of elements**	**# of paths**	**setup (sec)**
SIGMOD	0.5	11,526	11	0.4
Mondial	1	22,423	33	0.7
NASA	23	476,646	95	8.9
XMark	100	2,840,047	514	35.2
DBLP	127	3,332,130	125	68.9
Protein sequence	683	21,305,818	85	381.5

Table 5.1: XML documents used for experiments and setup time.

version 4.7.3 in C++11 mode. Our goal is to measure important quantities related to the execution of our scheme: verification time for the clients, proof generation time for the server, setup time for the data owner, and the size of the produced proof.

Experimental setup. For our collection of documents application, we used the Enron e-mail dataset [KY04] to build collections of e-mail documents (including headers) with total size varying between 10,000 and 1,000,000 characters. We set the public key size to be equal to 10% of the text size at all times (this can be seen as an upper bound on the size of patterns that can be verified). For the exact path XML application, we experimented with five XML documents of various sizes from the University of Washington XML repository [Uow16], as well as a large synthetic XML document generated using the XMark benchmark tool [Sch16]. A list of the documents and their sizes can be found in Table 5.1. Special characters were escaped both in the e-mail documents and the text content within XML elements. For computing subset labels and XML elements hashes within trie nodes, we used the hash-based approach with SHA-2. In both cases, we constructed accumulation trees of height 1 for the authentication of suffix and node accumulations. All quantities were measured ten times and the average is reported.

Working with pairings over elliptic curves. As discussed in Section 2 the bilinear accumulator employs a pairing e defined over two bilinear groups. For simplicity of presentation, we previously defined $e : \mathbb{G} \times \mathbb{G} \rightarrow \mathbb{G}_T$, i.e., both its inputs come from the same group (known in the literature as an *symmetric* pairing). In practice

text size	**setup (sec)**
100	1.4
1,000	10.7
10,000	99.6
100,000	976.5
1,000,000	10,455

(a) setup time

proof type	**proof (KB)**	**optimal (B)**
positive	3.4	435
negative	3.4	435
neg. end node	4	500
xml positive	1.2	178
xml negative	1.7	243

(b) proof size

Table 5.2: Setup cost for text documents and size of proofs.

however, *asymmetric* pairings of the form $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$, where $\mathbb{G}_1, \mathbb{G}_2$ are groups of the same prime order but $\mathbb{G}_1 \neq \mathbb{G}_2$, are significantly faster. The DCLXVI library we use here makes use of such a pairing over an elliptic curve of 256 bits, and offering bit-level security of 128 bits (corresponding to the strong level of 3072-bit RSA signatures according to NIST [BBB$^+$12]). Elements of $\mathbb{G}_2$ (corresponding to *witnesses* in our scheme) are defined over an extension of the field corresponding to elements of $\mathbb{G}_1$(resp. *accumulations*). The former are twice as large as the latter and arithmetic operations in $\mathbb{G}_2$ are roughly 2-3 times slower.

Setup cost. Table 5.2(a) shows the setup time versus the total length of the documents and depicts a strong linear relation between them. This is expected because of the suffix accumulation computations and the fact that the suffix tree has linearly many nodes. The practical cost is quite large (e.g., roughly 3 hours for a text of 1,000,000 characters). However, this operation only occurs once when the outsourcing takes place. For the XML case, Table 5.1 contains the necessary setup time for the documents we tested. The time grows with the size of the document but is quite small in practice, even for very large documents (e.g., a little above 6 minutes for a document of size 683MB). This happens because the crucial quantity is the number of distinct paths in the document (that will form the nodes of $\mathcal{X}_L$), and not the number of elements in the document itself.

Query time. Figures 4·8(a), (b) and (c) show the server's overhead for answer

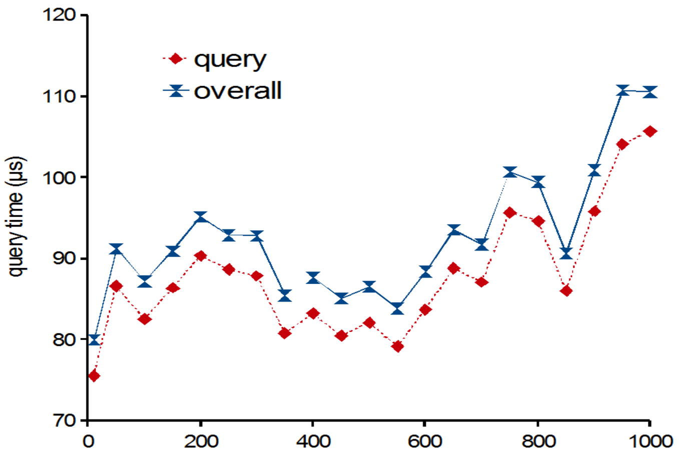


Figure 5·5: Computation time for query evaluation and proof construction at the server, for text pattern matching.

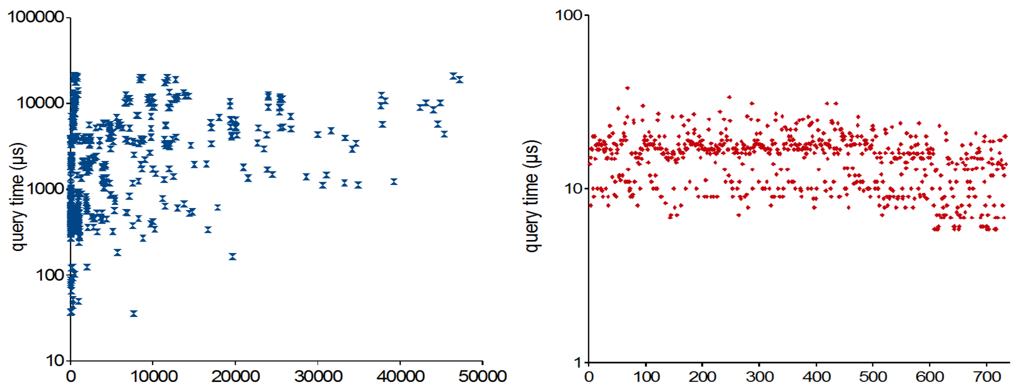


Figure 5·6: Overall computation time for positive (left) and negative (right) responses at the server, for XML pattern matching.

computation and proof generation, for text and XML pattern matching. For text pattern matching we experimented with pattern lengths of 10 to 1,000 characters at a text of 1,000,000 characters. To test the query time at the server, we focused on queries with negative answers and prefix matches finishing at the end of a node, which is the most demanding scenario (e.g., a pattern that starts with a letter that does not even appear in the text is answered by simply looking at the children of the root node of the suffix tree). To produce such queries, we identified matches at ends of various nodes, and "built" progressively larger patterns that ended with them.

We plot the overall time for query evaluation and proof generation versus the size of the found prefix. As can be seen, the cost is in the order of a few microseconds (μs) at all times. In the case of XML queries, we present findings both for the positive and negative case in Figures 4·8(b) and (c) respectively, for the NASA, XMark and DBLP datasets (note the different y-axis scales). For queries with positive answers, we tested on all existing label paths, whereas for negative ones we inserted a “junk” label at a random point along a valid path. In the first case the plot is versus the size of the answer; for the second case where the answer size is zero, we plotted the times across the x-axis by simply assigning an arbitrary id (1-742) to each query. The overhead is again very low, less than 1 millisecond for most instances in the positive case and less than 20μs in the negative. This discrepancy occurs because the server must compile the answer subtrees into a new pugiXML document (that will be sent to the client) for a positive answer –which does not entail any cryptographic operations. Finally, in both applications the plots are quite noisy. This follows because the answer computation time varies greatly with the topology of the trees (in both cases) and the size of node contents (for the XML case).

Comparison with query-evaluation time. In both cases, the server's overhead for proof generation is very low in our scheme since, once the answer is computed, he simply performs a constant number of lookups in his local database to find the corresponding accumulations and witnesses. This is highlighted in Figure 4·8(a) where the lower data series corresponds to the time it takes to simply evaluate the query (without any proof of integrity). As can be inferred, the pure cost for proof generation is less than 10μs at all times. This is also true for the XML case, but due to the different plot type, it was not easy to depict in a figure. In essence, in our scheme the server only performs exactly the same operations as if there was no authentication plus a constant number of memory look-ups, for both applications which makes it ideal for scenarios where a dedicated server needs to handle great workload at line-speed.

Verification time. In Figures 4·8(d),(e) and (f) we demonstrate the verification cost for clients for the text and XML pattern matching applications. In the first case, the time is measured as a function of the queried pattern length (or matching prefix in the case of a negative answer) and in the second as a function of the answer size (as before, for negative responses we plot versus an arbitrary id).

To test the verification time for our text application, we report findings for all three possible cases (match, mismatch and mismatch at end of node). We observe a strong linear correlation between the verification time and the length of the matched pattern. This follows because the main component of the verification algorithm is computing the term $g^{\mathbf{z}}$. Observe that verification for the positive case of a match is slightly faster, which corresponds to our protocol description. In that case, the client needs to perform operations over accumulations and witnesses related only to suffixes, without getting involved with suffix tree nodes. On the other hand, the case where a mismatch occurs at the end of a suffix tree node is slightly more costly than that of a simple mismatch since the client needs to also verify a received sequel with a corresponding witness. The verification overhead remains below 300ms even for arguably large pattern sizes consisting of up to 1,000 characters.

For XML path matching, we report findings for answer sizes of up to 50,000 elements. Observe again the strong linear correlation between the answer size and the verification time, for positive answers. This follows from the fact that the client performs one hash operation per element in the answer, followed by a constant number of bilinear pairings. The total overhead is very small, less than half a second even for large answer sizes. If the answer is negative (again, note the different y-axis scale) the overhead comes mostly from the fixed number of pairings and is much smaller.

Proof size and optimizations. With the DCLXVI library, bilinear group elements

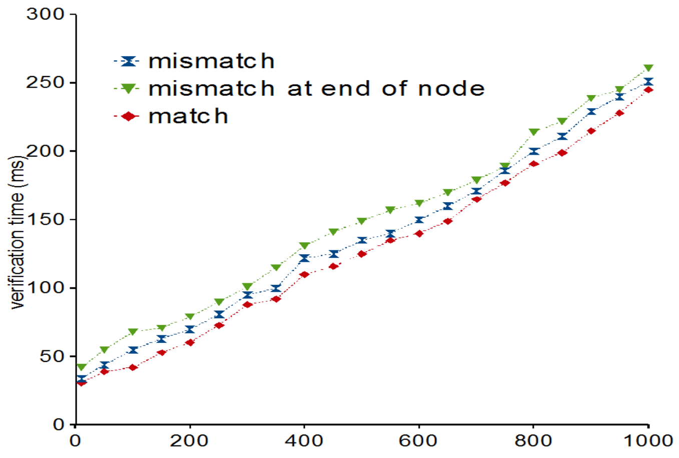


Figure 5·7: Computation time for verification of the three different cases at the client, for text pattern matching.

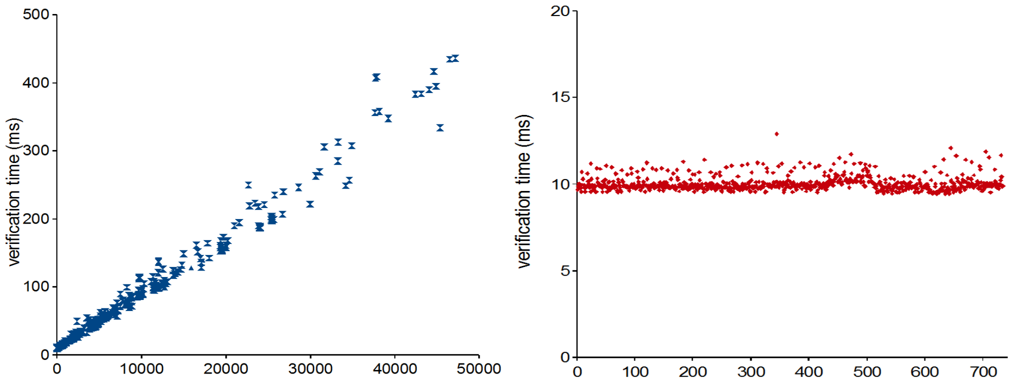


Figure 5·8: Computation time for verification of positive (left) and negative (right) responses at the client, for XML pattern matching.

are represented by their Jacobian coordinates, i.e., three values per element. As described in [NNS10], each coordinate of an element in $\mathbb{G}_1$ is represented by a number of double-precision floating-point variables. The total representation size is 2304 bits for elements of $\mathbb{G}_1$ and 4608 bits for elements of $\mathbb{G}_2$. In our scheme, proofs also contain additional structural information (e.g., position of match/mismatch in text, depth of edge, etc.) which was less than 50 bytes for all tested configurations.

Table 5.2(b) contains the proof sizes produced by our scheme for both applications. Recall that these numbers are independent of dataset, pattern, or answer size.

At all times the proof size is below 4Kb and as low as 1.2Kb for positive XML proofs. While these sizes are very attractive for most applications, further improvements (not implemented here) are possible. Elements can be instead represented by their two affine coordinates (x, y). Moreover, there is no need to transmit y-coordinates as all elements lie on the curve with equation $y^2 = x^3 + 3$, which is part of the public parameters of the scheme. Given x, the y-coordinate can be inferred by a single bit indicating which square root of $x^3 + 3$ it corresponds to. The result of these optimizations can be seen at the third column of the table. The proof size is as low as 435 bytes for text pattern matching and 178 bytes for XML path search. On the other hand, these techniques introduce a small additional overhead at the client (for computing y and transforming to Jacobian coordinates again). When reduced communication bandwidth is essential or proof caching occurs, this extra cost may be acceptable.

Discussion and comparison with alternative schemes. The above results highlight the practicality of our constructions. In particular for the server, who would have to handle the largest workload, the fact that all proof components are pre-computed implies only a small fixed overhead between simply evaluating a query and authenticating the answer with a proof on top of that. Verification time is also appealing for most real-world scenarios making our scheme ideal for settings with "thin" clients or even mobile devices. One component of our scheme that can be improved significantly is the one-time setup operation; pre-computing all proof components takes its toll, especially for the text pattern matching application. Finally, while proofs are arguably very short, they can be further compressed by the optimization discussed above.

To the best of our knowledge, the only other known constructions to achieve constant-size proofs rely on general verifiable computation schemes. As discussed previously, state-of-the-art implementations fall under two categories: circuit or RAM-based. For the former, (e.g., [PHGR13]) the proof generation cost is always at least as large as parsing a circuit that has the entire document as input. The latter are asymptotically better than the former, but still incur prohibitive costs for the server. In particular, as shown in [ZPK14], performing a BFS over a graph of roughly 9,000 edges takes 270 hours with [BFR$^+$13b] and 50 hours with [BCG$^+$13] for proof generation. For comparison, in our text pattern matching experiment, we tested patterns of up to 1,000 elements and an alphabet of 256 characters. Assuming a binary search tree at each node for finding children nodes matching the pattern, this corresponds to 8,000 memory reads in the worst case, and proof generation took less than 10μs. A different line of work for authenticated pattern matching is based entirely on cryptographic hashes (e.g, [DGK$^+$04, MND$^+$04, BCF$^+$04]). There is no existing built system for concrete comparison but, due to the different nature of operations, we expect these schemes to have faster setup and slightly better verification time than ours. However, the proofs grow with the pattern size for text pattern matching, and with the size of the *entire* document (in the worst case) for XML queries.

Chapter 6

Conclusion

This thesis studied the problem of verifying computations performed over remotely stored data, a research direction that has received increased attention recently, especially in relation with cloud computing and remote file hosting. We presented three novel authenticated data structures for the specific problems of verifying nested set operations, multi-dimensional range queries, and pattern matching queries. Our constructions aim for efficiency while sacrificing expressiveness (each one only works for a specific type of queries). The experimental evaluations indeed validate this claim, demonstrating the small overhead of our constructions, especially in comparison with existing general-purpose solutions.

For the problem of verifiable set operations, we presented the first function-specific construction that can handle nested operations while achieving proof size and verification cost that are entirely independent of the sizes of intermediate sets. Our core contribution is the development of novel composable "atomic" protocols for the cases of set intersection, union, and difference. Our experimental evaluation highlights the low cost for verification of complex operations, while at the same time keeping the server's cost for proof construction low.

For the problem of multi-dimensional range queries over outsourced databases, and contrary to existing literature, our solution can support queries on any set of attributes with setup cost, proof construction and verification time that are each only linear in the number of database attributes.

The central idea of our methods is the reduction of a multi-dimensional range query to set operations over appropriately pre-defined sets in the database. We provided a detailed asymptotic and empirical performance evaluation, which confirms the feasibility of our scheme.

For the problem of pattern matching queries, we presented a construction that can accommodate queries over text and XML documents with constant-size proofs, using careful encoding of answer-specific certification relations with cryptographic accumulators. We demonstrated the practicality of our schemes by experimenting on real datasets.

While introducing new more efficient constructions for the same types of problems is certainly a challenging research direction, it seems that a more interesting problem has to do with increasing the expressiveness of supported query types. For example, for the first two constructions one natural expansion would be to support aggregation functions on the computed results, e.g., to return only the average of value of a set instead of the entire set. For the pattern matching case, it is of great interest to design a scheme that supports more general query types, such as search for regular expressions on texts and general XPath queries for XML. These expansions may require significant modifications to our constructions, or even development of altogether new schemes.

6.1 Combining function-specific with general-purpose verifiable computation

As discussed previously, a common theme in our solutions is the sacrifice of expressiveness in favor of efficiency. However, the problem with function-specific constructions is that they come with no guarantee of composability. Simply put, given a scheme for a function f_1 and a scheme for a function f_2 there is no known "blackbox" way to construct a scheme for function $g := f_1 \circ f_2$, in a way that preserves the efficiency of the underlying schemes. To highlight the importance of such a mechanism, imagine a nested SQL query consisting of a `JOIN` query over two tables, followed by a `SELECT ... WHERE` query on the join result. Assuming the existing of two schemes for the two basic query types (e.g., [YPPK09a] and our construction from Chapter 4, respectively), can we build an efficient construction for their composition? Unfortunately, non-composability is almost inherent in this setting, as function-specific VC schemes may utilize entirely different types of data encodings and cryptographic tools. Therefore, it is not trivial to translate the output of one to appropriately encoded input for another.

One way to answer this is to compose our function-specific schemes with general-purpose schemes based on SNARKs (e.g., [PHGR13, BCG$^+$13]). That is, we can consider a hybrid scheme where the server proves using a SNARK that it knows a proof for a function-specific scheme, for the validity of its answer. The security of the hybrid scheme can be shown by a reduction to the security of our scheme based on the knowledge property of the SNARK (via a standard composition argument [BCCT13]). The advantages of using such a hybrid scheme may be multiple. For example, the proof size is constant and independent of the size of the query (this follows from the succinctness of the SNARK), which is not always guaranteed with an ADS (see, for example, our construction from Chapter 3). Moreover, the hybrid scheme might be much more efficient than simply using a generic solution in terms of server computation. This saving is due to the server using the SNARK only to certify a small computation (involving only the small verification circuit of the ADS scheme). Moreover, most of the server's work, when answering the query, is certified using the efficient function-specific scheme that is tailored to the particular problem at hand, resulting in less overhead for the server compared to the generic solution alone. This approach has already been explored in existing works, e.g., in [ZPK14] the authors combine a custom-made ADS for path queries in graphs, with a SNARK prover to increase the class of supported computations. This proof "bootstrapping" process can be seen as a special case of the composition technique of [Val08, BCCT13], which has since been implemented for SNARK composition in [BCTV14, CFH$^+$15].

Another way to achieve a "best-of-both-worlds-result", that balances generality and efficiency, is via the design of *composable* ADS constructions. At first thought, this seems counter-intuitive, as the goal of an ADS scheme is to target a specific problem and build an optimized solution. On the other hand, a composition approach essentially mandates that the proven statement from one part of the computation can be used as the input for proving the next part. However, this goal is not unachievable; indeed, our constructions from Chapters 3 and 4 are of this flavor. The former reduces the problem of proving complex set operations to that of proving a series of simpler ones, while the latter decomposes a multi-dimensional range query to multiple 1–dimensional ones. More recently, this technique has also been applied in the context of more general SQL queries in [ZKP15].

A final direction in this area, that has been explored to a smaller extent, is that of customized fusions of function-specific and general purpose schemes. For example, one way to do this is to integrate a particular ADS with a SNARK system, resulting in a construction that is general-purpose but, more importantly, it is much more efficient than other general-purpose schemes for the particular type of computation associated with the underlying ADS. This approach has been explored in [KPP$^+$14] that utilizes our set encoding technique from Chapter 3 with the SNARK of [GGPR13, PHGR13] to build a SNARK that is based on set circuits, a natural generalization of arithmetic circuits. The result is a general-purpose VC scheme that is significantly more efficient than the implementation of [PHGR13] for computations that involve set operations.

6.2 Other open problems

One important advantage of SNARK-based VC schemes is that they can be easily modified to ensure that the proof reveals nothing about the dataset beyond what is directly inferable from the result itself. This property is known to as zero-knowledge and it is particularly useful in cases where the server contributes to the computation with a (possibly sensitive) dataset of his own. With all existing SNARK-based schemes, this property can be achieved almost for free with a simple proof randomization technique. On the other hand, existing function-specific schemes do not have this property and there is no similar straight forward way to modify them. One notable exception to this is a recent line of works [GOT15, GOP$^+$15, GGOT15, GNP$^+$15], that propose the notion of zero-knowledge ADS schemes that achieve the same level of privacy as SNARK-based solutions. In particular, [GOP$^+$15] extends our construction from Chapter 3 in a privacy-preserving manner, albeit for a single operation. Devising new private function-specific VC schemes is an open and challenging problem that may require the development of novel cryptographic tools.

Another interesting problem has to do with the type of assumptions upon which the security of VC schemes is based. All SNARK-based schemes need to rely on non-falsifiable assumptions for security, as proven in [GW11], which is not necessarily true for ADS schemes. However utilizing such assumptions can yield more efficient solutions (see, for example, out construction from Chapter 3 and [ZKP15]). It is worth exploring alternatives techniques and possibly relaxing the security or interaction requirements of the model, in order to develop new constructions for both types of schemes. Two possible relaxations, that may yield better solutions, focus on proving security in the random oracle model or devising interactive protocols, i.e., constructions where client and server participate in a multi-round interaction to validate the integrity of the result.

References

Giuseppe Ateniese, Emiliano De Cristofaro, and Gene Tsudik. (If) size matters: Size-hiding private set intersection. In *Public Key Cryptography - PKC 2011 - 14th International Conference on Practice and Theory in Public Key Cryptography, Taormina, Italy, March 6-9, 2011. Proceedings*, pages 156–173, 2011.

Miklós Ajtai. Generating hard instances of lattice problems (extended abstract). In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, 1996.

Bao-Ling Adam, Yinsheng Qu, John W Davis, Michael D Ward, Mary Ann Clements, Lisa H Cazares, O John Semmes, Paul F Schellhammer, Yutaka Yasui, Ziding Feng, et al. Serum protein fingerprinting coupled with a pattern-matching algorithm distinguishes prostate cancer from benign prostate hyperplasia and healthy men. *Cancer research*, 62(13):3609–3614, 2002.

Dan Boneh and Xavier Boyen. Short signatures without random oracles and the SDH assumption in bilinear groups. *Journal of Cryptology*, 21(2):149–177, 2008.

Elaine Barker, William Barker, William Burr, William Polk, and Miles Smid. NIST recommendation for key management Part 1: General (revision 3), July 2012.

Gilles Brassard, David Chaum, and Claude Crépeau. Minimum disclosure proofs of knowledge. *Journal of Computer and System Sciences*, 37(2):156–189, 1988.

Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *Innovations in Theoretical Computer Science 2012, Cambridge, MA, USA, January 8-10, 2012*, pages 326–349, 2012.

Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. Recursive composition and bootstrapping for snarks and proof-carrying data. In *Proceedings of the Forty-fifth Annual ACM Symposium on Theory of Computing*, STOC '13, pages 111–120, New York, NY, USA, 2013. ACM.

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(10):1263–1278, 2004.

Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. Snarks for C: verifying program executions succinctly and in zero knowledge. In *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II*, pages 90–108, 2013.

Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 459–474, 2014.

Nir Bitansky, Ran Canetti, Omer Paneth, and Alon Rosen. On the existence of extractable one-way functions. In *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 505–514, 2014.

Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. In *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II*, pages 276–294, 2014.

Josh Cohen Benaloh and Michael de Mare. One-way accumulators: A decentralized alternative to digital sinatures (extended abstract). In *Advances in Cryptology - EUROCRYPT '93, Workshop on the Theory and Application of of Cryptographic Techniques, Lofthus, Norway, May 23-27, 1993, Proceedings*, pages 274–285, 1993.

Manuel Blum, William S. Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the correctness of memories. *Algorithmica*, 12(2/3):225–244, 1994.

E. R. Berlekamp. Factoring polynomials over large finite fields*. In *Proceedings of the second ACM symposium on Symbolic and algebraic manipulation*, SYMSAC '71, pages 223–, New York, NY, USA, 1971. ACM.

Michael Backes, Dario Fiore, and Raphael M. Reischuk. Verifiable delegation of computation on outsourced data. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 863–874, 2013.

Benjamin Braun, Ariel J. Feldman, Zuocheng Ren, Srinath T. V. Setty, Andrew J. Blumberg, and Michael Walfish. Verifying computations with state. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 341–357, 2013.

Siavosh Benabbas, Rosario Gennaro, and Yevgeniy Vahlis. Verifiable delegation of computation over large datasets. In *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, pages 111–131, 2011.

Kevin D. Bowers, Catherine Hart, Ari Juels, and Nikos Triandopoulos. Pillarbox: Combating next-generation malware with fast forward-secure logging. In *Research in Attacks, Intrusions and Defenses - 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings*, pages 46–67, 2014.

Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990.*, pages 322–331, 1990.

Mihir Bellare and Adriana Palacio. The knowledge-of-exponent assumptions and 3-round zero-knowledge protocols. In *Advances in Cryptology - CRYPTO 2004, 24th Annual International CryptologyConference, Santa Barbara, California, USA, August 15-19, 2004, Proceedings*, pages 273–289, 2004.

Elette Boyle and Rafael Pass. Limits of extractability assumptions with distributional auxiliary input. In *Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part II*, pages 236–261, 2015.

Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *CCS '93, Proceedings of the 1st ACM Conference on Computer and Communications Security, Fairfax, Virginia, USA, November 3-5, 1993.*, pages 62–73, 1993.

Paul G. Brown. Overview of scidb: large scale array storage, processing and analysis. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 963–968, 2010.

Dan Boneh, Amit Sahai, and Brent Waters. Functional encryption: Definitions and challenges. In *Theory of Cryptography - 8th Theory of Cryptography Conference, TCC 2011, Providence, RI, USA, March 28-30, 2011. Proceedings*, pages 253–273, 2011.

Dan Boneh and Brent Waters. Conjunctive, subset, and range queries on encrypted data. In *Theory of Cryptography, 4th Theory of Cryptography Conference, TCC 2007, Amsterdam, The Netherlands, February 21-24, 2007, Proceedings*, pages 535–554, 2007.

Gordon V. Cormack and Andrej Bratko. Batch and online spam filter comparison. In *CEAS 2006 - The Third Conference on Email and Anti-Spam, July 27-28, 2006, Mountain View, California, USA*, 2006.

Craig Costello, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. Geppetto: Versatile verifiable computation. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 253–270, 2015.

Kai-Min Chung, Yael Tauman Kalai, Feng-Hao Liu, and Ran Raz. Memory delegation. In *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, pages 151–168, 2011.

Jan Camenisch and Anna Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *Advances in Cryptology - CRYPTO 2002, 22nd Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 2002, Proceedings*, pages 61–76, 2002.

Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, 2002.

Sanjit Chatterjee and Alfred Menezes. On cryptographic protocols employing asymmetric pairings - the role of Ψ revisited. *Discrete Applied Mathematics*, 159(13):1311–1322, 2011.

Hong Chen, Xiaonan Ma, Windsor W. Hsu, Ninghui Li, and Qihua Wang. Access control friendly query verification for outsourced data publishing. In *Computer Security - ESORICS 2008, 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings*, pages 177–191, 2008.

Ran Canetti, Omer Paneth, Dimitrios Papadopoulos, and Nikos Triandopoulos. Verifiable set operations over outsourced databases. In *Public-Key Cryptography - PKC 2014 - 17th International Conference on Practice and Theory in Public-Key Cryptography, Buenos Aires, Argentina, March 26-28, 2014. Proceedings*, pages 113–130, 2014.

Ran Canetti, Ben Riva, and Guy N. Rothblum. Refereed delegation of computation. *Information and Computation*, 226:16–36, 2013.

Crypto++ Library, 2016. `http://www.cryptopp.com/`.

Bogdan Carbunar and Radu Sion. Uncheatable reputation for distributed computation markets. In *Financial Cryptography and Data Security, 10th International Conference, FC 2006, Anguilla, British West Indies, February 27-March 2, 2006, Revised Selected Papers*, pages 96–110, 2006.

Weiwei Cheng and Kian-Lee Tan. Query assurance verification for outsourced multidimensional databases. *Journal of Computer Security*, 17(1):101–126, 2009.

Ivan Damgård. Towards practical public key systems secure against chosen ciphertext attacks. In *Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1991, Proceedings*, pages 445–456, 1991.

DCLXVI Library, 2016. `http://cryptojedi.org/`.

Ivan Damgård, Sebastian Faust, and Carmit Hazay. Secure two-party computation with low communication. In *Theory of Cryptography - 9th Theory of Cryptography Conference, TCC 2012, Taormina, Sicily, Italy, March 19-21, 2012. Proceedings*, pages 54–74, 2012.

Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, January 2008.

Premkumar Devanbu, Michael Gertz, April Kwong, Chip Martel, Glen Nuckolls, and Stuart Stubblebine. Flexible authentication of XML documents. *Journal of Computer Security*, 6:841–864, 2004.

Cynthia Dwork, Moni Naor, Guy N. Rothblum, and Vinod Vaikuntanathan. How efficient can memory checking be? In *Theory of Cryptography, 6th Theory of Cryptography Conference, TCC 2009, San Francisco, CA, USA, March 15-17, 2009. Proceedings*, pages 503–520, 2009.

Pavel Emeliyanenko. High-performance polynomial GCD computations on graphics processors. In *2011 International Conference on High Performance Computing &amp; Simulation, HPCS 2012, Istanbul, Turkey, July 4-8, 2011*, pages 215–224, 2011.

Carol Friedman, Philip O Alderson, John HM Austin, James J Cimino, and Stephen B Johnson. A general natural-language text processor for clinical radiology. *Journal of the American Medical Informatics Association*, 1(2):161–174, 1994.

Dario Fiore and Rosario Gennaro. Publicly verifiable delegation of large polynomials and matrix computations, with applications. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 501–512, 2012.

Sebastian Faust, Carmit Hazay, and Daniele Venturi. Outsourced pattern matching. In *Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part II*, pages 545–556, 2013.

FLINT Library, 2016. `http://www.flintlib.org/`.

Michael J. Freedman, Kobbi Nissim, and Benny Pinkas. Efficient private matching and set intersection. In *Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004, Proceedings*, pages 1–19, 2004.

Esha Ghosh, Michael T. Goodrich, Olga Ohrimenko, and Roberto Tamassia. Fully-dynamic verifiable zero-knowledge order queries for network data. *IACR Cryptology ePrint Archive*, 2015:283, 2015.

Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings*, pages 465–482, 2010.

Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, pages 626–645, 2013.

Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–208, 1989.

Sharon Goldberg, Moni Naor, Dimitrios Papadopoulos, Leonid Reyzin, Sachin Vasant, and Asaf Ziv. NSEC5: provably preventing DNSSEC zone enumeration. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2014*, 2015.

Esha Ghosh, Olga Ohrimenko, Dimitrios Papadopoulos, Roberto Tamassia, and Nikos Triandopoulos. Zero-knowledge accumulators and set algebra. *IACR Cryptology ePrint Archive*, 2015:404, 2015.

Esha Ghosh, Olga Ohrimenko, and Roberto Tamassia. Zero-knowledge authenticated order queries and order statistics on a list. In *Applied Cryptography and Network Security - 13th International Conference, ACNS 2015, New York, NY, USA, June 2-5, 2015, Revised Selected Papers*, pages 149–171, 2015.

Jens Groth. Short pairing-based non-interactive zero-knowledge arguments. In *Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings*, pages 321–340, 2010.

Jens Groth. On the size of pairing-based non-interactive arguments. In *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II*, pages 305–326, 2016.

Michael T. Goodrich and Roberto Tamassia. *Algorithm design - foundations, analysis and internet examples.* Wiley, 2002.

M. T. Goodrich, R. Tamassia, and A. Schwerin. Implementation of an authenticated dictionary with skip lists and commutative hashing. In *DARPA Information Survivability Conference amp; Exposition II, 2001. DISCEX '01. Proceedings*, volume 2, pages 68–82 vol.2, 2001.

Craig Gentry and Daniel Wichs. Separating succinct non-interactive arguments from all falsifiable assumptions. In *Proceedings of the 43rd ACM Symposium on Theory of Computing, STOC 2011, San Jose, CA, USA, 6-8 June 2011*, pages 99–108, 2011.

Joseph JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.

Joe Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing, May 4-6, 1992, Victoria, British Columbia, Canada*, pages 723–732, 1992.

Ahmed E. Kosba, Dimitrios Papadopoulos, Charalampos Papamanthou, Mahmoud F. Sayed, Elaine Shi, and Nikos Triandopoulos. TRUESET: faster verifiable set computations. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 765–780, 2014.

Sandeep Kumar and Eugene H. Spafford. A pattern matching model for misuse intrusion detection. In *In Proceedings of the 17th National Computer Security Conference*, 1994.

Lea Kissner and Dawn Xiaodong Song. Privacy-preserving set operations. In *Advances in Cryptology - CRYPTO 2005: 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings*, pages 241–257, 2005.

Bryan Klimt and Yiming Yang. The enron corpus: A new dataset for email classification research. In *Machine Learning: ECML 2004, 15th European Conference on Machine Learning, Pisa, Italy, September 20-24, 2004, Proceedings*, pages 217–226, 2004.

Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings*, pages 177–194, 2010.

Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid Reyzin. Dynamic authenticated index structures for outsourced databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 121–132, 2006.

Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid Reyzin. Authenticated index structures for aggregation queries. *ACM Transactions on Information and System Security*, 13(4):32, 2010.

Libsnark Library, 2016. `https://github.com/scipr-lab/libsnark`.

Essam Mansour, Amin Allam, Spiros Skiadopoulos, and Panos Kalnis. ERA: Efficient serial and parallel suffix tree construction for very long strings. *Proceedings of the VLDB Endowment*, 5(1):49–60, 2011.

Ralph C. Merkle. A certified digital signature. In *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, pages 218–238, 1989.

Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi. Authenticated data structures, generically. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 411–424, 2014.

Silvio Micali. Computationally sound proofs. *SIAM Journal on Computing*, 30(4):1253–1298, 2000.

Charles 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, January 2004.

Einar Mykletun, Maithili Narasimha, and Gene Tsudik. Authentication and integrity in outsourced databases. *ACM Transactions on Storage*, 2(2):107–138, 2006.

Gary L. Miller and John H. Reif. Parallel tree contraction, part 2: Further applications. *SIAM Journal on Computing*, 20(6):1128–1147, 1991.

Moni Naor. On cryptographic assumptions and challenges. In *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, pages 96–109, 2003.

Lan Nguyen. Accumulators from bilinear pairings and applications. In *Topics in Cryptology - CT-RSA 2005, The Cryptographers' Track at the RSA Conference 2005, San Francisco, CA, USA, February 14-18, 2005, Proceedings*, pages 275–292, 2005.

Moni Naor and Kobbi Nissim. Certificate revocation and certificate update. *IEEE Journal on Selected Areas in Communications*, 18(4):561–570, 2000.

Michael Naehrig, Ruben Niederhagen, and Peter Schwabe. New software speed records for cryptographic pairings. In *Progress in Cryptology - LATINCRYPT 2010, First International Conference on Cryptology and Information Security in Latin America, Puebla, Mexico, August 8-11, 2010, Proceedings*, pages 109–123, 2010.

Charalampos Papamanthou. Cryptography for efficiency: Authenticated data structures based on lattices and parallel online memory checking. *IACR Cryptology ePrint Archive*, 2011:102, 2011.

Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 238–252, 2013.

HweeHwa Pang and Kyriakos Mouratidis. Authenticating the query results of text search engines. *Proceedings of the VLDB Endowment*, 1:126–137, 2008.

Bryan Parno, Jonathan M. McCune, and Adrian Perrig. *Bootstrapping Trust in Modern Computers*, volume 10 of *Springer Briefs in Computer Science*. Springer, 2011.

Tobias Pulls and Roel Peeters. Balloon: A forward-secure append-only persistent authenticated data structure. In *Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part II*, pages 622–641, 2015.

Dimitrios Papadopoulos, Stavros Papadopoulos, and Nikos Triandopoulos. Taking authenticated range queries to arbitrary dimensions. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 819–830, 2014.

Dimitrios Papadopoulos, Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Practical authenticated pattern matching with optimal proof size. *Proceedings of the VLDB Endowment*, 8(7):750–761, 2015.

F.P. Preparata, D.V. Sarwate, and Illinois University at Urbana-Champaign Coordinated Science Lab. *Computational Complexity of Fourier Transforms Over Finite Fields*. Defense Technical Information Center, 1976.

Charalampos Papamanthou, Elaine Shi, and Roberto Tamassia. Signatures of correct computation. In *Theory of Cryptography: 10th Theory of Cryptography Conference, TCC 2013, Tokyo, Japan, March 3-6, 2013. Proceedings*, pages 222–242, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Authenticated hash tables. In *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, pages 437–448, 2008.

Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Optimal verification of operations on dynamic sets. In *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, pages 91–110, 2011.

Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Authenticated hash tables based on cryptographic accumulators. *Algorithmica*, pages 1–49, 2015.

PugiXML XML parser, 2016. `http://pugixml.org/`.

Deepak Ravichandran and Eduard H. Hovy. Learning surface text patterns for a question answering system. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA.*, pages 41–47, 2002.

Michael s, Manuel Barbosa, Dario Fiore, and Raphael M. Reischuk. ADSNARK: nearly practical and privacy-preserving proofs on authenticated data. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 271–286, 2015.

Srinath T. V. Setty, Benjamin Braun, Victor Vu, Andrew J. Blumberg, Bryan Parno, and Michael Walfish. Resolving the conflict between generality and plausibility in verified computation. In *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013*, pages 71–84, 2013.

Jacob T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM*, 27(4):701–717, 1980.

Albrecht Schmidt. XMark XML benchmark, 2016. `http://www.xml-benchmark.org/`.

Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep K. Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005, Brighton, UK, October 23-26, 2005*, pages 1–16, 2005.

Ahmad-Reza Sadeghi, Thomas Schneider, and Marcel Winandy. Token-based cloud computing. In *Trust and Trustworthy Computing, Third International Conference, TRUST 2010, Berlin, Germany, June 21-23, 2010. Proceedings*, pages 417–429, 2010.

Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of the 13th USENIX Security Symposium, August 9-13, 2004, San Diego, CA, USA*, pages 223–238, 2004.

Roberto Tamassia. Authenticated data structures. In *Algorithms - ESA 2003, 11th Annual European Symposium, Budapest, Hungary, September 16-19, 2003, Proceedings*, pages 2–5, 2003.

Edward Tremel. Real-world performance of cryptographic accumulators. Undergraduate Honors Thesis, Brown University, 2013.

Justin Thaler, Mike Roberts, Michael Mitzenmacher, and Hanspeter Pfister. Verifiable computation with massively parallel interactive proofs. In *4th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'12, Boston, MA, USA, June 12-13, 2012*, 2012.

Roberto Tamassia and Nikos Triandopoulos. Certification and authentication of data structures. In *Proceedings of the 4th Alberto Mendelzon International Workshop on Foundations of Data Management, Buenos Aires, Argentina, May 17-20, 2010*, 2010.

University of Washington XML data repository, 2016. `http://www.cs.washington.edu/research/xmldatasets/`.

Paul Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In *Proceedings of the 5th Conference on Theory of Cryptography*, TCC'08, pages 1–18, Berlin, Heidelberg, 2008. Springer-Verlag.

Victor Vu, Srinath T. V. Setty, Andrew J. Blumberg, and Michael Walfish. A hybrid architecture for interactive verifiable computation. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 223–237, 2013.

Joachim von zur Gathen and Jürgen Gerhard. *Modern computer algebra (2. ed.)*. Cambridge University Press, 2003.

Michael Walfish and Andrew J. Blumberg. Verifying computations without reexecuting them. *Communications of the ACM*, 58(2):74–84, 2015.

Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973*, pages 1–11, 1973.

Riad S. Wahby, Srinath T. V. Setty, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2014*, 2015.

Jia Xu. Authenticating aggregate range queries over dynamic multidimensional dataset. *IACR Cryptology ePrint Archive*, 2010:244, 2010.

Attila Altay Yavuz, Peng Ning, and Michael K. Reiter. Efficient, compromise resilient and append-only cryptographic schemes for secure audit logging. In *Financial Cryptography and Data Security - 16th International Conference, FC 2012, Kralendijk, Bonaire, Februray 27-March 2, 2012, Revised Selected Papers*, pages 148–163, 2012.

Ting-Fang Yen, Alina Oprea, Kaan Onarlioglu, Todd Leetham, William K. Robertson, Ari Juels, and Engin Kirda. Beehive: large-scale log analysis for detecting suspicious activity in enterprise networks. In *Annual Computer Security Applications Conference, ACSAC '13, New Orleans, LA, USA, December 9-13, 2013*, pages 199–208, 2013.

Yin Yang, Dimitris Papadias, Stavros Papadopoulos, and Panos Kalnis. Authenticated join processing in outsourced databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 5–18, 2009.

Yin Yang, Stavros Papadopoulos, Dimitris Papadias, and George Kollios. Authenticated indexing for outsourced spatial databases. *Very Large Data Bases Journal*, 18(3):631–648, 2009.

Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. IntegriDB: Verifiable SQL for outsourced databases. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1480–1491, New York, NY, USA, 2015. ACM.

Yupeng Zhang, Charalampos Papamanthou, and Jonathan Katz. ALITHEIA: Towards practical verifiable graph processing. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, (CCS), AZ, USA, November 3-7, 2014*, pages 856–867, 2014.

Dimitrios (Dimitris) Papadopoulos

Boston University
Computer Science Department
111 Cummington Mall MCS 138
Boston, MA 02215

dipapado@bu.edu
http://cs-people.bu.edu/dipapado

Education

Boston University
Ph.D., Computer Science, 2016
Fields: Applied Cryptography, Secure Cloud Computing
Thesis: Function-specific schemes for verifiable computation

National Technical University of Athens
Diploma, Applied Mathematics, 2010
Thesis: Design and development of application
for the implementation of mixed map labeling algorithms

Research

IBM Research, Zurich
Research Intern (Summer 2014)
Project: Multi-Owner Authenticated Data Structures
Mentor: Christian Cachin

Verisign Labs
Research Intern (Summer 2015)
Project: Implementation of a NSEC5-ready nameserver
Mentor: Duane Wessels

Teaching

Boston University
Teaching Fellow, CS558 Network Security (Spring 2014)
Teaching Fellow, CS237 Probability in Computing (Fall 2012)

Awards & Fellowships

BU Computer Science Research Excellence Award (2015)
BU Computer Science Teaching Fellow Award (2013)
Gatzoyiannis Scholarship (2012-15)
Gerondelis Fellowship for Academic Performance (2012)

Publications

- S. Goldberg, M. Naor, D. Papadopoulos, L. Reyzin, S. Vasant, and A. Ziv. NSEC5: provably preventing DNSSEC zone enumeration. In *22nd Annual Network and Dis-*

tributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2014, 2015.

- D. Papadopoulos, C. Papamanthou, R. Tamassia, and N. Triandopoulos. Practical authenticated pattern matching with optimal proof size. *Proceedings of the VLDB Endowment*, 8(7):750–761, 2015.

- D. Papadopoulos, S. Papadopoulos, and N. Triandopoulos. Taking authenticated range queries to arbitrary dimensions. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 819–830, 2014.

- A. E. Kosba, D. Papadopoulos, C. Papamanthou, M. F. Sayed, E. Shi, and N. Triandopoulos. TRUESET: faster verifiable set computations. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 765–780, 2014.

- R. Canetti, O. Paneth, D. Papadopoulos, and N. Triandopoulos. Verifiable set operations over outsourced databases. In *Public-Key Cryptography - PKC 2014 - 17th International Conference on Practice and Theory in Public-Key Cryptography, Buenos Aires, Argentina, March 26-28, 2014. Proceedings*, pages 113–130, 2014 .

- M. A. Bekos, M. Kaufmann, D. Papadopoulos, and A. Symvonis. Combining traditional map labeling with boundary labeling. In *SOFSEM 2011: Theory and Practice of Computer Science - 37th Conference on Current Trends in Theory and Practice of Computer Science, Nový Smokovec, Slovakia, January 22-28, 2011. Proceedings*, pages 111–122, 2011.