RICE UNIVERSITY

Efficient Tamper-Evident Data Structures for Untrusted Servers

by

Scott Alexander Crosby

A T S
P F
R D

Doctor of Philosophy

A , T C :

Dan S. Wallach, Chair
Associate Professor of Computer Science

Moshe Y. Vardi
Professor in Computational Engineering

Farinaz Koushanfar
Assistant Professor of Electrical and
Computer Engineering

Houston, Texas

December, 2009

ABSTRACT

Efficient Tamper-Evident Data Structures for Untrusted Servers

by

Scott Alexander Crosby

Many real-world applications run on untrusted servers or are run on servers that are subject to strong insider attacks. Although we cannot prevent an untrusted server from modifying or deleting data, with tamper-evident data structures, we can discover when this has occurred. If an untrusted server knows that a particular reply will not be checked for correctness, it is free to lie. Auditing for correctness is thus a frequent but overlooked operation. In my thesis, I present and evaluate new efficient data structures for tamper-evident logging and tamper-evident storage of changing data on untrusted servers, focussing on the costs of the entire system.

The first data structure is a new tamper-evident log design. I propose new semantics of tamper-evident logs in terms of the auditing process, required to detect misbehavior. To accomplish efficient auditing, I describe and benchmark a new tree-based data structure that can generate such proofs with logarithmic size and space, significantly improving over previous linear constructions while also offering a flexible query mechanism with authenticated results.

The remaining data structures are designs for a persistent authenticated dictionary (PAD) that allows users to send lookup requests to an untrusted server and get authenticated answers, signed by a trusted author, for both the current and historical versions of the dataset. Improving on prior constructions that require logarithmic storage and time, I present new classes of efficient PAD algorithms offering constant-sized authenticated answers or constant storage per update. I implement 21 different versions of PAD algorithms and perform a comprehensive evaluation using contemporary cloud-computing prices for computing and bandwidth to determine the most monetarily cost-effective designs.

Acknowledgments

A graduate student career is a long and arduous process with many experiences and many people to thank. I would like to thank my advisor for his knowledge, support, and ideas over my time at Rice University. Some advisors micromanage their students. Dan's style was to encourage good research and let me explore many different research areas. My presentation and writing skills have been greatly improved by his suggestions and feedback.

The security group at Rice gave me many productive discussions. Seth Nielson directed me toward using SWIG and gave me valuable feedback throughout my time here. Tsuen-Wan "Johnny" Ngan and Algis Rudy welcomed me into the group when I arrived. I would also like to thank the other members of the Security group that I have worked with on different projects, including Dan Sandler and Anwis Das. Several of my research ideas were fleshed out in invaluable discussions with the other systems group grad students at Rice, Animesh Nandi, Atul Singh, and Ajay Gulati as well as my past office mates, Shu Du, Kaushik Ram, and Guohui Wang.

Outside of graduate school, Steve Glassman and Vern Paxson gave me internships at Google and ICSI respectively. Tracy Volz gave valuable critiques for my posters. I would also like to thank the CS departmental administrative staff and IT staff for solving the inevitable electronic and bureaucratic bugs I've encountered.

Of course, I would never have gotten to this point without friends and family. My grandparents and parents instilled a lifelong desire for learning, without which I might not have chosen this career. I would like to give a special thanks to Ruth Tancrede, my high school math teacher. I probably would not be here without her encouragement, knowledge, skill with students, and kindness she so generously shared.

Graduate school is stressful and I'd like to thank the Valhalla crowd for keeping me sane, Todd, Colleen, Chris, Jeff, Julie, Lisa, Zack, Lucinda, and everyone else I've met there. A special thanks to a special friend of mine, Jean Ellis for reminding me about the real purpose of life, her invaluable support, and her patience during the last year's crunch.

Contents

Illustrations

Tables

To Mom, Dad, and Dawn

Chapter 1

Introduction

The Internet offers new opportunities for building systems where the user of a computing service may no longer own the computers supplying the service. Users want assurance of correct behavior despite the owner's full control due to their ownership and physical access. In my thesis, I will show that tamper-evident data structures can operate efficiently on untrusted servers with a variety of rich semantics and robust auditing.

There are many examples of systems built on remote or untrusted machines. In peer-to-peer (p2p) technology a large number of users collaboratively build a system out of their individual machines. The recent growth of cloud computing and software-as-a-service offers a new option for storing data "in the cloud" on potentially remote servers whose use is rented. And of course, external or internal security breaches can cause any trusted server to misbehave. Rather than have the security of a system rest upon the correct behavior of the machines it is running on, this thesis uses cryptographic techniques to encourage correct behavior by detecting malicious activity.

In this thesis, I model an insider attacker having full knowledge and administrative control over the server, capable of knowing all cryptographic key material stored on it. This inside attack model subsumes an external attack model. I further assume the Dolev-Yao [1] model, where cryptography is perfect, signatures cannot be forged, and cryptographic hash functions are collision and preimage free. I do not focus on privacy. Depending on the application, privacy may be attained by encrypting data before sending it to an untrusted server. For a further discussion on privacy, please see the related work in Section 2.2.

While it is impossible to prevent the deletion of records on an untrusted server, tampering with stored records can be detected by using cryptographic data structures. In this thesis, I present and evaluate new data structures for efficiently storing changing data on untrusted servers subject to strong insider attacks. The first algorithm I present is a new design for a tamper-evident log that is built by and stored on an untrusted machine and audited for correct behavior by trusted auditors. The second set of algorithms I present are several designs for persistent authenticated dictionaries (PADs). Authenticated dictionaries allow data to be stored in a tamper-evident fashion on an untrusted server and accessed as a key-value data store. Lookups return the answer and a proof of its correctness, signed by the author. PADs extend authenticated dictionaries into supporting queries to previous versions of the dictionary.

There are a wide variety of applications of tamper-evident data structures, including remote backup services, publishing systems, electronic voting systems, banking, price-lists, stock ticker data, forensic records, legal records, timestamping systems, “cloud computing,” fileservers, peer-to-peer computing, swarm downloading protocols such as BitTorrent [2], data aggregation, smartcard storage, outsourced databases, and many others. Tamper-evident algorithms may also detect misbehavior on the author because the author’s signature forces the author to make a commitment, which may be later audited to detect incorrect behavior.

1.1 Tamper evident logging

The first algorithm I present is a new cryptographic data structure for tamper evident logging. There are over 10,000 U.S. regulations that govern the storage and management of data [3, 4]. Many countries have legal, financial, medical, educational and privacy regulations that require businesses to retain a variety of records. Unfortunately, in many organizations, the servers used to create and store these logs are housed within the organization. Malicious users, including insiders with high-level access and the ability to subvert the logging system, may want to perform unlogged activities or tamper with the recorded history. To include these risks, our threat model for tamper-evident logging assumes that the log is built and stored on a completely untrusted server and that users who generate events to be logged may later collude with the logger to tamper with their previously stored events.

Current solutions to prevent tampering include administrative controls or commercial write-once hardware. Cryptographic techniques offer stronger security and can be verified by remote parties. Current semantics for tamper-evident logging assume that the logger behaves correctly until it doesn't, with the goal of detecting any tampering of events stored before the logger misbehaves. Unfortunately, there is no way to know which logged events are and are not valid. Conversely, my proposed semantics are much stronger and assume that the logger is never trusted.

Previous tamper-evident log designs overlook a critical design problem: How exactly tampering is to be discovered. The core of a tamper-evident log is *auditing*. No matter what algorithm is used, *tampering can not be detected unless some trusted auditor is looking for it*. If a server *knows* that any particular output will not be audited for correctness, then the server is free to lie when generating that output. Auditing is therefore a frequent and thus performance-critical operation and its efficiency must be optimized when designing a tamper-evident log.

1.1.1 Contributions

In Chapter 3, I present the *history tree*, a new data-structure for tamper-evident logging, offering efficient random-access and logarithmic overheads. My log design is logarithmic instead of linear in all operations, making frequent auditing feasible, even on very large logs. The history tree offers other useful features, including permitting authorized purges from the log, finding events matching a predicate, and improved robustness against lost, missing, or corrupt data. In my design, I focused on the practicalities, including a design to represent the history tree on write-once append-only storage. I implement the history tree and benchmark it using real-world log traces and have shown that it can insert thousands of events per second and can be scaled to tens of thousands of events per second.

Authorized purging from a tamper evident log and predicate searching are implemented by *Merkle aggregation*, which is a new generic and efficient technique for combining annotations and Merkle trees in a way that lets annotations be checked for correctness and consistency. Merkle aggregation is tamper-evident, allows aggregation to occur on untrusted servers, and is also supported by several of my PAD designs.

1.2 Persistent authenticated dictionaries

The second class of algorithms I investigate are persistent authenticated dictionaries. The simple abstraction of an authenticated dictionary [5, 6] and persistent authenticated dictionary [7,8] can underlie a wide variety of services, ranging from version control systems [9], public key revocation lists [5], stock ticker data, pricing data, and any other situation where one author wants to use cloud services to publish data to multiple consumers or store data remotely when data integrity is the paramount issue. Even with a trusted server, these algorithms are useful whenever data integrity is critical, because of constant possibility of the server being subverted by an external attacker.

An *authenticated dictionary* at its simplest is a data store, stored on an untrusted server. The data store supports key-value lookup operations and can indicate that a key is not present. In a *persistent authenticated dictionary*, the key store changes contents over time, and lookup queries indicate which version or *snapshot* to look at. Explicit versioning, plus an external channel to alert clients to the latest version, will defeat version rollback attacks.

We assume that a trusted author creates the data being stored on the server and uses digital signatures and cryptography to prevent tampering by the server. Clients generate lookup requests and use digital signatures to verify that the returned data is correct. In the case of a publishing system, clients and the author are different. In the case of outsourced storage or backup services, the client and the author may be the same.

The challenge in designing a PAD is how to minimize the costs of updates and lookups as well as minimizing the storage on the server for authentication information. Instead of storing each snapshot as a separate and independent authenticated dictionary, with storage proportional to the number of keys times the number of snapshots, existing work for persistent authenticated dictionaries proposes more efficient techniques based around applicative search tree algorithms [7], with logarithmic proof size and logarithmic storage per update.

PADs based on search trees can be parameterized on the type of search tree used, such as AVL trees, red-black trees, treaps, or skiplists and the techniques used to efficiently store the forest of search trees for the different snapshots. My designs incorporate more efficient representations for storing a persistent search tree that have constant storage per update [10], but have not been previously used for authenticated dictionaries. In addition, this thesis presents a new approach for building a PAD. By individually signing each key and value pair along with some auxiliary metadata, the server can reply to lookup queries with a *constant sized result*, regardless of the number of keys or snapshots in the PAD.

Algorithm cost isn’t measured just in big-O notation. Digital signatures are more expensive than cryptographic hash operations. Serialization overheads will inflate both the runtime, and the size of messages sent over the network. Real-world benchmarks are required in order to determine the actual constant factors. PAD algorithms are designed to run over the network, which means that they incur the costs of both CPU time and bandwidth. Different PAD algorithms also trade off update costs for lookup costs and the ideal algorithm for an application depends on its relative ratio between updates and lookups. By using the prices charged by contemporary cloud-computing providers for bandwidth and CPU time, we simplify our comparison by reducing each algorithm to its *monetary cost* per update and per lookup. We then compares the monetary costs of each PAD algorithm across different lookup to update ratios.

1.2.1 Contributions

Previous PAD designs had logarithmic storage per update, logarithmic proof size, and constant update size. In Chapter 4, I present several new designs for persistent authenticated dictionaries (PADs), first showing how to adapt efficient representations of a persistent search tree to create PAD designs offering constant storage per update and a design using half of the storage of prior techniques. I then present 9 different *tuple PAD* designs based on a new paradigm for designing PADs that trades off a higher update cost for constant-sized results for lookups. Through a series of optimizations, I reduce the number of needed signatures, reduce the storage overheads to constant storage per update, and reduce the communication costs to constant communication per update. One of the more interesting optimizations is the first use of *speculation* in building tamper-evident or authenticated data structures. The author can sign a statement about a future that hasn't happened yet, as long as there is the ability to correct errors in these signed statements.

Chapter 5 contains an evaluation of the different PAD designs, performing both a big-O analysis, and reporting benchmarked results of all 21 algorithms, including CPU usage and message sizes. I compare my new designs with prior approaches. I determine the most efficient balanced tree algorithm among treaps, red-black trees, and skiplists for building an authenticated dictionary. I also identify the most efficient persistency data structure.

The chapter finishes with a comprehensive monetary evaluation across all 21 algorithms. This analysis showed that 5 different algorithms can be the ‘cheapest’, depending on the circumstances. Surprisingly, even with a slow implementation in an interpreted language, it is often more expensive to send the reply to a lookup request than it is to compute it.

This evaluation also measures the update costs, verification costs, and proof sizes of ordinary authenticated dictionaries based on trees or tuples. A regular authenticated dictionary is a special case of a persistent authenticated dictionary when the server purges unneeded data from older versions, thus saving space.

Chapter 2

Background

There are many cryptographic primitives that are designed to detect tampering. The simplest is the digital signature and cryptographic hash. Digital signatures are unforgeable without the private key. Tampering with the signed value should cause the signature verification to fail. A cryptographic *hash* or *digest* function H reduces a variable length input x into a fixed length output or *digest*. Assuming that a cryptographic hash function satisfies certain properties, it tamper evident.

A cryptographic hash function has to fulfill three properties. First, it must be *collision resistant*, which means that it is infeasible for an attacker to generate $H(x) = H(y)$ with $x \neq y$. Second, it must be *pre-image resistant*, meaning that it is difficult to generate x in $y = H(x)$ given only y. Finally, a cryptographic hash must be *second pre-image resistant*, meaning that it is difficult to generate x' with $H(x') = H(x)$, knowing x. Assuming we have such a hash function, we can now demonstrate a simple form of tamper-evidence. Given a trusted hash $y = H(x)$ of some unknown value x, any attempt by an attacker to tamper and supply an alternative input x' can be discovered by checking $y \overset{?}{=} H(x')$. If the attacker were able to find $x'' \neq x$ with $y = H(x'')$, then the hash function H violates its assumed properties.

We can extend this tamper-evidence by having the input x contain other hashes, forming a recursive data structure, the Merkle tree [11]. Each node in the tree has a hash value which is the cryptographic digest of the hashes of its children nodes and any data stored in that node. The root node's hash value then fixes the contents of the entire tree. Figure 2.1

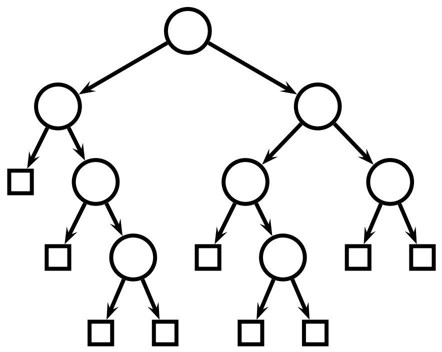

Figure 2.1 : Graphical representation of a Merkle tree. Squares represent null children.

presents a simple Merkle tree.

The benefit of a Merkle tree is that an untrusted server can prove that a particular node is in the tree without sending the entire contents of the tree. Certain subtrees are not necessary in the proof. For instance, in Figure 2.2, to prove that the data in node *A* is in the tree, a server can *stub* out sibling nodes on the path from the root to node *A* by including just their hashes and not the contents, and generate a *pruned tree*. Merkle's paradigm of adding hashes to a binary tree can be applied to any acyclic data structure, including balanced binary trees [5, 6] or skip lists [12]. The only challenge is that mutation of children nodes will invalidate hashes in the parent nodes. Techniques from the world of pure functional programming are often directly applicable, as we will see later.

Merkle-style hashing data structures have been used in smartcard storage [13], outsourced databases [14], distributed filesystems [15–18], graph and geometric searching [19], authenticated responses to XML queries [20], tamper-evident logging [21–23], certificate revocation lists [5], and many others.

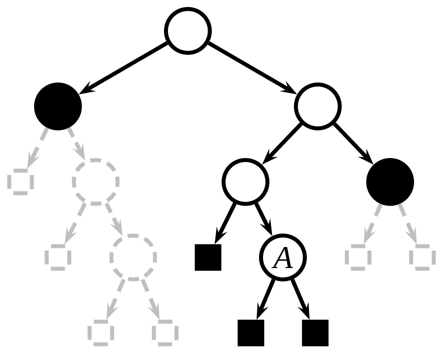

Figure 2.2 : Graphical representation of a pruned Merkle tree. Nodes that are solid discs only have their hash included in the proof. Nodes that are open circles have their key and value included in the proof. Grayed out nodes are omitted.

2.1 Threat models

Tamper evident or authenticated data structures can function in a wide variety of threat models. In cases where the server storing data is not trusted, but the author is, the root of a Merkle tree storing the data can be signed by the trusted author. Responses to lookup queries include a pruned tree. Clients verify the response by using the pruned tree to reconstruct the root hash and checking the signature.

In addition to concerns we might have with untrusted storage, some applications might be concerned with untrusted signers, who might wish to behave incorrectly by, for example, modifying or deleting what they said in the past. Our data structures can be used to prove this misbehavior. If a dishonest server signs inconsistent data structures, those signatures are irrefutable evidence of its misbehavior. There are also applications where the author and server are honest, but may inadvertently suffer corruption or make mistakes in data versioning or when tracking data provenance. Tamper-evidence data structures, combined with auditing, can detect these corruptions.

2.2 Related work

In the space of untrusted servers, there are two broad classes of problems: privacy and integrity. We are not concerned with protecting the secrecy of the data stored on untrusted servers; this can be addressed with external techniques, most likely some form of encryption [24–26]. For instance, in the case of a tamper-evident log, clients can encrypt events before sending them to the logger and in the case of a PAD accessed by trusted clients, the author can mask a key-value pair by encrypting the value and storing it under the cryptographic hash of the key.

Encryption, however, does not hide the access patterns of clients. In private information retrieval (PIR) the goal is lookup privacy, where clients can lookup items stored on a server without the server knowing which item was examined [27]. A naïve approach with linear communications complexity is for clients to download the entire database and then perform the query themselves. Research in PIR focuses on techniques that require sublinear communication [28, 29]. Security can be offered information-theoretically, where security is guaranteed without assuming any hardness results by splitting the database among servers that are assumed to not all conspire with each other. Computational PIR's security rests on problems that are assumed to be computationally intractable [30]. These systems tend to be very expensive, often costing $O(\sqrt{n})$ to perform a query. Related to PIR, oblivious RAM allows a secure processor to use an untrusted external RAM to store its state, while hiding the contents and access patterns [31]. Although not a focus of these algorithms, some of these algorithms offer tamper detection. A full survey of PIR and oblivious ram techniques are beyond the scope of this thesis. See [29] or [32] for recent results.

Although it is impossible to prevent an untrusted server from misbehaving, tampering can be detected through many approaches. Authenticated dictionaries were originally proposed by Naor and Nissim for a public-key-infrastructure certificate revocation system, where a trusted author generates a dictionary of valid certificates and users can inquire as to whether or not a certificate is valid, without trusting the server [5, 6]. They were later extended to support efficient updates [33–35] and queries to older versions [7].

RSA accumulators [36] are a useful way to authenticate a set with a concise $O(1)$ summary, which can be signed using digital signatures. Dynamic accumulators [37] permit efficient incremental update of an accumulator without requiring that it be regenerated. Accumulators have been widely proposed for use in systems such as ours (see, e.g., Goodrich et al. [38, 39]). They do not support proofs of non-membership that are needed to implement a dictionary.

There has been recent interest in creating append-only databases for regulatory compliance. These databases permit the ability to access old versions and trace tampering [40]. A variety of different data structures are used, including a B-tree [41] and a full text index [42]. The security of these systems depends on a write-once semantics of the underlying storage that cannot be independently verified by a remote auditor.

Forward-secure digital signature schemes [43] or stream authentication [44] can be used for signing commitments in our logging scheme or any other logging scheme. Kelsey and Schneier [23] have the logger encrypt entries with a key destroyed after use, preventing an attacker from reading past log entries. A hash function is iterated to generate the encryption keys. The initial hash is sent to a trusted auditor so that it may decrypt events. Logcrypt [45] extends this to public key cryptography.

Ma and Tsudik [46] consider tamper-evident logs built using forward-secure sequential aggregating signature schemes [47, 48]. Their design is round-based. Within each round, the logger evolves its signature, combining a new event with the existing signature to generate a new signature, and also evolves the authentication key. At the end of a round, the final signature can authenticate any event inserted.

Davis et. al. [22] permits keyword searching in a log by trusting the logger to build parallel hash chains for each keyword. Techniques have also been designed for keyword searching encrypted logs [49, 50]. A tamper-evident store for voting machines has been proposed, based on append-only signatures [51], but the signature sizes grow with the number of signed messages [52]. An alternative design of an electronic voting machine store has been proposed that is tamper-evident, history-independent, subliminal free, and designed to function on write-once storage [53].

Many timestamping services have been proposed in the literature. Haber and Stornetta [54] introduce a time-stamping service based on hash chains, which influenced the design of Surety, a commercial timestamping service that publishes their head commitment in a newspaper once a week. Chronos is a digital timestamping service inspired by a skip list, but with a hashing structure similar to our history tree [55]. This and other timestamping designs [56, 57] are round-based. In each round, the logger collects a set of events and stores the events within that round in a tree, skip list, or DAG. At the end of the round the logger publicly broadcasts (e.g., in a newspaper) the commitment for that round. Clients then obtain a logarithmically-sized, tamper-evident proof that their events are stored within that round and are consistent with the published commitment. Efficient algorithms have been constructed for outputting time stamp authentication information for successive events within a round in a streaming fashion, with minimal storage on the server [58]. Unlike these systems, our history tree allows events to be added to the log, commitments generated, and audits to be performed at any time.

Maniatis and Baker [59] introduced the idea of *timeline entanglement*, where every participant in a distributed system maintains a log. Every time a message is received, it is added to the log, and every message transmitted contains the hash of the log head. This process spreads commitments throughout the network, making it harder for malicious nodes to diverge from the canonical timeline without there being evidence somewhere that could be used in an audit to detect tampering. Auditorium [60] uses this property to create a shared “bulletin board” that will record the existance of tampering even when $N-1$ systems are faulty.

Secure aggregation has been investigated as a distributed protocol in sensor networks for computing sums, medians, and other aggregate values when the host doing the aggregation is not trusted. Techniques include trading off approximate results in return for sublinear communication complexity [61], or using MAC codes to detect one-hop errors in computing aggregates [62]. Other aggregation protocols have been based around hash tree structures similar to the ones we developed for Merkle aggregation. These structures combine aggregation and cryptographic hashing, and include distributed sensor-network aggregation protocols for computing authenticated sums [63] and generic aggregation [64]. The sensor network aggregation protocols interactively generate a secure aggregate of a set of measurements. In Merkle aggregation, we use intermediate aggregates as a tool for performing efficient queries. Also, our Merkle aggregation construction is more efficient than these designs, requiring fewer cryptographic hashes to verify an event.

Chapter 3

Secure logging

Audit logs are useful for a variety of forensic purposes, such as tracing database tampering [65] or building a versioned filesystem with verifiable audit trails [17]. Tamper-evident logs have also been used to build Byzantine fault-tolerant systems [66] and protocols [67], as well as to detect misbehaving hosts in distributed systems [68]. Logging systems are required for regulatory compliance and are therefore in wide commercial use (albeit many without much in the way of security features).

Many authenticated or tamper-evident data structures have been proposed for a wide variety of purposes [5–7, 9, 19, 20]. These store data created by a *trusted author* whose signature is used as a root-of-trust for authenticating responses of lookup queries. They thus have no need to detect inconsistencies across versions. For instance, in SUNDR [16], a trusted network filesystem is implemented on untrusted storage. Although version vectors [69] are used to detect when the server presents forking-inconsistent views to clients, only trusted clients sign updates for the filesystem.

Tamper-evident logs are fundamentally different: An *untrusted* logger is the sole author of the log and is responsible for both building and signing it. A log is a dynamic data structure, with the author signing a stream of commitments, a new commitment each time a new event is added to the log. Each commitment *snapshots* the entire log up to that point. If each signed commitment is the root of an authenticated data structure, well-known authenticated dictionary techniques [33–35] can detect tampering *within* each snapshot. However, without additional mechanisms to prevent it, an untrusted logger is free to have different snapshots make *inconsistent claims about the past*. To be secure, a tamper-evident log system must both detect tampering within each signed log *and* detect when different instances of the log make inconsistent claims.

Current solutions for detecting when an untrusted server is making inconsistent claims over time require linear space and time. For instance, to prevent undetected tampering, existing tamper evident logs [22, 23, 70] which rely upon a hash chain require auditors examine every intermediate event between snapshots. One proposal [59] for a tamper-evident log was based on a skip list. It has logarithmic lookup times, assuming the log is known to be internally consistent. However, proving internal consistency requires scanning the full contents of the log. (See Section 3.3.4 for further analysis of this.)

In the same manner, CATS [71], a network-storage service with strong accountability properties, snapshots the internal state, and only probabilistically detects tampering by auditing a subset of objects for correctness between snapshots. Pavlou and Snodgrass [40] show how to integrate tamper-evidence into a relational database, and can prove the existence of tampering, if suspected. Auditing these systems for consistency is expensive, requiring each auditor visit each snapshot to confirm that any changes between snapshots are authorized.

If an untrusted logger knows that a just-added event or returned commitment will not be audited, then any tampering with the added event or the events fixed by that commitment will be undiscovered, and, by definition, the log is not tamper-evident. To prevent this, *a tamper-evident log requires frequent auditing*. To this end, we propose a tree-based history data structure, logarithmic for all auditing and lookup operations. Events may be added to the log, commitments generated, and audits may be performed independently of one another and at any time. No batching is used. Unlike past designs, we explicitly focus on how tampering will be discovered, through auditing, and we optimize the costs of these audits. Our *history tree* allows loggers to efficiently prove that the sequence of individual logs committed to, over time, make consistent claims about the past.

In Section 3.1 we present our security model and propose semantics for tamper-evident logging. In Section 3.2 we demonstrate our semantics on the classic hash chain based log. In Section 3.3 we present the history tree and in Section 3.4 we prove that it is tamper-evident. In Section 3.5 we describe *Merkle aggregation*, a way to annotate events with attributes which can then be used to perform tamper-evident queries over the log and *safe deletion* of events, allowing unneeded events to be removed in-place, with no additional trusted party, while still being able to prove that no events were improperly purged. Section 3.6 describes a prototype implementation for tamper-evident logging of syslog data traces. Section 3.7 discusses approaches for scaling the logger's performance. A summary appears in Section 3.8.

3.1 Security Model

In this chapter, we make the usual cryptographic assumptions that an attacker cannot forge digital signatures or find collisions in cryptographic hash functions. We assume that clients will encrypt their events before storing them if they want privacy. For simplicity, we assume a single monolithic log on a single host computer. Our goal is to detect tampering as it is impractical to prevent the destruction or alteration of digital records that are in the custody of a Byzantine logger. Replication strategies, outside the scope of this paper, can help ensure availability of the digital records [72].

Tamper-evidence requires auditing. If the log is never examined, then tampering cannot be detected. To this end, we divide a logging system into three logical entities—many *clients* which generate events for appending to a log or history, managed on a centralized but totally untrusted *logger*, which is ultimately audited by one or more trusted *auditors*.

Because of the possibility of insider attacks subverting the logger, we do not trust the logger at any time. We assume clients and auditors have very limited storage capacity while loggers are assumed to have unlimited storage. By auditing the published commitments and demanding proofs, auditors can be convinced that the log's integrity has been maintained. At least one auditor is assumed to be incorruptible. In our system, we distinguish between clients and auditors, while a single host could, in fact, perform both roles.

Insider attacks are not theoretical. Horse race gambling uses a pari-mutuel system where the payout for a particular wager depends on the distribution of all wagers across the different outcomes. The *totalizer* is a security critical component and is responsible for storing wagers and summing all of the wagers for each outcome. An insider with full access to the system subverted the stored wagers for a $3 million dollar payoff [73, 74]. In addition, security policies often explicitly assume that an audit log will exist for later forensic tracing [75]. Such audit logs are of limited use when they are not tamper-evident.

We must trust clients to behave correctly while they are following the event insertion protocol, but we trust clients nowhere else. Of course, a malicious client could insert garbage, but we wish to ensure that an event, once correctly inserted, cannot be undetectably hidden or modified, even if the original client is subsequently colluding with the logger in an attempt to tamper with old data.

Our threat model uses the minimum trust needed to implement a tamper-evident log. The logger is never trusted. Clients are only trusted to generate the correct events and fulfill the event insertion protocol, for if events are not generated correctly, no security guarantee can be offered. If no auditor is honest, then there is no honest agent that will raise the alarm when detecting tampering. Therefore, at least one honest auditor is required.

To ensure these semantics, an untrusted logger must regularly prove its correct behavior to auditors and clients. *Incremental proofs*, demanded of the logger, prove that the current commitment and prior commitment make consistent claims about past events. *Membership proofs* ask the logger to return a particular event from the log along with a proof that the event is consistent with the current commitment. Membership proofs may be demanded by clients after adding events or by auditors verifying that older events remain correctly stored by the logger. These two styles of proofs are sufficient to yield tamper-evidence. As any vanilla lookup operation may be followed by a request for proof, the logger must behave faithfully or risk its misbehavior being discovered.

3.1.1 Semantics of a tamper evident history

We now formalize our desired semantics for secure histories. Each time an event X is sent to the logger, it assigns an index i and appends it to the log, generating a version-i commitment C_i that depends on all of the events to-date, $X_0 \ldots X_i$. The commitment C_i is bound to its version number i, signed, and published.

Although the stream of histories that a logger commits to $(C_0 \ldots C_i, C_{i+1}, C_{i+2} \ldots)$ are supposed to be mutually-consistent, each commitment fixes an *independent* history. Because histories are not known, a priori, to be consistent with one other, we will use primes ($'$) to distinguish between different histories and the events contained within them. In other words, the events in log C_i (i.e., those committed by commitment C_i) are $X_0 \ldots X_i$ and the events in log C'_j are $X'_0 \ldots X'_j$, and we will need to prove their correspondence.

Membership auditing

Membership auditing is performed both by clients, verifying that new events are correctly inserted, and by auditors, investigating that old events are still present and unaltered. The logger is given an event index i and a commitment C_j, $i \leq j$ and is required to return the ith element in the log, X_i, and a proof that C_j implies X_i is the ith event in the log.

Incremental auditing

While a verified membership proof shows that an event was logged correctly in *some* log, represented by its commitment C_j, additional work is necessary to verify that the sequence of logs committed by the logger is consistent over time. In *incremental auditing*, the logger is given two commitments C_j and C'_k, where $j \leq k$, and is required to prove that the two commitments make consistent claims about past events. A verified incremental proof demonstrates that $X_a = X'_a$ for all $a \in [0, j]$. Once verified, the auditor knows that C_j and C'_k commit to the same shared history, and the auditor can safely discard C_j.

A dishonest logger may attempt to tamper with its history by rolling back the log, creating a new fork on which it inserts new events, and abandoning the old fork. Such tampering will be caught if the logging system satisfies *historical consistency* (see Section 3.1.3) and by a logger's inability to generate an incremental proof between commitments on different (and inconsistent) forks when challenged.

3.1.2 Client insertion protocol

Once clients receive commitments from the logger after inserting an event, they must immediately redistribute them to auditors. This prevents the clients from subsequently colluding with the logger to roll back or modify their events. To this end, we need a mechanism, such as a gossip protocol, to distribute the signed commitments from clients to multiple auditors. It's unnecessary for every auditor to audit every commitment, so long as some auditor audits every commitment. (We further discuss tradeoffs with other auditing strategies in Section 3.3.1.)

In addition, in order to deal with the logger presenting different views of the log to different auditors and clients, auditors must obtain and reconcile commitments received from multiple clients or auditors, perhaps with the gossip protocol mentioned above. Alternatively the logger may publish its commitment in a public fashion so that all auditors receive the same commitment [54]. All that matters is that auditors have access to a diverse collection of commitments and demand incremental proofs to verify that the logger is presenting a consistent view.

3.1.3 Definition: tamper evident history

We now define a tamper-evident history system as a five-tuple of algorithms:

H.A $(X) \rightarrow C_j$**.** Given an event X, appends it to the history, returning a new commitment.

H.I .G $(C_i, C_j) \rightarrow P$**.** Generates an incremental proof between C_i and C_j, where $i \leq j$.

H.M .G $(i, C_j) \rightarrow (P, X_i)$**.** Generates a membership proof for event i from commitment C_j, where $i \leq j$. Also returns the event, X_i.

P.I .V $(C_i', C_j) \rightarrow \{\top, \bot\}$**.** Checks that P proves that C_j fixes every entry fixed by C_i' (where $i \leq j$). Outputs $\top$ if no divergence has been detected.

P.M .V $(i, C_j, X_i') \rightarrow \{\top, \bot\}$**.** Checks that P proves that event X_i' is the i'th event in the log defined by C_j (where $i \leq j$). Outputs $\top$ if true.

The first three algorithms run on the logger and are used to append to the log H and to generate *proofs* P. Auditors or clients verify the proofs with algorithms {I .V , M .V }. Ideally, the proof P sent to the auditor is more concise than retransmitting the full history H. Only commitments need to be signed by the logger. Proofs do not require digital signatures; either they demonstrate consistency of the commitments and the contents of an event or they don't. With these five operations, we now define "tamper evidence" as a system satisfying:

Historical Consistency If we have a valid incremental proof between two commitments C_j and C_k, where $j \leq k$, (P.I .V $(C_j, C_k) \rightarrow \top$), and we have a valid membership proof P' for the event X'_i, where $i \leq j$, in the log fixed by C_j (i.e., P'.M .V (i, $C_j, X'_i) \rightarrow \top$) and a valid membership proof for X''_i in the log fixed by C_k (i.e., P''. M .V $(i, C_k, X''_i) \rightarrow \top$), then X'_i must equal X''_i. (In other words, if two commitments commit consistent histories, then they must both fix the same events for their shared past.)

3.1.4 Other threat models

Forward integrity Classic tamper-evident logging uses a different threat model, forward integrity [76]. The forward integrity threat model has two entities: clients who are fully trusted but have limited storage, and loggers who are assumed to be honest until suffering a Byzantine failure. In this threat model, the logger must be prevented from undetectably tampering with events logged prior to the Byzantine failure, but is allowed to undetectably tamper with events logged after the Byzantine failure. One flaw with this model is that if a logger is found to have suffered the Byzantine failure, there is no way to know what parts of the log can be trusted. A more fundamental flaw is that forward integrity offers no security guarantee at all under an insider attack because the logger is always Byzantine.

Although we feel our threat model better characterizes the threats faced by tamper-evident logging, our history tree and the semantics for tamper-evident logging are applicable to this alternative threat model with only minor changes. Under the semantics of forward-integrity, membership auditing just-added events is unnecessary because tamper-evidence only applies to events occurring before the Byzantine failure. Auditing a just-added event is unneeded if the Byzantine failure hasn't happened and irrelevant afterwards. Incremental auditing is still necessary. A client must incrementally audit received commit-

$$\begin{tikzcd}
\cdots & \boxed{C_{j-3}} \arrow[l] \arrow[d] & \boxed{C_{j-2}} \arrow[l] \arrow[d] & \boxed{C_{j-1}} \arrow[l] \arrow[d] & \boxed{C_j} \arrow[l] \arrow[d] \\
 & \boxed{X_{j-3}} & \boxed{X_{j-2}} & \boxed{X_{j-1}} & \boxed{X_j}
\end{tikzcd}$$

Figure 3.1 : A hash-chain log.

$$C_0 = H(X_0, \square)$$

$$C_n = H(X_n, C_{n-1})$$

Figure 3.2 : Algebraic construction of a hash-chain history

ments to prevent a logger from tampering with events occurring before a Byzantine failure by rolling back the log and creating a new fork. Membership auditing is required to look up and examine old events in the log.

Itkis [77] has a similar threat model. His design exploited the fact that if a Byzantine logger attempts to roll back its history to before the Byzantine failure, the history must fork into two parallel histories. He proposed a procedure that tested two commitments to detect divergence without online interaction with the logger and proved an $O(n)$ lower bound on the commitment size. We achieve a tighter bound by virtue of the logger cooperating in the generation of these proofs.

Trusted hardware Rather than relying on auditing, an alternative model is to rely on the logger's hardware itself to be tamper-resistant [4, 78]. Naturally, the security of these systems rests on protecting the trusted hardware and the logging system against tampering by an attacker with complete physical access. Although our design could certainly use trusted hardware as an auditor, cryptographic schemes like ours rest on simpler assumptions, namely the logger can and must prove it is operating correctly.

3.2 Hash chain history

We now demonstrate our formulation of tamper-evidence and the auditing process using a hash chain history. Figure 3.1 demonstrates the construction of a hash chain history. Commitments are denoted by C_i, and events in the history are denoted by X_i. $\square$ denotes the null or empty hash value. This figure is equivalent to the algebraic construction given in Figure 3.2.

For the case of a hash chain history, an *incremental proof*, the proof that the history committed to by C''_j agrees with the history committed to by C'_i with $i \leq j$ is simply $P = (X'_{i+1} \ldots X'_j)$. The auditor can verify the proof by combining the events in P with C'_i, to reconstruct C_j which is compared to C''_j. If they match, then the events committed to by C'_i match those committed to by C''_j. The auditor can now discard C'_i. Auditors must request these proofs to validate that prior events remain in the log. As auditors regularly audit a growing log, the requested incremental proofs will, in aggregate, include every event in the log and the total size of those proofs will grow linearly.

A logger may also generate a *membership proof*, demonstrating that a particular event is fixed by a particular commitment. A membership proof for X_i from commitment C'_j, $i \leq j$ consists of the commitment C_{i-1} and the intermediate entries $P = (X_i \ldots X_j)$ from which C_j is computed and compared to C'_j.

A hash chain offers very efficient event insertion. A membership proof that a just-inserted event was inserted correctly will be constant size. However, hash chain historical lookups and incremental proofs are very expensive as they require sending every intermediate event in the log. Our history tree reduces these costs to logarithmic.

3.3 History tree

We now present our new data structure for representing a tamper-evident history. We start with a Merkle tree [11], which has a long history of uses for authenticating static data. In a Merkle tree, data is stored at the leaves and the hash at the root is a tamper-evident summary of the contents. Merkle trees support logarithmic path lengths from the root to the leaves, permitting efficient random access. Although Merkle trees are a well-known tamper-evident data structure and our use is straightforward, the novelty in our design is in using a versioned computation of hashes over the Merkle tree to efficiently prove that different log snapshots, represented by Merkle trees, with *distinct* root hashes, make consistent claims about the past.

A filled history tree of depth d is a binary Merkle hash tree, storing 2^d events on the leaves. Interior nodes, $I_{i,r}$ are identified by their index i and layer r. Each leaf node $I_{i,0}$, at layer 0, stores event X_i. Interior node $I_{i,r}$ has left child $I_{i,r-1}$ and right child $I_{i+2^{r-1},r-1}$. (Figures 3.3 through 3.5 demonstrate this numbering scheme.) When a tree is not full, subtrees containing no events are represented as $\square$. This can be seen starting in Figure 3.3, a version-2 tree having three events. Figure 3.4 shows a version-6 tree, adding four additional events. Although the trees in our figures have a depth of 3 and can store up to 8 leaves, our design clearly extends to trees with greater depth and more leaves.

Each node in the history tree is *labeled* with a cryptographic hash which, like a Merkle tree, fixes the contents of the subtree rooted at that node. For a leaf node, the label is the hash of the event; for an interior node, the label is the hash of the concatenation of the labels of its children.

An interesting property of the history tree is the ability to efficiently reconstruct old versions or *views* of the tree. Consider the history tree given in Figure 3.4. The logger could reconstruct C_2'' analogous to the version-2 tree in Figure 3.3 by pretending that nodes

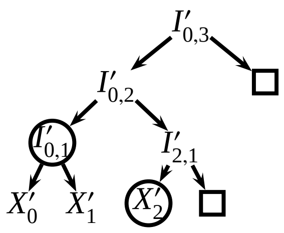

Figure 3.3 : A version-2 history tree with commitment $C'_2 = I'_{0,3}$.

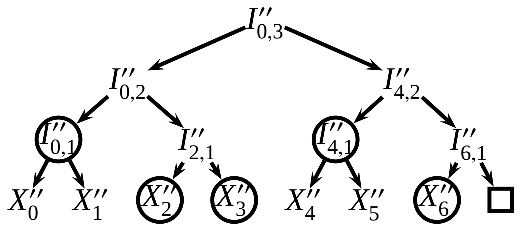

Figure 3.4 : A version-6 history tree with commitment $C''_6 = I''_{0,3}$.

$I''_{4,2}$ and X''_3 were $\square$ and then recomputing the hashes for the interior nodes and the root. If the reconstructed C''_2 matched a previously advertised commitment C'_2, then both trees must have the same contents and commit the same events.

This forms the intuition of how the logger generates an incremental proof P between two commitments, C'_2 and C''_6. Initially, the auditor only possesses commitments C'_2 and C''_6; it does not know the underlying Merkle trees that these commitments fix. The logger must show that both histories commit the same events, i.e., $X''_0 = X'_0, X''_1 = X'_1$, and $X''_2 = X'_2$. To do this, the logger sends a *pruned tree P* to the auditor, shown in Figure 3.5. This pruned tree includes just enough of the full history tree to compute the commitments C_2 and C_6. Unnecessary subtrees are *elided* out and replaced with *stubs*. Events can be either included in the tree or replaced by a stub containing their hash. Because an incremental proof involves *three* history trees, the trees committed by C'_2 and C''_6 with unknown contents

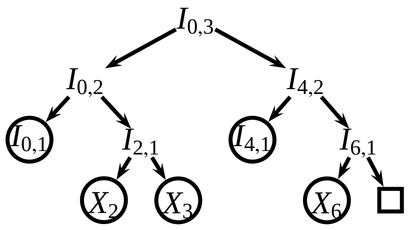

Figure 3.5 : An incremental proof P between a version-2 and version-6 commitment. Hashes for the circled nodes are included in the proof. Other hashes can be derived from their children. Circled nodes in Figures 3.3 and 3.4 must be shown to be equal to the corresponding circled nodes here.

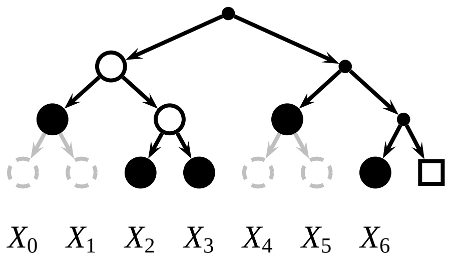

Figure 3.6 : Graphical notation for a history tree analogous to the proof in Figure 3.5. Solid discs represent hashes included in the proof. Other nodes are not included. Dots and open circles represent values that can be recomputed from the values below them; dots may change as new events are added while open circles will not. Grey circle nodes are unnecessary for the proof.

and the pruned tree P, we distinguish them by using a different number of primes ($'$).

From P, shown in Figure 3.5, we reconstruct the corresponding root commitment for a version-6 tree, C_6. We recompute the hashes of interior nodes based on the hashes of their children until we compute the hash for node $I_{0,3}$, which will be the commitment C_6. If $C''_6 = C_6$ then the corresponding nodes, circled in Figures 3.4 and 3.5, in the pruned tree P and the implicit tree committed by C''_6 must match.

Similarly, from P, shown in Figure 3.5, we can reconstruct the version-2 commitment C_2 by pretending that the nodes X_3 and $I_{4,2}$ are $\square$ and, as before, recomputing the hashes for interior nodes up to the root. If $C'_2 = C_2$, then the corresponding nodes, circled in Figures 3.3 and 3.5, in the pruned tree P and the implicit tree committed by C'_2 must match, or $I'_{0,1} = I_{0,1}$ and $X'_2 = X_2$.

If the events committed by C'_2 and C''_6 are the same as the events committed by P, then they must be equal; we can then conclude that the tree committed by C''_6 is consistent with the tree committed by C'_2. By this we mean that the history trees committed by C'_2 and C''_6 both commit the same events, or $X''_0 = X'_0$, $X''_1 = X'_1$, and $X''_2 = X'_2$, even though the events $X''_0 = X'_0$, $X''_1 = X'_1$, X''_4, and X''_5 are unknown to the auditor.

3.3.1 Is it safe to skip nodes during an audit?

In the pruned tree in Figure 3.5, we omit the events fixed by $I_{0,1}$, yet we still preserve the semantics of a tamper-evident log. Even though these earlier events may not be sent to the auditor, they are still fixed by the unchanged hashes above them in the tree. Any attempted tampering will be discovered in future incremental or membership audits of the skipped events. With the history tree, auditors only receive the portions of the history they need to audit the events they have chosen to audit. Skipping events makes it possible to conduct a variety of selective audits and offers more flexibility in designing auditing policies.

Existing tamper-evident log designs based on a classic hash-chain have the form $C_i = H(C_{i-1} \parallel X_i)$, $C_{-1} = \square$ and do not permit events to be skipped. With a hash chain, an incremental or membership proof between two commitments or between an event and a commitment must include *every* intermediate event in the log. In addition, because intermediate events cannot be skipped, each auditor, or client acting as an auditor, must eventually receive every event in the log. Hash chaining schemes, as such, are only feasible with low event volumes or in situations where every auditor is already receiving every event.

When membership proofs are used to investigate old events, the ability to skip nodes can lead to dramatic reductions in proof size. For example, in our prototype described in Section 3.6, in a log of 80 million events, our history tree can return a complete proof for any randomly chosen event in 3100 bytes. In a hash chain, where intermediate events cannot be skipped, an average of 40 million hashes would be sent.

Auditing strategies In many settings, it is possible that not every auditor will be interested in every logged event. Clients may not be interested in auditing events inserted or commitments received by other clients. One could easily imagine scenarios where a single logger is shared across many organizations, each only incentivized to audit the integrity of its own data. These organizations could run their own auditors, focusing their attention on commitments from their own clients, and only occasionally exchanging commitments with other organizations to ensure no forking has occurred. One can also imagine scenarios where independent accounting firms operate auditing systems that run against their corporate customers' log servers.

The log remains tamper-evident if clients gossip their received commitments from the logger to at least one honest auditor who uses it when demanding an incremental proof. By not requiring that every commitment be audited by every auditor, the total auditing overhead across all auditors can be proportional to the total number of events in the log—far cheaper than the number of events times the number of auditors as we might otherwise require.

Skipping nodes offers other time-security tradeoffs. Auditors may conduct audits probabilistically, selecting only a subset of incoming commitments for auditing. If a logger were to regularly tamper with the log, its odds of remaining undetected would become vanishingly small.

3.3.2 Construction of the history tree

Now that we have an example of how to use a tree-based history, we will formally define its construction and semantics. A version-n history tree stores $n + 1$ events, $X_0 \ldots X_n$. Hashes are computed over the history tree in a manner that permits the reconstruction of the hashes of interior nodes of older versions or *views*. We denote the hash on node $I_{i,r}$ by $A^v_{i,r}$ which is parametrized by the node's index, layer and view being computed. A version-v view on a version-n history tree reconstructs the hashes on interior nodes for a version-v history tree that only included events $X_0 \ldots X_v$. When $v = n$, the reconstructed root commitment is C_n. The hashes are computed with the recurrence defined in Figure 3.7.

A history tree can support arbitrary size logs by increasing the depth when the tree fills (i.e., $n = 2^d - 1$) and defining $d = \lceil \log_2(n + 1) \rceil$. The new root, one level up, is created with the old tree as its left child and an empty right child where new events can be added. For simplicity in our illustrations and proofs, we assume a tree with fixed depth d.

Once a given subtree in the history tree is complete and has no more slots to add events, the hash for the root node of that subtree is *frozen* and will not change as future events are added to the log. The logger caches these frozen hashes (i.e., the hashes of frozen nodes) into $\mathrm{FH}_{i,r}$ to avoid the need to recompute them. By exploiting the frozen hash cache, the

$$A^v_{i,0} = \begin{cases} H(0 \| X_i) & \text{if } v \geq i \end{cases} \tag{3.1}$$

$$A^v_{i,r} = \begin{cases} H(1 \| A^v_{i,r-1} \| \square) & \text{if } v < i + 2^{r-1} \\ H(1 \| A^v_{i,r-1} \| A^v_{i+2^{r-1},r-1}) & \text{if } v \geq i + 2^{r-1} \end{cases} \tag{3.2}$$

$$C_n = A^n_{0,d} \tag{3.3}$$

$$A^v_{i,r} \equiv \mathrm{FH}_{i,r} \qquad \text{whenever } v \geq i + 2^r - 1 \tag{3.4}$$

Figure 3.7 : Recurrence for computing hashes.

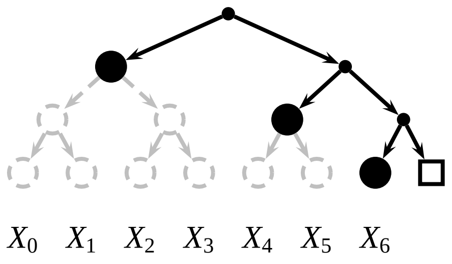

Figure 3.8 : A proof skeleton for a version-6 history tree.

logger can recompute $A^v_{i,r}$ for any node with at most $O(d)$ operations. In a version-n tree, node $I_{i,r}$ is frozen when $n \geq i + 2^r - 1$. When inserting a new event into the log, $O(1)$ expected case and $O(d)$ worse case nodes will become frozen. (In Figure 3.3, node $I'_{0,1}$ is frozen. If event X_3 is added, nodes $I'_{2,1}$ and $I'_{0,2}$ will become frozen.)

Now that we have defined the history tree, we will describe the incremental proofs generated by the logger. Figure 3.6 abstractly illustrates a pruned tree equivalent to the proof given in Figure 3.5, representing an incremental proof from C_2 to C_6. Dots represent unfrozen nodes whose hashes are computed from their children. Open circles represent frozen nodes which are not included in the proof because their hashes can be recomputed from their children. Solid discs represent frozen nodes whose inclusion is necessary by being leaves or stubs. Grayed out nodes represent elided subtrees that are not included in the pruned tree. From this pruned tree and equations (3.1)-(3.4) (shown in Figure 3.7) we can compute $C_6 = A^6_{0,3}$ *and* a commitment from an earlier version-2 view, $A^2_{0,3}$.

This pruned tree is incrementally built from a *proof skeleton*, seen in Figure 3.8—the minimum pruned tree of a version-6 tree consisting only of frozen nodes. The proof skeleton for a version-n tree consists of frozen hashes for the left siblings for the path from X_n to the root. From the included hashes and using equations (3.1)-(3.4), this proof skeleton suffices to compute $C_6 = A^6_{0,3}$.

From Figure 3.8 the logger incrementally builds Figure 3.6 by splitting frozen interior nodes. A node is split by including its children's hashes in the pruned tree instead of itself. By recursively splitting nodes on the path to a leaf, the logger can *include* that leaf in the pruned tree. In this example, we split nodes $I_{0,2}$ and $I_{2,1}$. For each commitment C_i that is to be reconstructable in an incremental proof the pruned tree P must include a path to the event X_i. The same algorithm is used to generate the membership proof for an event X_i.

Given these constraints, we can now define the five history operations in terms of the equations in Figure 3.7.

H.A $(X) \rightarrow C_n$**.** Event is assigned the next free slot, n. C_n is computed by equations (3.1)-(3.4).

H.I .G $(C_i, C_j) \rightarrow P$**.** The pruned tree P is a version-j proof skeleton including a path to X_i.

H.M .G $(i, C_j) \rightarrow (P, X_i)$**.** The pruned tree P is a version-j proof skeleton including a path to X_i and the event X_i.

P.I .V $(C_i'', C_j') \rightarrow \{\top, \bot\}$. From P apply equations (3.1)-(3.4) to compute $A^i_{0,d}$ and $A^j_{0,d}$. This can only be done if P includes a path to the leaf X_i. Return $\top$ if $C_i'' = A^i_{0,d}$ and $C_j' = A^j_{0,d}$.

P.M .V $(i, C_j', X_i') \rightarrow \{\top, \bot\}$. From P apply equations (3.1)-(3.4) to compute $A^j_{0,d}$. Also extract X_i from the pruned tree P, which can only be done if P includes a path to event X_i. Return $\top$ if $C_j' = A^j_{0,d}$ and $X_i = X_i'$.

Although incremental and membership proofs have different semantics, they both follow an identical tree structure and can be built and audited by a common implementation. In addition, a single pruned tree P can embed paths to several leaves to satisfy multiple auditing requests.

What is the size of a pruned tree used as a proof? The pruned tree necessary for satisfying a self-contained incremental proof between C_i and C_j or a membership proof for i in C_j requires that the pruned tree include a path to nodes X_i and X_j. This resulting pruned tree contains at most $2d$ frozen nodes, logarithmic in the size of the log.

In a real implementation, the log may have moved on to a later version, k. If the auditor requested an incremental proof between C_i and C_j, the logger would return the latest commitment C_k, and a pruned tree of at most $3d$ nodes, based around a version-k tree including paths to X_i and X_j. More typically, we expect auditors will request an incremental proof between a commitment C_i and the latest commitment. The logger can reply with the latest commitment C_k and pruned tree of at most $2d$ nodes that included a path to X_i.

The frozen hash cache In our description of the history tree, we described the *full representation* when we stated that the logger stores frozen hashes for all frozen interior nodes in the history tree. This cache is redundant whenever a node's hash can be recomputed from its children. We expect that logger implementations, which build pruned trees for audits and queries, will maintain and use the cache to improve efficiency.

When generating membership proofs, incremental proofs, and query lookup results, there is no need for the resulting pruned tree to include redundant hashes on interior nodes when they can be recomputed from their children. We assume that pruned trees used as proofs will use this *minimum representation*, containing frozen hashes only for stubs, to reduce communication costs.

Can overheads be reduced by exploiting redundancy between proofs? If an auditor is in regular communication with the logger, demanding incremental proofs between the previously seen commitment and the latest commitment, there is redundancy between the pruned subtrees on successive queries.

If an auditor previously requested an incremental proof between C_i and C_j and later requests an incremental proof P between C_j and C_n, the two proofs will share hashes on the path to leaf X_j. The logger may send a *partial proof* that omits these common hashes, and only contains the expected $O(\log_2(n-j))$ frozen hashes that are not shared between the paths to X_j and X_n. This devolves to $O(1)$ if a proof is requested after every insertion. The auditor need only cache d frozen hashes to make this work.

Tree history time-stamping service Our history tree can be adapted to implement a round-based time-stamping service. After every round, the logger publishes the last commitment in public medium such as a newspaper. Let C_i be the commitment from the prior round and C_k be the commitment of the round a client requests that its document X_j be timestamped. A client can request a pruned tree including a path to leaves X_i, X_j, X_k. The pruned tree can be verified against the published commitments to prove that X_j was submitted in the round and its order within that round, without the cooperation of the logger.

If a separate history tree is built for each round, our history tree is equivalent to the threaded authentication tree proposed by Buldas et al. [57] for time-stamping systems.

3.3.3 Storing the log on secondary storage

Our history tree offers a curious property: it can be easily mapped onto write-once append-only storage. Once nodes become frozen, they become immutable, and are thus safe to output. This ordering is predetermined, starting with (X_0), $(X_1, I_{0,1})$, (X_2), $(X_3, I_{2,1}, I_{0,2})$, $(X_4)\ldots$. Parentheses denote the nodes written by each A transaction. If nodes within each group are further ordered by their layer in the tree, this order is simply a post-order traversal of the binary tree. Data written in this linear fashion will minimize disk seek overhead, improving the disk's write performance. Given this layout, and assuming all events are the same size on disk, converting from an (*index*, *layer*) to the byte index used to store that node takes $O(\log n)$ arithmetic operations, permitting efficient direct access.

In order to handle variable-length events, event data can be stored in a separate write-once append-only *value store*, while the leaves of the history tree contain offsets into the value store where the event contents may be found. Decoupling the history tree from the value store also allows many choices for how events are stored, such as databases, compressed files, or standard flat formats.

3.3.4 Comparing to other systems

In this section, we evaluate the time and space tradeoffs between our history tree and earlier hash chain and skip list structures. In all three designs, membership proofs have the same structure and size as incremental proofs, and proofs are generated in time proportional to their size.

Maniatis and Baker [59] present a tamper-evident log using a deterministic variant of a

	Hash chain	Skip list	History tree
A Time	$O(1)$	$O(1)$	$O(\log_2 n)$
I .G proof size to C_k	$O(n-k)$	$O(n)$	$O(\log_2 n)$
M .G proof size for X_k	$O(n-k)$	$O(n)$	$O(\log_2 n)$
Cache size	-	$O(\log_2 n)$	$O(\log_2 n)$
I .G partial proof size	-	$O(n-j)$	$O(\log_2(n-j))$
M .G partial proof size	-	$O(\log_2 (n-i))$	$O(\log_2(n-i))$

Table 3.1 : We characterize the time to add an event to the log and the size of full and partial proofs generated in terms of n, the number of events in the log. For partial proofs audits, j denotes the number of events in the log at the time of the last audit and i denotes the index of the event being membership-audited.

skip list [79]. The skip list history is like a hash-chain incorporating extra skip links that hop over many nodes, allowing for logarithmic lookups.

In Table 3.1 we compare the three designs. All three designs have $O(1)$ storage per event and $O(1)$ commitment size. For skip list histories and tree histories, which support partial proofs (described in Section 3.3.2), we present the cache size and the expected proof sizes in terms of the number of events in the log, n, and the index, j, of the prior contact with the logger or the index i of the event being looked up. Our tree-based history strictly dominates both hash chains and skip lists in proof generation time and proof sizes, particularly when individual clients and auditors only audit a subset of the commitments or when partial proofs are used.

Canonical representation A hash chain history and our history tree have a canonical representation of both the history and of proofs within the history. In particular, from a given commitment C_n, there exists one unique path to each event X_i. When there are multiple paths auditing is more complex because the alternative paths must be checked for consistency with one another, both within a single history, and between the stream of histories $C_i, C_{i+1}, \ldots$ committed by the logger. Extra paths may improve the efficiency of looking up past events, such as in a skip list, or offer more functionality [22], but cannot be trusted by auditors and must be checked.

Maniatis and Baker [59] claim to support logarithmic-sized proofs, however they suffer from this multi-path problem. To verify internal consistency, an auditor with no prior contact with the logger must receive every event in the log in every incremental or membership proof.

Efficiency improves for auditors in regular contact with the logger that use partial proofs and cache $O(\log_2 n)$ state between incremental audits. If an auditor has previously verified the log's internal consistency up to C_j, the auditor will be able to verify the log's internal consistency up to a future commitment C_n with the receipt of events $X_{j+1} \ldots X_n$ Once an auditor knows that the skip list is internally consistent the links that allow for logarithmic lookups can be trusted and subsequent membership proofs on old events will run in $O(\log_2 n)$ time. Skip list histories were designed to function in this mode, with each auditor eventually receiving every event in the log.

Auditing is required Hash chains and skip lists only offer a complexity advantage over the history tree when adding new events, but this advantage is fleeting. If the logger knows that a given commitment will never be audited, it is free to tamper with the events fixed by that commitment, and the log is no longer provably tamper evident. Every commitment returned by the logger must have a non-zero chance of being audited and any evaluation of tamper-evident logging must include the costs of this unavoidable auditing. With multiple auditors, auditing overhead is further multiplied. After inserting an event, hash chains and skip lists suffer an $O(n - j)$ disadvantage the moment they do incremental audits between the returned commitment and prior commitments. They cannot reduce this overhead by, for example, only auditing a random subset of commitments.

Even if the threat model is weakened from our always-untrusted logger to the forward-integrity threat model (See Section 3.1.4), hash chains and skip lists are less efficient than the history tree. Clients can forgo auditing just-added events, but are still required to do incremental audits to prior commitments, which are expensive with hash chains or skip lists.

3.4 Proof for tamper evidence of the history tree

In this section, we prove that the history tree is historically consistent. Recall that the logger creates a stream of commitments, $C_i, C_{i+1}, \ldots$. Each of these commitments commits some history, but these histories are not known to be consistent with each other.

In Theorem 1, we prove that if the logger generates a verified incremental proof between two commitments, then the logs represented by those commitments are consistent and contain identical events. Note that this proof requires referring to several pruned trees that are not known, in advance, to be consistent. We attach Greek letter prefixes to the variables representing a history tree, including $A^v_{i,r}, C_i, X_i$, to distinguish between the various trees until we prove them to be equal. For instance, in the statement of an incremental proof $H.\mathrm{I}\quad.\mathrm{G}\quad(\alpha C_j, \beta C_k) \to P$, we know that the commitment values αC_j and βC_k commit to some set of unknown events, but we do not know whether they are the same until the incremental proof is checked.

Lemma 1 If the reconstructed hashes for a particular view for two frozen subtrees are equal to each other, then corresponding events in those two trees are identical. Algebraically, if $v \geq i$ and $\alpha A^v_{i,r} = \beta A^v_{i,r}$ then $\alpha X_a = \beta X_a$ for all $a \in [i, i + 2^r - 1] \cap [0, v]$, the set of all leaves in that subtree defined for version v.

Proof is by induction over the layer r.

Case $r = 0$ (leaf nodes): By assumption, $v \geq i$. We combine the assumption $\alpha A^v_{i,r} = \beta A^v_{i,r}$ and apply equation 3.1, to $\alpha A^v_{i,r}$ and $\beta A^v_{i,r}$ and get $\alpha X_i = \alpha A^v_{i,r} = \beta A^v_{i,r} = \beta X_i$ which proves $\alpha X_a = \beta X_a$ for $a \in [i, i + 2^0 - 1]$.

Case $r > 0$ (interior nodes): There are two subcases, corresponding to the two cases in equation (3.2).

Subcase $v < i + 2^{r-1}$ (empty right child): We apply equation (3.2) to the definition of $\alpha A^v_{i,r}$ and $\beta A^v_{i,r}$ and derive:

$$H(\alpha A^v_{i,r-1} \parallel \square) = \alpha A^v_{i,r} = \beta A^v_{i,r} = H(\beta A^v_{i,r-1} \parallel \square)$$

By the collision resistance of H, the left children have the same reconstructed hash, $\alpha A^v_{i,r-1} = \beta A^v_{i,r-1}$.

Our inductive hypothesis is true for the left child: $\alpha A^v_{i,r-1} = \beta A^v_{i,r-1}$. We apply it and get $\alpha X_a = \beta X_a$ for all $a \in [i, i + 2^{r-1} - 1] \cap [0, v]$ (the child subtree). However, in this subcase, $v < i + 2^{r-1}$ so therefore $[i, i + 2^{r-1} - 1] \cap [0, v] = [i, i + 2^r - 1] \cap [0, v]$ (i.e., the parent subtree covers the same interval). Thus, $\alpha X_a = \beta X_a$ for all $a \in [i, i+2^{r-1}-1] \cap [0, v]$ implies $\alpha X_a = \beta X_a$ for all $a \in [i, i + 2^r - 1] \cap [0, v]$, as required.

Subcase $v \geq i + 2^{r-1}$ (frozen left child): We apply equation (3.2) to the definition of $\alpha A^v_{i,r}$ and $\beta A^v_{i,r}$ and derive:

$$H(\alpha A^v_{i,r-1} \parallel \alpha A^v_{i+2^{r-1},r-1}) = \alpha A^v_{i,r} = \beta A^v_{i,r} = H(\beta A^v_{i,r-1} \parallel \beta A^v_{i+2^{r-1},r-1})$$

By the collision resistance of H, each child has the same reconstructed hash, or $\alpha A^v_{i,r-1} = \beta A^v_{i,r-1}$ and $\alpha A^v_{i+2^{r-1},r-1} = \beta A^v_{i+2^{r-1},r-1}$.

Our inductive hypothesis is true for each child. We apply it to $\alpha A^v_{i,r-1} = \beta A^v_{i,r-1}$ (the left child) and $\alpha A^v_{i+2^{r-1},r-1} = \beta A^v_{i+2^{r-1},r-1}$ (the right child) and get $\alpha X_a = \beta X_a$ for all $a \in [i, i + 2^{r-1} - 1] \cap [0, v]$ (the left child subtree) and $\alpha X_a = \beta X_a$ for all $a \in [i + 2^{r-1}, (i + 2^{r-1}) +$

Prefix	Use
None	The pruned tree used in the incremental proof.
α	The older commitment in the incremental proof.
β	The newer commitment in the incremental proof.
γ	The pruned tree used to prove membership from the older commitment.
δ	The pruned tree used to prove membership from the newer commitment.
ϵ	The event verified in the membership proof for the older commitment.
ζ	The event verified in the membership proof for the newer commitment.

Table 3.2 : Table of Greek letter prefixes and their uses.

$2^{r-1} - 1] \cap [0, v]$ (the right child subtree). Therefore, combining these intervals, $\alpha X_a = \beta X_a$ for all $a \in [i, i + 2^r - 1] \cap [0, v]$ (the parent subtree). ■

Corrolary 1 If a commitment αC_i and a reconstructed hash of a view $\beta A^i_{0,d}$ have the same value, they match the same events. More formally, $\alpha C_i = \beta A^i_{0,d}$, then $\alpha X_a = \beta X_a$ for all $a \in [0, i]$.

Proof: By the definition of αC_i in equation (3.3), $\alpha C_i = \alpha A^i_{0,d}$. Then apply Lemma 1 to $\alpha A^i_{0,d}$ and $\beta A^i_{0,d}$. ■

Theorem 1 (Historical Consistency) We prove that this the history tree satisfies the requirements of tamper evidence described in Section 3.1.1. If P.I .V $(\alpha C_j, \beta C_k) \rightarrow \top$ with $j \leq k$ then for any $i \leq j$, pruned trees γP and δP, if γP.M .V $(i, \alpha C_j, \epsilon X_i) \rightarrow \top$ and δP.M .V $(i, \beta C_k, \zeta X_i) \rightarrow \top$ then $\epsilon X_i = \zeta X_i$. This means that if we have a verified incremental proof between two commitments $\alpha C_j, \beta C_k$, the two commitments commit to the identical events as long as the membership proofs for those events also verify under their respective commitments.

Tamper evidence involves proving that two histories, represented by their commitments αC_j and βC_k, commit the same events. When we prove tamper evidence, we have three additional pruned trees, P, γP, and δP, which also must also be tested to be consistent with the commitments. Our proof of historical consistency requires 5 distinct history trees: three pruned trees, used as proofs and two history trees, denoted only by their commitments. We also use additional variables to represent events. The trees denoted by our prefixes are described in Table 3.2. Note that the unprefixed $A^j_{0,d}$ and X denote variables in the pruned tree P. The proof proceeds in 4 parts.

Part 1 (incremental proof): From the definition of P.I .V $(\alpha C_j, \beta C_k) \rightarrow \top$, we know that $\alpha C_j = A^j_{0,d}$ and $\beta C_k = A^k_{0,d}$. We are able to reconstruct $A^j_{0,d}$ and $A^k_{0,d}$ because P includes a path to the leaves X_j and X_k. Applying Corollary 1 to $\alpha C_j = A^j_{0,d}$, the events committed to by P are the same as the events committed to by αC_j, thus $X_a = \alpha X_a$ for all $a \in [0, j]$. Similarly, applying Corollary 1 to $\beta C_k = A^k_{0,d}$, the events committed to by P are the same as the events committed to by βC_k, or $X_a = \beta X_a$ for all $a \in [0, k]$.

Part 2 (membership proof): From the definition of γP.M .V $(i, \alpha C_j, \epsilon X_i) \rightarrow \top$, we know that $\alpha C_j = \gamma A^j_{0,d}$ and $\gamma X_i = \epsilon X_i$. We apply Corollary 1 to $\alpha C_j = \gamma A^j_{0,d}$ and derive $\alpha X_a = \gamma X_a$ for all $a \in [0, j]$.

Part 3 (membership proof): From the definition of δP.M .V $(i, \beta C_k, \zeta X_i) \rightarrow \top$, we know that $\beta C_k = \delta A^k_{0,d}$ and $\delta X_i = \zeta X_i$. We apply Corollary 1 to $\beta C_k = \delta A^k_{0,d}$ and derive $\beta X_a = \delta X_a$, for all $a \in [0, k]$.

Part 4 (combining the pieces): For any $i \leq j$, we can combine our four parts together. We have $X_i = \alpha X_i$ and $X_i = \beta X_i$ from part 1, $\alpha X_i = \gamma X_i$ and $\gamma X_i = \epsilon X_i$ from part 2, and $\beta X_i = \delta X_i$ and $\delta X_i = \zeta X_i$ from part 3. We thus conclude that $\epsilon X_i = \zeta X_i$. ■

3.5 Merkle aggregation

Our history tree permits $O(\log_2 n)$ access to arbitrary events, given their index. In this section, we extend our history tree to support efficient, tamper-evident content searches through a feature we call *Merkle aggregation*, which encodes auxiliary information into the history tree. Merkle aggregation permits the logger to perform authorized purges of the log while detecting unauthorized deletions, a feature we call *safe deletion*.

As an example, imagine that a client flags certain events in the log as “important” when it stores them. In the history tree, the logger propagates these flags to interior nodes, setting the flag whenever either child is flagged. To ensure that the tagged history is tamper-evident, this flag can be incorporated into the hash label of a node and checked during auditing. As clients are assumed to be trusted when inserting into the log, we assume clients will properly annotate their events. Membership auditing will detect if the logger incorrectly stored a leaf with the wrong flag or improperly propagated the flag. Incremental audits would detect tampering if any frozen node had its flag altered. Now, when an auditor requests a list of only flagged events, the logger can generate that list along with a proof that the list is complete. If there are relatively few “important” events, the query results can skip over large chunks of the history.

To generate a proof that the list of flagged events is complete, the logger traverses the full history tree H, pruning any subtrees without the flag set, and returns a pruned tree P containing only the visited nodes. The auditor can ensure that no flagged nodes were omitted in P by performing its own recursive traversal on P and verifying that every stub is unflagged.

Figure 3.9 shows the pruned tree for a query against a version-5 history with events X_2 and X_5 flagged. Interior nodes in the path from X_2 and X_5 to the root will also be flagged. For subtrees containing no matching events, such as the parent of X_0 and X_1, we only need to retain the root of the subtree to vouch that its children are unflagged.

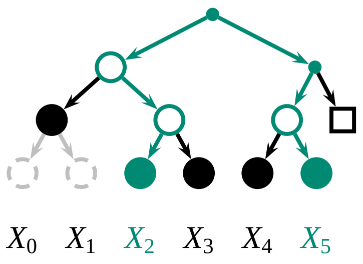

Figure 3.9 : Demonstration of Merkle aggregation with some events flagged as important (highlighted). Frozen nodes that would be included in a query are represented as solid discs.

3.5.1 General attributes

Boolean flags are only one way we may flag log events for later queries. Rather than enumerate every possible variation, we abstract an aggregation strategy over attributes into a 3-tuple, $(\tau, \oplus, \Gamma)$. τ represents the type of attribute or attributes that an event has. $\oplus$ is a deterministic function used to compute the attributes on an interior node in the history tree by *aggregating* the attributes of the node's children. Γ is a deterministic function that maps an event to its attributes. In our example of client-flagged events, the aggregation strategy is $(\tau :=\quad, \oplus := \vee, \Gamma(x) := x.isFlagged)$.

For example, in a banking application, an attribute could be the dollar value of a transaction, aggregated with the function, permitting queries to find all transactions over a particular dollar value and detect if the logger tampers with the results. This corresponds to $(\tau :=\quad, \oplus :=\quad, \Gamma(x) := x.value)$. Or, consider events having internal timestamps, generated by the client, arriving at the logger out of order. If we attribute each node in the tree with the earliest and latest timestamp found among its children, we can now query the logger for all nodes within a given time range, regardless of the order of event arrival.

There are at least three different ways to implement keyword searching across logs using Merkle aggregation. If the number of keywords is fixed in advance, then the attribute τ for events can be a bit-vector or sparse bit-vector combined with $\oplus := \vee$. If the number of keywords is unknown, but likely to be small, τ can be a sorted list of keywords, with $\oplus := \cup$ (set union). If the number of keywords is unknown and potentially unbounded, then a Bloom filter [80] may be used to represent them, with τ being a bit-vector and $\oplus := \vee$. Of course, the Bloom filter would then have the potential of returning false positives to a query, but there would be no false negatives.

Merkle aggregation is extremely flexible because Γ can be *any* deterministic computable function. However, once a log has been created, $(\tau, \oplus, \Gamma)$ are fixed for that log, and the set of queries that can be made is restricted based on the aggregation strategy chosen. In Section 3.6 we describe how we were able to apply these concepts to the metadata used in Syslog logs.

3.5.2 Formal description

To make attributes tamper-evident in history trees, we modify the computation of hashes over the tree to include them. Each node now has a hash label denoted by $A^v_{i,r}.H$ and an annotation denoted by $A^v_{i,r}.A$ for storing attributes. Together these form the node data that is attached to each node in the history tree. Note that the hash label of node, $A^v_{i,r}.H$, does *not* fix its own attributes, $A^v_{i,r}.A$. Instead, we define a *subtree authenticator* $A^v_{i,r}.* = H(A^v_{i,r}.H \parallel A^v_{i,r}.A)$ that fixes the attributes and hash of a node, and recursively fixes every hash and attribute in its subtree. Frozen hashes $\mathrm{FH}_{i,r}.A$ and $\mathrm{FH}_{i,r}.H$ and $\mathrm{FH}_{i,r}.*$ are defined analogously to the non-Merkle-aggregation case.

We could have defined this recursion in several different ways. This representation allows us to elide unwanted subtrees with a small stub, containing one hash and one set of attributes, while exposing the attributes in a way that makes it possible to locally detect if

$$A^v_{i,r}.* = H(A^v_{i,r}.H \,\|\, A^v_{i,r}.A) \tag{3.5}$$

$$A^v_{i,0}.H = \begin{cases} H(0 \,\|\, X_i) & \text{if } v \geq i \end{cases} \tag{3.6}$$

$$A^v_{i,0}.A = \begin{cases} \Gamma(X_i) & \text{if } v \geq i \end{cases} \tag{3.7}$$

$$A^v_{i,r}.H = \begin{cases} H(1 \,\|\, A^v_{i,r-1}.* \,\|\, \square) & \text{if } v < i + 2^{r-1} \\ H(1 \,\|\, A^v_{i,r-1}.* \,\|\, A^v_{i+2^{r-1},r-1}.*) & \text{if } v \geq i + 2^{r-1} \end{cases} \tag{3.8}$$

$$A^v_{i,r}.A = \begin{cases} A^v_{i,r-1}.A & \text{if } v < i + 2^{r-1} \\ A^v_{i,r-1}.A \oplus A^v_{i+2^{r-1},r-1}.A & \text{if } v \geq i + 2^{r-1} \end{cases} \tag{3.9}$$

$$C_n = A^n_{0,d}.* \tag{3.10}$$

Figure 3.10 : Hash computations for Merkle aggregation

the attributes were improperly aggregated.

Our new mechanism for computing hash and aggregates for a node is given in equations (3.5)-(3.10) in Figure 3.10. There is a strong correspondence between this recurrence and the previous one in Figure 3.7. Equations (3.6) and (3.7) extract the hash and attributes of an event, analogous to equation (3.1). Equation (3.9) handles aggregation of attributes between a node and its children. Equation (3.8) computes the hash of a node in terms of the subtree authenticators of its children.

I .G and M .G operate the same as with an ordinary history tree, except that wherever a frozen hash was included in the proof ($\mathrm{FH}_{i,r}$), we now include both the hash of the node, $\mathrm{FH}_{i,r}.H$, and its attributes $\mathrm{FH}_{i,r}.A$. Both are required for recomputing $A^v_{i,r}.A$ and $A^v_{i,r}.H$ for the parent node. A , I .V , and M .V are the same as before except for using the equations (3.5)-(3.10) for computing hashes and propagating attributes. Merkle aggregation inflates the storage and proof sizes by a factor of $(A + B)/A$ where A is the size of a hash and B is the size of the attributes.

3.5.3 Queries over attributes

In Merkle aggregation queries, we permit query results to contain false positives, i.e., events that do not match the query Q. Extra false positive events in the result only impact performance, not correctness, as they may be filtered by the auditor. We forbid false negatives; every event matching Q will be included in the result.

Unfortunately, Merkle aggregation queries can only match attributes, not events. Consequently, we must conservatively transform a query Q over events into a predicate Q^Γ over attributes and require that it be *stable*, with the following properties: If Q matches an event then Q^Γ matches the attributes of that event (i.e., $\forall_x \; Q(x) \Rightarrow Q^\Gamma(\Gamma(x))$). Furthermore, if Q^Γ is true for either child of a node, it must be true for the node itself (i.e., $\forall_{x,y} \; Q^\Gamma(x) \vee Q^\Gamma(y) \Rightarrow Q^\Gamma(x \oplus y)$ and $\forall_x \; Q^\Gamma(x) \vee Q^\Gamma(\Box) \Rightarrow Q^\Gamma(x \oplus \Box)$).

Stable predicates can falsely match nodes or events for two reasons: events' attributes may match Q^Γ without the events matching Q, or nodes may occur where $(Q^\Gamma(x) \vee Q^\Gamma(y))$ is false, but $Q^\Gamma(x \oplus y)$ is true. We call a predicate Q *exact* if there can be no false matches. This occurs when $Q(x) \Leftrightarrow Q^\Gamma(\Gamma(x))$ and $Q^\Gamma(x) \vee Q^\Gamma(y) \Leftrightarrow Q^\Gamma(x \oplus y)$. Exact queries are more efficient because a query result does not include falsely matching events and the corresponding pruned tree proving the correctness of the query result does not require extra nodes.

Given these properties, we can now define the additional operations for performing authenticated queries on the log for events matching a predicate Q^Γ.

H.Q $\quad (C_j, Q^\Gamma) \rightarrow P$ Given a predicate Q^Γ over attributes τ, returns a pruned tree where every elided subtrees does not match Q^Γ.

P.Q .V $(C'_j, Q^\Gamma) \rightarrow \{\top, \bot\}$ Checks the pruned tree P and returns $\top$ if every stub in P does not match Q^Γ and the reconstructed commitment C_j is the same as C'_j.

Building a pruned tree containing all events matching a predicate Q^Γ is similar to building the pruned trees for membership or incremental auditing. The logger starts with a proof skeleton then recursively traverses it, splitting interior nodes when $Q^\Gamma(FH_{i,r}.A)$ is true. Because the predicate Q^Γ is stable, no event in any elided subtree can match the predicate. If there are t events matching the predicate Q^Γ, the pruned tree is of size at most $O((1 + t)\log_2 n)$ (i.e., t leaves with $\log_2 n$ interior tree nodes on the paths to the root).

To verify that P includes all events matching Q^Γ, the auditor does a recursive traversal over P. If the auditor finds an interior stub where $Q^\Gamma(FH_{i,r}.A)$ is true, the verification fails because the auditor found a node that was supposed to have been split. (Unfrozen nodes will always be split as they compose the proof skeleton and only occur on the path from X_j to the root.) The auditor must also verify that pruned tree P commits the same events as the commitment C'_j by reconstructing the root commitment C_j using the equations (3.5)-(3.10) and checking that $C_j = C'_j$.

As with an ordinary history tree, a Merkle aggregating tree requires auditing for tamper-detection. If an event is never audited, then there is no guarantee that its attributes have been properly included. Also, a dishonest logger or client could deliberately insert false log entries whose attributes are aggregated up the tree to the root, causing garbage results to be included in queries. Even so, if Q is stable, a malicious logger cannot hide matching events from query results without detection.

3.5.4 Applications

Safe deletion Merkle aggregation can be used for expiring old and obsolete events that do not satisfy some predicate and prove that no other events were deleted inappropriately. While Merkle aggregation queries prove that no matching event is excluded from a query result, safe deletion requires the contrapositive: proving to an auditor that each purged event was legitimately purged because it did not match the predicate.

Let $Q(x)$ be a stable query that is true for all events that the logger must keep. Let $Q^{\Gamma}(x)$ be the corresponding predicate over attributes. The logger stores a pruned tree that includes all nodes and leaf events where $Q^{\Gamma}(x)$ is true. The remaining nodes may be elided and replaced with stubs. When a logger cannot generate a path to a previously deleted event X_i, it instead supplies a pruned tree that includes a path to an ancestor node A of X_i where $Q^{\Gamma}(A)$ is false. Because Q is stable, if $Q^{\Gamma}(A)$ is false, then $Q^{\Gamma}(\Gamma(X_i))$ and $Q(X_i)$ must also be false.

Safe deletion and auditing policies must take into account that if a subtree containing events $X_i \ldots X_j$ is purged, the logger is unable to generate incremental or membership proofs involving commitments $C_i \ldots C_j$. The auditing policy must require that any audits using those commitments be performed before the corresponding events are deleted, which may be as simple as requiring that clients periodically request an incremental proof to a later or long-lived commitment.

Safe deletion will not save space when using the append-only storage described in Section 3.3.3. However, if data-destruction policies require destroying a subset of events in the log, safe deletion may be used to prove that no unauthorized log events were destroyed.

"Private" search Merkle aggregation enables a weak variant of private information retrieval [27], permitting clients to have privacy for the specific contents of their events. To aggregate the attributes of an event, the logger only needs the attributes of an event, $\Gamma(X_i)$, not the event itself. To verify that aggregation is done correctly also only requires the attributes of an event. If clients encrypt their events and digitally sign their public attributes, auditors may verify that aggregation is done correctly while clients preserve their event privacy from the logger and other clients and auditors.

Bloom filters, in addition to providing a compact and approximate way to represent the presence or absence of a large number of keywords, can also enable private indexing (see, e.g., Goh [81]). The logger has no idea what the individual keywords are within the Bloom filter; many keywords could map to the same bit. This allows for private keywords that are still protected by the integrity mechanisms of the tree.

Annotations can be useful even when not used in predicate queries. For example, a log might track resource use. If resource use is measured by an integer and aggregated by addition, then τ is an integer and $a \oplus b := a + b$. The annotations on interior nodes aggregate the total resources used in that subtree, and the annotation on the root summarizes the total resources used.

3.6 Syslog prototype implementation

Syslog is the standard Unix-based logging system [82], storing events with many attributes. To demonstrate the effectiveness of our history tree, we built an implementation capable of storing and searching syslog events. Using events from syslog traces, captured from our departmental servers, we evaluated the storage and performance costs of tamper-evident logging and secure deletion.

Each syslog event includes a timestamp, the host generating the event, one of 24 *facilities* or subsystem that generated the event, one of 8 logging *levels*, and the *message*. Most events also include a *tag* indicating the program generating the event. Solutions for authentication, management, and reliable delivery of syslog events over the network have already been proposed [83] and are in the process of being standardized [84], but none of this work addresses the logging semantics that we wish to provide.

Our prototype implementation was written in a hybrid of Python 2.5.2 and C++ and was benchmarked on an Intel Core 2 Duo 2.4GHz CPU with 4GB of RAM in 64-bit mode under Linux. Our present implementation is single-threaded, so the second CPU core is underutilized. Our implementation uses SHA-1 hashes and 1024-bit DSA signatures, borrowed from the OpenSSL library.

In our implementation, we use the array-based post-order traversal representation discussed in Section 3.3.3. The value store and history tree are stored in separate write-once append-only files and mapped into memory. Nodes in the history tree use a fixed number of bytes, permitting direct access. Generating membership and incremental proofs requires RAM proportional to the size of the proof, which is logarithmic in the number of events in the log. Merkle aggregation query result sizes are presently limited to those which can fit in RAM, approximately 4 million events.

The storage overheads of our tamper-evident history tree are modest. Our prototype stores five attributes for each event. Tags and host names are encoded as 2-of-32 bit Bloom filters. Facilities and hosts are encoded as bit-vectors. To permit range queries to find every event in a particular range of time, an interval is used to encode the message timestamp. All together, there are twenty bytes of attributes and twenty bytes for a SHA-1 hash for each node in the history tree. Leaves have an additional twelve bytes to store the offset and length of the event contents in the value store.

We ran a number of simulations of our prototype to determine the processing time and space overheads of the history tree. To this end, we collected a trace of four million events from thirteen of our departmental server hosts over 106 hours. We observed 9 facilities, 6 levels, and 52 distinct tags. 88.1% of the events are from the mail server and 11.5% are from 98,743 failed ssh connection attempts. Only .393% of the log lines are from other sources. In testing our history tree, we replay this trace 20 times to insert 80 million events. Our syslog trace, after the replay, occupies 14.0 GB, while the history tree adds an additional 13.6 GB.

3.6.1 Performance of the logger

The logger is the only centralized host in our design and may be a bottleneck. The performance of a real world logger will depend on the auditing policy and relative frequency between inserting events and requesting audits. Rather than summarize the performance of the logger for one particular auditing policy, we benchmark the costs of the various tasks performed by the logger.

Our captured syslog traces averaged only ten events per second. Our prototype can insert events at a rate of 1,750 events per second, including DSA signature generation. Inserting an event requires four steps, shown in Table 3.3, with the final step, signing the resulting commitment, responsible for most of the processing time. Throughput would increase to 10,500 events per second if the DSA signatures were computed elsewhere (e.g., leveraging multiple CPU cores). (Section 3.7 discusses scalability in more detail.) This corresponds to 1.9MB/sec of uncompressed syslog data (1.1 TB per week).

We also measured the rate at which our prototype can generate membership and incremental proofs. The size of an incremental proof between two commitments depends upon the distance between the two commitments. As the distance varies from around two to two million events, the size of a self-contained proof varies from 1200 bytes to 2500 bytes. The speed for generating these proofs varies from 10,500 proofs/sec to 18,000 proofs/sec, with shorter distances having smaller proof sizes and faster performance than longer distances.

Step	Task	% of CPU	Rate (events/sec)
A	Parse syslog message	2.4%	81,000
B	Insert event into log	2.6%	66,000
C	Generate commitment	11.8%	15,000
D	Sign commitment	83.3%	2,100
	Membership proofs (with locality)	-	8,600
	Membership proofs (no locality)	-	32

Table 3.3 : Performance of the logger in each of the four steps required to insert an event and sign the resulting commitment and in generating membership proofs. Rates are given assuming nothing other than the specified step is being performed.

For both incremental and membership proofs, compressing by gzip [85] halves the size of the proofs, but also halves the rate at which proofs can be generated.

After inserting 80 million events into the history tree, the history tree and value store require 27 GB, several times larger than our test machine's RAM capacity. Table 3.3 presents our results for two membership auditing scenarios. In our first scenario we requested membership proofs for random events chosen among the most recent 5 million events inserted. Our prototype generated 8,600 self-contained membership proofs per second, averaging 2,400 bytes each. In this high-locality scenario, the most recent 5 million events were already sitting in RAM. Our second scenario examined the situation when audit requests had low locality by requesting membership proofs for random events anywhere in the log. The logger's performance was limited to our disk's seek latency. Proof size averaged 3,100 bytes and performance degraded to 32 membership proofs per second. (We discuss how this might be overcome in Section 3.7.2.)

To test the scalability of the history tree, we benchmarked insert performance and auditing performance on our original 4 million event syslog event trace, without replication, and the 80 million event trace after 20x replication. Event insertion and incremental auditing are roughly 10% slower on the larger log.

3.6.2 Performance of auditors and clients

The history tree places few demands upon auditors or clients. Auditors and clients must verify the logger's commitment signatures and must verify the correctness of pruned tree replies to auditing requests. Our machine can verify 1,900 DSA-1024 signatures per second. Our current tree parser is written in Python and is rather slow. It can only parse 480 pruned trees per second. Once the pruned tree has been parsed, our machine can verify 9,000 incremental or membership proofs per second. Presently, one auditor cannot verify proofs as fast as the logger can generate them, but auditors can clearly operate independently of one another, in parallel, allowing for exceptional scaling, if desired.

3.6.3 Merkle aggregation results

In this subsection, we describe the benefits of Merkle aggregation in generating query results and in safe deletion. In our experiments, due to limitations of our implementation in generating large pruned trees, our Merkle aggregation experiments used the smaller four million event log.

We used 86 different predicates to investigate the benefits of safe deletion and the overheads of Merkle aggregation queries. We used 52 predicates, each matching one tag, 13 predicates, each matching one host, 9 predicates, each matching one facility, 6 predicates, one matching each level, and 6 predicates, each matching the k highest logging levels.

The predicates matching tags and hosts use Bloom filters, are *inexact*, and may have false positives. This causes 34 of the 65 Bloom filter query results to include more nodes than our "worst case" expectation for exact predicates. By using larger Bloom filters, we reduce the chances of spurious matches. When a 4-of-64 Bloom filter is used for tags and

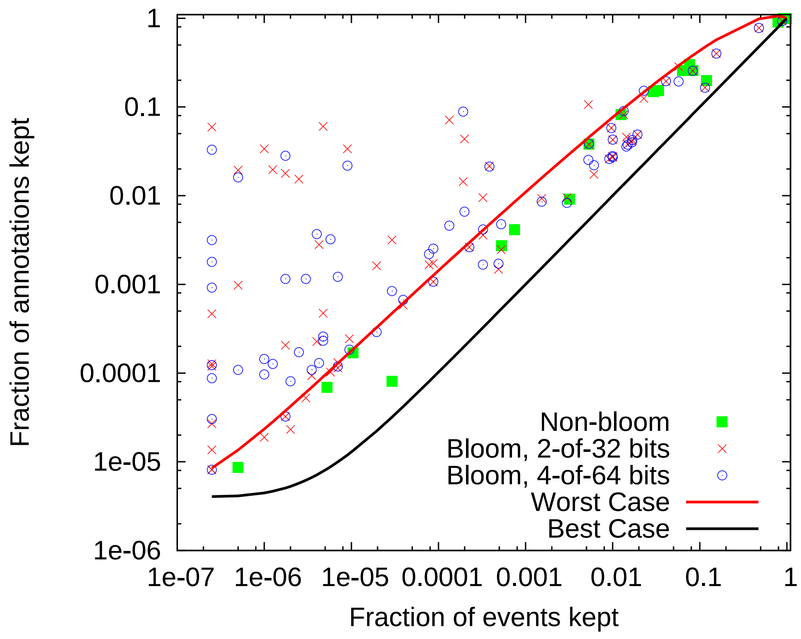

Figure 3.11 : Safe deletion overhead. For a variety of queries, we plot the fraction of hashes and attributes kept after deletion versus the fraction of events kept.

hostnames, pruned trees resulting from search queries average 15% fewer nodes, at the cost of an extra 64 bits of attributes for each node in the history tree. In a real implementation, the exact parameters of the Bloom filter would best be tuned to match a sample of the events being logged.

Merkle aggregation and safe deletion Safe deletion allows the purging of unwanted events from the log. Auditors define a stable predicate over the attributes of events indicating which events must be kept, and the logger keeps a pruned tree of only those matching events. In our first test, we simulated the deletion of all events except those from a particular host. The pruned tree was generated in 14 seconds, containing 1.92% of the events in the full log and serialized to 2.29% of the size of the full tree. Although 98.08% of the events were purged, the logger was only able to purge 95.1% of the nodes in the history tree because the logger must keep the hash label and attributes for the root nodes of elided subtrees.

When measuring the size of a pruned history tree generated by safe deletion, we assume the logger caches hashes and attributes for all interior nodes in order to be able to quickly generate proofs. For each predicate, we measure the *kept ratio*, the number of interior node or stubs in a pruned tree of all nodes matching the predicate divided by the number of interior nodes in the full history tree. In Figure 3.11 for each predicate we plot the kept ratio versus the fraction of events matching the predicate. We also plot the analytic best-case and worst-case bounds, based on a continuous approximation. The minimum overhead occurs when the matching events are contiguous in the log. The worst-case occurs when events are maximally separated in the log. Our Bloom-filter queries do worse than the "worst-case" bound because Bloom filter matches are inexact and will thus trigger false positive matches on interior nodes, forcing them to be kept in the resulting pruned tree. Although many Bloom filters did far worse than the "worst-case," among the Bloom filters that matched fewer than 1% of the events in the log, the logger is still able to purge over 90% of the nodes in the history tree and often did much better than that.

Merkle aggregation and authenticated query results In our second test, we examine the overheads for Merkle aggregation query lookup results. When the logger generates the results to a query, the resulting pruned tree will contain both matching events and history tree overhead, in the form of hashes and attributes for any stubs. For each predicate, we measure the *query overhead ratio*—the number of stubs and interior nodes in a pruned tree divided by the number of events in the pruned tree. In Figure 3.12 we plot the query overhead ratio versus the fraction of events matching the query for each of our 86 predicates. This plot shows, for each event matching a predicate, proportionally how much extra

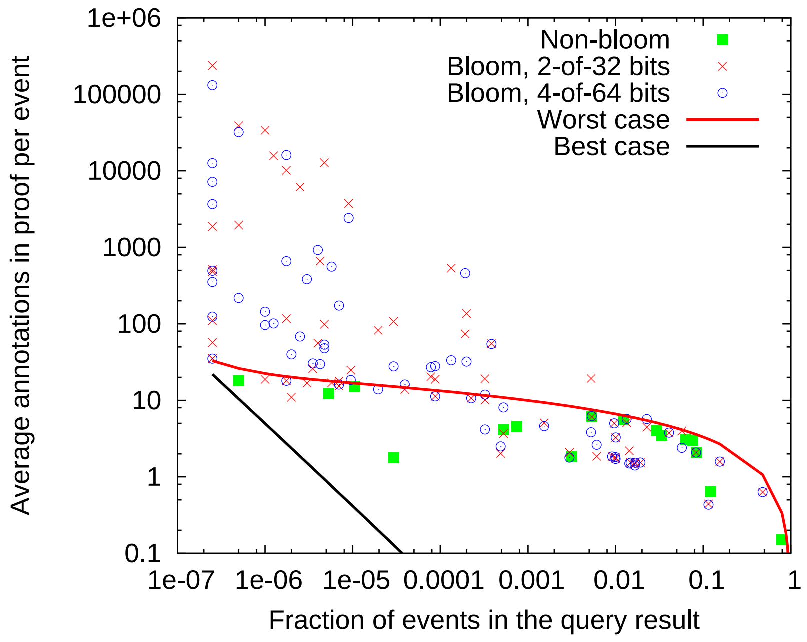

Figure 3.12 : Query overhead per event. We plot the ratio between the number of hashes and matching events in the result of each query versus the fraction of events matching the query.

overhead is incurred, per event, for authentication information. We also plot the analytic best-case and worst-case bounds, based on a continuous approximation. The minimum overhead occurs when the matching events are contiguous in the log. The worst-case occurs when events are maximally separated in the log. With exact predicates, the overhead of authenticated query results is very modest, and again, inexact Bloom filter queries will sometimes do worse than the "worst case."

3.7 Scaling a tamper-evident log

In this section, we discuss techniques to improve the insert throughput of the history tree by using concurrency, and to improve the auditing throughput with replication. We also discuss a technique to amortize the overhead of a digital signature over several events.

3.7.1 Faster inserts via concurrency

Our tamper-evident log offers many opportunities to leverage concurrency to increase throughput. Perhaps the simplest approach is to offload signature generation. From Table 3.3, signatures account for over 80% of the runtime cost of an insert. Signatures are not included in any other hashes and there are no interdependencies between signature computations. Furthermore, signing a commitment does not require knowing anything other than the root commitment of the history tree. Consequently, it's easy to offload signature computations onto additional CPU cores, additional hosts, or hardware crypto accelerators to improve throughput.

It is possible for a logger to also generate commitments concurrently. If we examine Table 3.3, parsing and inserting events in the log is about two times faster than generating commitments. Like signatures, commitments have no interdependencies on one other; they depend only on the history tree contents. As soon as event X_j is inserted into the tree and $O(1)$ frozen hashes are computed and stored, a new event may be immediately logged. Computing the commitment C_j only requires read-only access to the history tree, allowing it to be computed concurrently by another CPU core without interfering with subsequent events. By using shared memory and taking advantage of the append-only write-once semantics of the history tree, we would expect concurrency overhead to be low.

We have experimentally verified the maximum rate at which our prototype implementation, described in Section 3.6, can insert syslog events into the log at 38,000 events per second using only one CPU core on commodity hardware. This is the maximum throughput our hardware could potentially support. In this mode we assume that digital signatures, commitment generation, and audit requests are delegated to additional CPU cores or hosts. With multiple hosts, each host must build a replica of the history tree which can be done at least as fast as our maximum insert rate of 38,000 events per second. Additional CPU cores on these hosts can then be used for generating commitments or handling audit requests.

For some applications, 38,000 events per second may still not be fast enough. Scaling beyond this would require fragmenting the event insertion and storage tasks across multiple logs. To break interdependencies between them, the fundamental history tree data structure we presently use would need to evolve, perhaps into disjoint logs that occasionally entangle with one another as in timeline entanglement [59]. Designing and evaluating such a structure is future work.

3.7.2 Logs larger than RAM

For exceptionally large audits or queries, where the working set size does not fit into RAM, we observed that throughput was limited to disk seek latency. Similar issues occur in any database query system that uses secondary storage, and the same software and hardware techniques used by databases to speed up queries may be used, including faster or higher throughput storage systems or partitioning the data and storing it in-memory across a cluster of machines. A single large query can then be issued to the cluster node managing each sub-tree. The results would then be merged before transmitting the results to the auditor. Because each sub-tree would fit in its host's RAM, sub-queries would run quickly.

3.7.3 Signing batches of events

When large computer clusters are unavailable and the performance cost of DSA signatures is the limiting factor in the logger's throughput, we may improve performance of the logger by allowing multiple updates to be handled with one signature computation.

Normally, when a client requests an event X to be inserted, the logger assigns it an index i, generates the commitment C_i, signs it, and returns the result. If the logger has insufficient CPU to sign every commitment, the logger could instead delay returning C_i until it has a signature for some later commitment C_j ($j \geq i$). This later signed commitment could then be sent to the client expecting an earlier one. To ensure that the event X_i in the log committed by C_j was X, the client may request a membership proof from commitment C_j to event i and verify that $X_i = X$. This is safe due to the tamper-evidence of our structure. If the logger were ever to later sign a C_i inconsistent with C_j, it would fail an incremental proof.

In our prototype, inserting events into the log is twenty times faster than generating and signing commitments. The logger may amortize the costs of generating a signed commitment over many inserted events. The number of events per signed commitment could vary dynamically with the load on the logger. Under light load, the logger could sign every commitment and insert 1,750 events per second. With increasing load, the logger might sign one in every 16 commitments to obtain an estimated insert rate of 17,000 events per second. Clients will still receive signed commitments within a fraction of a second, but several clients can now receive the same commitment. Note that this analysis only considers the maximum insert rate for the log and does not include the costs of replying to audits. The overall performance improvements depend on how often clients request incremental and membership proofs.

3.8 Summary

In this chapter, we have shown that regular and continous auditing is a critical operation for any tamper-evident log system, for without auditing, clients cannot detect if a Byzantine logger is misbehaving by not logging events, removing unaudited events, or forking the log. From this requirement we have developed a new tamper-evident log design, based on a new Merkle tree data structure that permits a logger to produce concise proofs of its correct behavior. Our system eliminates any need to trust the logger, instead allowing clients and auditors of the logger to efficiently verify its correct behavior with only a constant amount of local state. By sharing commitments among clients and auditors, our design is resistant even to sophisticated forking or rollback attacks, even in cases where a client might change its mind and try to repudiate events that it had logged earlier.

We also proposed Merkle aggregation, a flexible mechanism for encoding auxiliary attributes into a Merkle tree that allows these attributes to be aggregated from the leaves up to the root of the tree in a verifiable fashion. This technique permits a wide range of efficient, tamper-evident queries, as well as enabling verifiable, safe deletion of “expired” events from the log.

Our prototype implementation supports thousands of events per second, and can easily scale to very large logs. We also demonstrated the effectiveness of Bloom filters to enable a broad range of queries. By virtue of its concise proofs and scalable design, our techniques can be applied in a variety of domains where high volumes of logged events might otherwise preclude the use of tamper-evident logs.

Chapter 4

PAD designs

Authenticated dictionaries were first proposed for representing certificate revocation lists in a public key infrastructure, allowing the certificate revocation list to be served from untrusted machines and signed by the trusted author [5]. Clients send lookup requests for a particular key to the server, which then replies with a proof containing either the corresponding value, or a “no such value” reply, authenticated by the author’s signature. Persistent authenticated dictionaries extend this to also support lookups on earlier versions of the dictionary and were introduced by Anagnostopoulos et al. [7], using applicative (i.e., functional or mutation-free) red-black trees and skiplists, requiring $O(\log n)$ storage per update. In this chapter we will describe our persistent authenticated dictionary designs, their potential features, and various threat models they can run under. We present improved tree-based PADs and present a new foundation for PAD designs, tuple-based PADs, which offer constant-sized lookup proofs.

In Section 4.1 we discuss threat models and features that PADs may support. In Section 4.2, we show how to adapt Sarnak and Tarjan’s construction [10] in order to build PADs with lower storage overheads, including a design with constant storage per update. In Section 4.3 we develop *super-efficient* PADs based around a different design principle, offering constant-sized authentication results, as well as constant storage per update. In Section 4.4 we discuss approaches for scaling our PAD designs. In Section 4.5 we describe future work, extensions, and applications of our PAD designs. In Chapter 5 we will evaluate all of the PAD designs presented in this chapter.

4.1 Definitions and models

In this chapter, we focus on authenticating set-membership and non-membership queries over a dynamic set, stored on an untrusted server. To prevent the server from lying about the data being stored, the author supplies authentication information to the server permitting lookup responses to be verified.

The authenticated dictionary [5] abstraction supports the ordinary dictionary operations, I (K , V) and D (K), which *update* the contents. Lookups, L (K) → (V , *P*) return both the answer or □ if no such key exists, and a *lookup proof P* of the correctness of their result. Ultimately, a server must prove that a given query result is consistent with some external data, such as an author's signature on the tree's root hash.

Authenticated dictionaries become persistent [7] when they allow the author to take snapshots of the contents of the dictionary. Queries can be on the current version, or any historical snapshot. Each snapshot results in a new *version* of the PAD. The author then sends an *update blob* to the server containing data and authentication information that the server stores in a *repository*, used to respond to lookup requests from clients. Clients send lookup requests containing a lookup key and a version number to the server and receive a *lookup proof* of the membership of the key and its corresponding value, or a proof of non-membership of that key, authenticated by the author's signature. PADs ideally have efficient storage of all the snapshots, presumably sharing state from one snapshot to the next. Snapshots can be taken at any time. For simplicity when we evaluate costs, we will assume a snapshot is taken after every update. The security guarantee offered by a PAD is that a server cannot convince a client of the membership or non-membership of a key and value in a particular snapshot unless the author placed it there.

4.1.1 Threat model

We make typical assumptions for the security of cryptographic primitives. We assume that we have idealized cryptographic one-way hash functions (i.e., collisions never occur and the input can never be derived from the output), and that public key cryptography systems' semantics are similarly idealized. We also assume the existence of a trusted PKI or other means to identify the public key associated with an author. In addition, there are several threat models that a PAD design can function under:

Third-party trust model This threat model implements a publishing paradigm with three parties: an *author* with limited storage and possibly intermittent connectivity, an untrusted *server* with significant storage and a consistent online connection, and multiple *clients* who perform queries and have limited storage. We assume that the author of the data is trusted by clients who wish to detect if the server is tampering with the stored data or returning incorrect responses to lookups. The author asks the server to insert or remove (key, value) pairs, providing any necessary authentication information. When clients contact the server they will verify the resulting proof which will include validating the consistency of the server's data structure as well as the author's digital signature.

An example of this is the original use of an authenticated dictionary to manage the list of valid and revoked certificates (the CRL) without trusting the server distributing the list. A CRL server naturally extends to supporting historical lookups and might be used to answer questions such as "Was certificate *C* valid on January 14th, 1998, when contract *D* was signed by that certificate?" Another example occurs with a remote backup service where the author is the client and the author wants to access historical versions of its backed-up files.

Untrusted author We also consider scenarios where the author of a PAD is not trusted, which can be relevant to a variety of financial auditing and regulatory compliance scenarios. For instance, the author may wish to maliciously change past values of the PAD, possibly in collusion with the server. Or, the author may be responsible for collecting and aggregating records, such as a list of bank accounts and balances and attempt to misbehave. Fortunately, if the author ever signs inconsistent answers or it improperly aggregates records, its misbehavior can be caught by clients and auditors. For details, please see *root authenticators* in Section 4.1.2.

An example application of a PAD with an untrusted author occurs in pari-mutuel gambling, used in horse racing. When betting is open, preliminary odds are continuously computed, by summing all of the wagers up to that point in time for each outcome, and displayed to bettors. Reporting incorrect odds to bettors, either by accident or fraud, can alter betting patterns, and thus payoffs. If the server distributes the continuously changing totals through a PAD, its signature is a commitment and can be used to prove misbehavior.

Buggy server There are also applications, such as libraries or archives, where the author and server are honest, but may inadvertently suffer corruption or make mistakes in data versioning or when tracking data provenance.

For instance, digital archives are responsible for managing and preserving large, constantly growing data sets. Three issues that are faced by such archives are detecting lost data, detecting corruption, and tracking metadata, such as who created an archived object and when. In addition, many datasets are subject to constant revision as new data arrives [72]. Digital signatures and cryptographic hashes are ideal for binding metadata to data and detecting corruption. However, an authenticated dictionary or PAD is ideal for discovering when data has gone missing.

4.1.2 Features

An authenticated dictionary (persistent or not) may support many features. In this section, we describe features supported by the dictionaries we investigate.

Super-efficiency. The proof returned on a lookup request is constant-sized. Our tuple-based PADs, described in Section 4.3 offer super-efficiency.

Partial persistence. The PADs we consider are actually partially persistent, meaning that although any version of the authenticated dictionary may be queried, only the latest version can be modified.[*] Whenever we use the term "persistent" in this thesis, we really mean "partially persistent."

History independence. Some data structures can hide information as to the order in which they were constructed. For instance, if data items are stored, sorted in an array, no information would remain as to the insertion order. History independence can derive from randomization; Micciancio [88] shows a 2-3 tree whose structure depends on coin tosses, not the keys' insertion order.

History independence can also derive from data structures that have a canonical or unique representation [89]. To this end, a data structures can be "set-unique" [90], meaning that a given set of keys in the dictionary has a unique and canonical representation (see Section 4.2.2). Some of our tree-based and tuple-based PAD designs are history-independent.

[*]In the persistency literature [86], the term "persistent" is reserved for data structures where any version, present or past, may be updated, thus forming a tree of versions. Path copying trees, described in Section 4.2.5, are an example of such a data structure. Confluently persistent data structures permit merge operations between snapshots [87].

In a persistent dictionary, history independence means that if multiple updates occur between two adjacent snapshots, the client learns nothing as to the order in which the updates occurred and the server learns nothing if it receives the updates as a batch. In addition, it must not be possible for a client to learn anything about the keys in one snapshot, given query responses from any other snapshots.

Aggregates. Any tree data structure may include aggregates that summarize the children of a given node (e.g., their minimum and maximum values, or their sum). These aggregates are valuable on their own and may be used for searching or other applications (see Merkle aggregation, described in Section 4.2.1). Our tree-based PADs support aggregates.

Root authenticators. For each snapshot, it would be beneficial if there was a single value that fixes or commits the entire dictionary at that particular time. This value can then be stored and replicated efficiently by clients, stored in a time-stamping system [54, 57], or tamper-evident log [21–23]. Root authenticators simplify the process of discovering when an untrusted author or server may be lying about the past. Mistrusting clients need only to discover that the author has signed different root authenticators for the same snapshot.

4.2 Tree-based PADs

In this section, we describe how we can build PADs with balanced search trees. Tree-based PADs have lookup proof sizes, update sizes and lookup proof verification times that are logarithmic in the number of keys in the dictionary. Tree-based PADs offer a range of query time and storage-space tradeoffs. In this section, we first describe the three components from which we build our tree-based PADs: Merkle trees, treaps, and persistent binary search trees. We then show how to combine them.

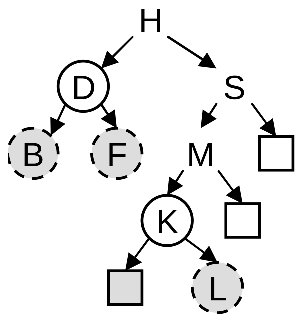

Figure 4.1 : Graphical notation for a lookup proof for M or a proof of non-membership for N. Circles denote the roots of elided subtrees whose children, grayed out, need not be included.

4.2.1 Authenticated dictionaries based on Merkle trees

Given a search tree, where each node contains a key, value, and two child pointers, we can build an authenticated dictionary by building a Merkle tree [11]. For each node x, we assign a *subtree authenticator* $x.H$ with the following recurrence: $x.H = H(x.key, H(x.val), x.left.H, x.right.H)$. H denotes a cryptographic hash function. The *root authenticator*, $root.H$, authenticates the whole tree. It may then be published or signed by the author.

A *lookup proof*, seen in Fig. 4.1 and returned on a L request, is a proof that a key k_q is or is not in the tree. It consists of a pruned tree containing the search path to k_q. Subtree authenticators for the sibling nodes on the search path are included in the proof as well as subtree authenticators of the children of the node containing k_q, if k_q is found. From this pruned tree, the root authenticator is reconstructed and compared to the given root authenticator. We can prove that a key is not in the tree by showing that the unique in-order location where that key would otherwise be stored is empty.

For a balanced search tree, a lookup proof has size $O(\log n)$, and can be generated in $O(\log n)$ time if the subtree authenticators are precomputed. Conventional implementations of authenticated search trees implement a logical *subtree authenticator cache* storing the subtree authenticator for each node in the node itself. Note that this cache is optional, because the server could certainly recompute any hash on the fly from the existing tree. Without a cache, generating a lookup proof requires $O(n)$ time for recomputing subtree authenticators of elided subtrees. Of course, the cache has obvious performance benefits. In Section 4.2.5, we will consider how, where, and when these subtree authenticators are cached and investigate tradeoffs in caching strategies.

Smaller proofs result if the search tree is very close to being balanced. During updates, the search tree can be rebalanced by applying the update-rules of any balanced tree algorithm such as a treap or red-black tree. We do not need to include red-black bits or treap priority values in the hash. These are only hints needed by the author to generate balanced trees, and thus only affect efficiency, not correctness or security.

Merkle aggregation. Merkle aggregation was described in Section 3.5 as applied to annotating events in a Merkle tree storing a tamper-evident log. These annotations are then aggregated up to the root of the tree where they may be directly queried or used to perform authenticated searches. To prevent tampering, the annotations of a node are included in the subtree authenticator of its parent. If the author is not trusted, these annotations can be checked by auditors to verify the author's proper behavior.

Merkle aggregation has a straightforward extension to binary search trees that include keys and values in interior nodes. We let the *subtree aggregate* of a node x be $x.A$, Γ be a function that computes the annotation associated with a key and value pair, and $\oplus$ be a function that aggregates. If we define $x.* = H(x.H, x.A)$, then we can describe the Merkle aggregation over a search tree with the formulas: $x.A = \Gamma(x.key, x.val) \oplus x.left.A \oplus x.right.A$ and $x.H = H(x.key, H(x.val), x.left.*, x.right.*)$. Wherever a host previously stored or included the hash of a node in a proof, it will now include the node's hash and aggregate, which can be cached or recomputed as-needed.

Merkle aggregation applied to a search tree or persistent search has several potential uses. Just as with a tamper-evident log, annotations can be used as an auxiliary indexing data structure, allowing items in the dictionary to be found without needing to know their key. Merkle aggregation can also be used as a generic aggregation strategy, allowing aggregation to occur in a tamper-evident fashion on an untrusted server.

4.2.2 Treap

Treaps [91] are a randomized search tree that can implement a dictionary with a $O(\log n)$ expected cost of an insert, delete, or lookup. Treaps support efficient set union, difference, and intersection operations [92]. We could use any other balanced search tree that supports $O(1)$ expected (not amortized) node mutations per update, such as AVL or red-black trees [93]. We like treaps for their set-uniqueness properties (discussed further below).

Each node in a treap is given a key, value, priority, and left and right child pointers. Nodes in a treap obey the standard search-key order; a node's key always compares greater than all of the keys in its left subtree and less than all of the keys in its right subtree. In addition, each node in a treap obeys the heap property on its priorities; a node's priority is always less than the priorities of its descendants. Operations that mutate the tree will perform rotations to preserve the heap property on the priorities. When the priorities are assigned at random, the resulting tree will be probabilistically balanced. Furthermore, given an assignment of priorities to nodes, a treap on a given set is unique.[†] We exploit

[†]Proof sketch: If all priorities are unique for a given set of keys, then there exists one unique minimum-priority node, which becomes the root. This uniquely divides the set of keys in the treap into two sets, those less than and greater than that node's key, stored in the left and right subtrees, respectively. By induction, we can assume that the subtrees are also unique.

this uniqueness by creating *deterministic treaps*, assigning priorities using a cryptographic digest of the key, creating a set-unique representation.

Assuming that the cryptographic digest is a random oracle, in expectation, each insert and delete only mutates $O(1)$ nodes, consisting of one node having a child pointer modified and $O(1)$ rotations. The expected path length to a key in the treap is $O(\log n)$. The worst case is n.

Benefits of a set-unique representation. Deterministic treaps are set-unique, which means that all authenticated dictionaries with the same contents have identical tree structures. If we build Merkle trees from these treaps, then any two authenticated dictionaries with identical contents will have identical root hashes. Set-uniqueness makes our treaps history independent. The root hash that authenticates a treap leaks no information about the insertion order of the keys or the past contents of the treap, which may be valuable, for example, with electronic vote storage or with zero-knowledge proofs.

History-independence is also useful if an dictionary is used to store or synchronize replicated state in a distributed system. Updates may arrive to replicas out-of-order, perhaps through multicast or gossip protocols. Also, by using a set-unique authenticated data structure, we can efficiently determine if two replicas are inconsistent.

History independence makes it easier to recover from backups or create replicas. If a host tries to recover the dictionary contents from a backup or another replica, history independence assures that the recovered dictionary has the same root hash. Were a non-set-unique data structure, such as red-black tree, used the different insertion order between the original dictionary and that used when recovering would likely lead to different root hashes even though the recovered dictionary had the same contents.

4.2.3 Skiplist

Anagnostopoulos et al. [7] described PADs based on path copying red-black trees and skiplists. In this section, we describe skiplists and how they can represent an authenticated dictionary. We improve on their constructions of a skiplist authenticated dictionary in two ways. First, we represent a skiplist as a strict applicative binary tree to make it amenable to being stored using any persistent search tree design. We also present a lookup proof construction that is approximately half the size of previous approaches.

Our applicative tree-representation of skiplists is based on the skiplist authentication trees by Goodrich et al. [12]. A skiplist [79] is a datastructure offering logarithmic lookups, inserts, and deletes. A classic skiplist is a singly-linked list except that nodes may have several outgoing links, stored in a variable-sized array, which can skip over a large number of list nodes. An alternative formulation of skiplists exists, shown in Figure 4.2, where each variable-sized array is represented as a 'tower' of nodes where each node has only two outgoing links.

This forms a representation of a skiplist resembling a set of parallel sorted linked lists. Each key in the skiplist is assigned a maximum level $L_{\max}$ when it is inserted, and it will be placed in the level-$L_{\max}$ linked list and all lower-level linked lists.

Maximum levels are assigned using an exponential distribution. The level-0 list contains every list node. The level-ith list contains one in every 2^i list nodes on average. In this example, keys $\{3, 6, 9, 15\}$ are at level 0, key $\{8\}$ is at level 1 and keys $\{5, 11\}$ are at level 3. If the level of a key is chosen deterministically from the key, the skiplist over a set of keys is set-unique. Searching a skiplist involves starting in the upper left and 'skipping' many nodes by using the higher level links. Skiplists offer an expected $O(\log n)$ update time and lookup time. Just as with a treap, the worst-case lookup and update time is $O(n)$.

During lookups, not every edge in a skiplist is used. Extra edges, represented in grey in

Figure 4.2, are only needed for performing updates. Our insight is that *completely omitting* the extra edges lets us store a skiplist as if it were an ordinary binary tree; it can then be managed using any of our persistent tree algorithms or implementations. To this end, we have redesigned the update operations to not require these extra edges.

In Figure 4.3 we present our final representation of this skiplist, similar to the skiplist authentication trees construction of Goodrich et al. [12]. The difference between our constructions is that their construction requires level-0 nodes without right siblings to include the key of their successor. Our improved lookup proof construction, described below, makes that unnecessary.

In addition to a new formulation of skiplists as binary trees, our lookup proofs improve on prior work in authenticated skiplists. Lookup proofs showing membership consist of a path from the root to node containing a lookup key. A lookup proof showing non-membership must prove that the interval between two successor keys in the skiplist does not contain the lookup key.

In the original formulation of authenticated skiplists, non-membership is proved by including the right siblings of each node in the path from the root to the lookup key in the proof. For example, to prove that the key 7 is not in the skiplist in Figure 4.3, the server includes the bold-faced edges along with the $(-\infty, \infty)$ edge at L_3 and the $(5, 11)$ edge at L_2. When proving non-membership of a lookup key that occurs after a level-0 node without a right sibling, the proof of non-membership uses the right successor key stored in that node.

We can improve on this construction. Observe that in a skiplist, the successor of a level-0 node without a right sibling is always the key stored in the right sibling of the first ancestor of that node with a right sibling. If the lookup proof already contains the right sibling of every node in the lookup path, then the successor node is already included in the proof, removing the need for any nodes to explicitly store the keys of their successors. By removing the non-tree-like behavior of storing successor keys, this construction simplifies the design and implementation of update operations.

We can further optimize the proof when the author is trusted to correctly build the skiplist. Instead of including every right sibling in the lookup proof, we only need to include *one* right sibling. If we want to show that a key K is not in the skiplist, we do a search for K. If we find a level L_0 node N with key $k_1 < K$ and a right child containing $k_2 > K$, then by including both N and its right child, we can prove that K is not in the skiplist. If N does not have a right child, then the successor key to k_1 is stored in the right sibling of the first ancestor of N that has a right sibling, if that right sibling has key $k_2 > K$, then K is not in the dictionary. Only this one right sibling needs to be included in the proof. For example, in Figure 4.3, the level L_0 node 6's first ancestor with a right sibling is the level L_1 node 5, whose right sibling contains an 8. This is 6's successor in the skiplist. The highlighted edges and nodes would suffice to prove that the value 7 is absent from the data structure. This optimization makes our construction of a skiplist lookup proof include approximately half of the number of nodes as prior constructions that include right siblings for every node in the lookup path.

4.2.4 Red-black trees

Authenticated dictionaries can also be built based on red-black trees [7], offering $O(1)$ expected node mutations, $O(\log n)$ worst-case update costs, and $O(\log n)$ worst case path length. Red-black trees offer a tighter bound than skiplists or treaps. Red-black costs have a logarithmic worst-case bound, not a logarithmic expected-case bound. Unfortunately, red-black trees are not history independent. Note that for simplicity in reporting results in our evaluation, we may refer to red-black trees as having $O(\log n)$ expected costs, instead of the tighter bound of $O(\log n)$ worst-case costs.

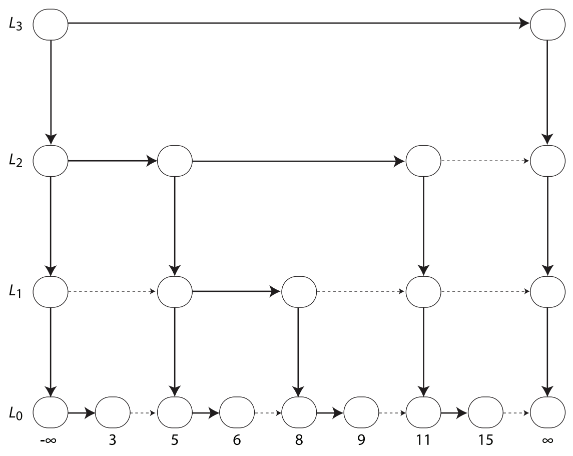

Figure 4.2 : Skiplist representation. Dashed arrows represent redundant edges that are omitted in our implementation.

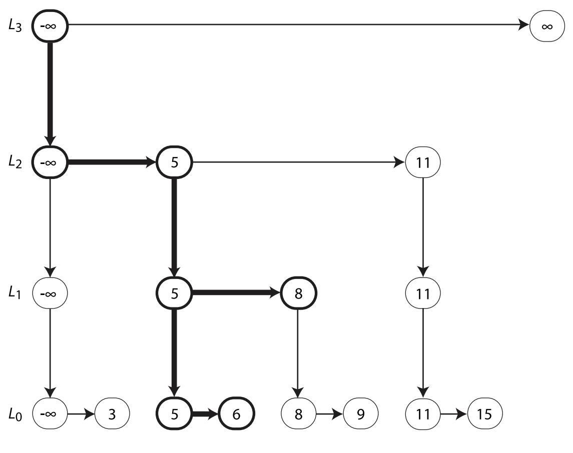

Figure 4.3 : Skiplist query for “7.” Highlighted nodes will be included in the result proof to demonstrate that “7” is absent from the result.

4.2.5 Persistent binary search trees

Persistent search tree data structures extend ordinary search tree data structures to support lookups in past snapshots or versions. Persistent data structures were developed to support these features and have been extensively studied [94, 95], particularly with respect to functional programming [96, 97]. In this section we summarize the algorithms proposed by Sarnak and Tarjan [10], who considered approaches for persistent red-black search trees. Their techniques apply equally well to treaps, red-black trees, or our version of a skiplist (Described in Sections 4.2.2 and 4.2.4 and 4.2.3 respectively.).

Logically, a persistent dictionary built with search trees is simply a forest of trees, i.e., a separate tree for each snapshot. The root of each of these trees is stored in a *snapshot array*, indexed by snapshot version. Historical snapshots are frozen and immutable. The most recent, or *current* snapshot can be updated in place to include inserted or removed keys. Whenever a snapshot is taken, a new root is added to the snapshot array and that snapshot is thereafter immutable.

Three strategies Sarnak and Tarjan proposed for representing the logical forest are *copy everything*, *path copying*, and *versioned nodes*. They range from $O(n)$ space to $O(1)$ space per update. Note that these different physical representations store the same logical forest. The simplest, *copy everything*, copies the entire tree on every snapshot and costs $O(n)$ storage for a snapshot containing n keys.

Path copying uses a standard applicative tree, avoiding the redundant storage of subtrees that are identical across snapshots. Nodes in a path-copying tree are immutable. Where the normal, mutating treap, red-black, or skiplist algorithm would modify a node's children pointers, an applicative tree instead makes a modified clone of the node with the new children pointers. The parent node will also be cloned, with the clone pointing at the new child.

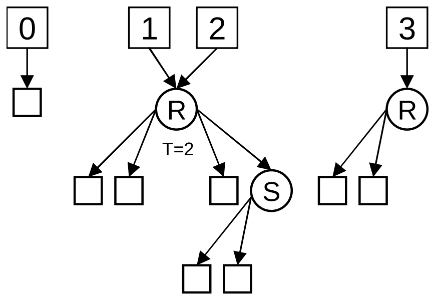

Figure 4.4 : Four snapshots in a Sarnak-Tarjan versioned-node tree, starting with an empty tree, then inserting R, then inserting S, then deleting S. We show the archived children to the left of a node and the current children to the right. Note that R is modified in-place for snapshot 2, but cloned for snapshot 3.

This propagates up to the root, creating a new root. For any of red-black trees, treaps, or skiplists, each update will create $O(1)$ new nodes and $O(\log n)$ cloned nodes in expectation. When a snapshot is taken after every update, skiplists and treaps will use $O(\log n)$ expected storage per update while red-black trees will have a worst-case bound of $O(\log n)$ storage per update.

Versioned nodes are Sarnak and Tarjan’s final technique for implementing partially persistent search trees and can represent the logical forest with $O(1)$ expected amortized storage per update. We will first explain how versioned node trees work and then, in Sect. 4.2.6, we will show how to build these techniques into search trees with Merkle hashes.

Rather than allocating new nodes, as with path copying, versioned nodes may contain pointers to older children as well as the current children. While we could have an infinite set of old children pointers, versioned nodes only track two sets of children (*archived* and *current*) and a *timestamp* T. The archived pointers archive one prior version, with T used to indicate the snapshot time at which the update occurred so that L V’s know whether to use the archived or current children pointers. A versioned node cannot have its children updated twice. If a node x’s children need to be updated a second time, it will be cloned, as in path copying. The clone's children will be set to the new children. *x*'s parent must also be updated to point to the new clone, which may recursively cause it to be cloned as well if its archived pointers were already in use. In Fig. 4.4 we present an example of a versioned node tree.

Each update to a treap or red-black tree requires an expected $O(1)$ rotations, each of which requires updating the children of 2 versioned nodes, requiring a total of $O(1)$ expected amortized storage per update. To support multiple updates within a single snapshot, we include a last-modified version number in each versioned node. If the children pointers of a node are updated several times within the same snapshot, we may update them in place. As with path copying trees, saving a copy of the root node in the snapshot array is sufficient to find the data for subsequent queries.

4.2.6 Making trees persistent and authenticated

A persistent tree is just a forest of individual trees, one for each snapshot, each of which is an independent authenticated dictionary with the proscribed structure of a tree. As each snapshot is an ordinary search tree, tree-based PADs naturally extend to support queries of a given value's successor, predecessor, and so forth.

In a PAD, the author only needs to store or access one search tree, that of the latest snapshot. Trees representing earlier snapshots are not needed to perform updates and thus need not be stored by the author. The server on the other hand needs to be able to store a representation of *every* snapshot's search tree in order to respond to lookup requests from clients.

The choice of how we represent the logical forest of trees is completely invisible to clients and has no effect on nature of a lookup proofs in historical snapshots or on the root authenticator for a snapshot. We can represent the logical forest of trees representing each snapshot using any of the persistent tree algorithms in Section 4.2.5. Different representations have different performance and storage cost tradeoffs, in particular the costs of storing or generating subtree authenticators for elided subtrees, which are needed when generating lookup proofs.

If *copy everything* is used to represent the forest of trees, lookup proofs can be computed in time proportional to the depth of the tree, which is expected to be $O(\log n)$ for treaps and skiplists and $O(\log n)$ in the worst case for red-black trees. Each node occurs in exactly one snapshot and each node can cache its subtree authenticator. When *path copying* is used to represent the forest of trees, each node is immutable once created. The subtree rooted at that node is fixed and the subtree authenticator is constant and can be cached directly on that node. Lookup proofs can be computed in $O(\log n)$ expected time and updates cost $O(\log n)$ expected storage. PADs based on path-copying red-black trees were proposed by Anagnostopoulos et al. [7].

Caching subtree authenticators in Sarnak-Tarjan versioned nodes adds extra complexity. Unlike before, the descendants of a node are no longer immutable and the subtree authenticator of a node is no longer constant for all snapshots in which it occurs. For example, in Fig. 4.4, the node containing R in the version 1 and 2 trees has different authenticators in snapshots 1 and 2. In this section, we present novel techniques for building authenticated data-structures out of persistent data structures based on versioned nodes by controlling when and how subtree authenticators are recomputed or cached. In these designs, each update costs $O(1)$ storage to create new versioned nodes plus whatever overhead is used for caching subtree authenticators.

In our designs, we store subtree authenticators for the current snapshot, mutating it in place on each update to the tree. This *ephemeral subtree authenticator* can be used to generate lookup proofs for the current snapshot in $O(\log n)$ time. For historical snapshots, however, it cannot be used.

For historical snapshots, a simple solution is to not cache any subtree authenticators at all. In this *cache nothing* case, the server can calculate the subtree authenticator for a node on-the-fly from its descendants and generate a lookup proof in $O(n)$ time. Obviously, we want to generate proofs faster than that. By spending additional space to cache the changing subtree authenticators, we can reduce the cost of generating lookup proofs.

Each versioned node can cache the changing authenticator for every version in a *versioned reference* which can be stored as an append-only resizable vector of pairs containing version number transition points v_i and values r_i, $((v_1, r_1), (v_2, r_2), \ldots (v_k, r_k)))$. The reference is undefined for $v < v_1$. The reference is r_1 for $v_1 \leq v < v_2$, r_2 for $v_2 \leq v < v_3$, and so forth. The reference is r_k for versions $\geq v_k$. $r_i = \square$ means that the cache is invalid and the subtree authenticator must be recomputed by visiting the node's children. Lookups by version number use binary search over the vector in $O(\log k)$ time in the worst case. Only $O(1)$ time is required to update each cache if we copy subtree authenticator from the ephemeral cache at the end of the snapshot.

Note that in this cache design, the most recently cached subtree authenticator remains valid forever. If a cached subtree authenticator is about to becomes stale, the authenticator cache must be either updated with the new subtree authenticator, or explicitly invalidated for the next snapshot. Note that if the authenticator cache is invalidated for the next snapshot, it remains valid for prior snapshots. Similar updates will also be necessary for the authenticator caches in the modified node's ancestors.

Our first caching option, *cache everything*, ensures that the authenticator cache always hits. On each update to the tree, we update the cache for each node in the path to the root. This means that we lose the $O(1)$ benefit of using versioned nodes, because we must pay a $O(\log n)$ expected cost to maintain the cached authenticators for treaps and skiplists or a $O(\log n)$ worst-case cost for red-black trees. Generating a lookup proof will cost $O(\log v \cdot \log n)$ time in expectation for a $O(\log n)$ expected binary searches in the subtree authenticator cache. In the example presented in Fig. 4.4, the nodes containing R in the version 1 and 3 trees have 2 and 1 cached authenticators respectively. The node containing S has 1 cached subtree authenticator.

Although PADs implemented by versioned nodes implemented using the cache-everything strategy have the same big-O space usage as PADs implemented by trees that use path copying, the constant factors are smaller. Appending another hash and timestamp threshold to $O(\log n)$ versioned references implemented by resizable arrays is much more concise than cloning $O(\log n)$ nodes.

We are not required to cache every subtree authenticator. Authenticators may be recomputed as needed, offering a diverse set of choices for caching strategies and time-space tradeoffs. Caching strategies may be generic, or exploit spacial or temporal locality, as long as a cached authenticator is updated or invalidated in any snapshot where a descendant changes. Caching strategies may also purge authenticators at any time to save space. Although many application-specific strategies are possible, we will only present one generic caching strategy with provable bounds.

Our *median layer cache* offers $O(1)$ storage per update while generating lookup proofs in historical snapshots in $O(\sqrt{n} \log v)$ time in expectation by permanently caching subtree authenticators on exactly those nodes at depth D chosen to be close to the median layer $\frac{\log_2 n}{2}$ in the tree. As nodes enter or leave the median layer, or the median layer itself changes, we maintain the invariant that for each snapshot, the versioned nodes in the median layer for that particular snapshot have cached authenticators.

When an update occurs, in the typical case where only leaves' values change, we update the subtree authenticator cache in the ancestor median layer node. In addition, all other ancestors of the changed node potentially have stale authenticators, forcing us to explicitly invalidate their caches for the upcoming snapshot. In the atypical case, many nodes may enter or leave the median layer at a time, due to changes of the number of keys in the tree or rotations among the first D layers of the tree. However, only $O(1)$ expected additional storage per-update is required to account for these effects.

Computing lookup proofs for the median layer tree can be done in $O(\sqrt{n}\log v)$ time in expectation. Generating a lookup proof requires calculating $O(\log n)$ subtree authenticators in expectation at depths $d = 1, d = 2, \ldots, d = O(\log n)$. (Recall that $D = \log_2 \sqrt{n}$.) There are three cases for computing any one single subtree authenticator. In the first case, the subtree authenticator for a node at depth $d = D$ is cached and can be used directly.

Computing a subtree authenticator for a node x at depth $d < D$ (i.e., x is higher than the median layer, closer to the root), requires recursing down until hitting nodes at the median layer, then using the cached authenticators. This recursion will visit at most $2^{D-d} = O\left(\frac{\sqrt{n}}{2^d}\right)$ nodes. Computing a subtree authenticator for a node x at depth $d > D$ (i.e., x is below the median layer, closer to the leaves) requires visiting every descendant of x. In expectation, a node at depth $d > D$ has $O\left(\frac{n}{2^d}\right) = O\left(\frac{\sqrt{n}}{2^{d-D}}\right)$ descendants.

4.2.7 Details on the median layer cache

This section includes further detail on the design and expected running time of a subtree authenticator cache that caches on the median layer of a treap. We maintain the invariant that in *every* snapshot of the tree, we store the subtree authenticator in the persistent cache for *every* node at depth m, with $\frac{\log_2 n}{2} - 1 \leq m \leq \frac{\log_2 n}{2} + 1$ where n is the number of keys. m is allowed to vary within a range to add a hysteresis as to when the median layer changes. Note that there are at most $\sqrt{n/4} \leq 2^m \leq \sqrt{n * 4}$ nodes at depth m. Additionally, as before, each node also stores an ephemeral subtree authenticator.

When updating the tree, we use amortized expected time bounds because many nodes may require wholesale updating to maintain the invariant with rotations changing depth of a subtree or the median layer changing due to changing key-count. We evaluate each of several scenario's that involve refreshing the cache, calculate the expected number of cache entries modified, and the probability of the scenario. In expectation, the total storage cost is $O(n+v)$. Our analysis assumes that each subtree underneath a median layer node has the same nodecount.

The *top* of the tree refers to any node occurring in the first m layers, between the root and median cache. The *bottom* of the tree is any node at or below depth m.

Median layer changing due to increasing or decreasing keycount We only change the median layer whenever the keycount increases or decreases by a factor of 4, which can occur no more frequently than once every $\left(1-\frac{1}{4}\right)n$ updates. This requires invalidating every median node's current cache using $O(\sqrt{n})$ space, and storing new values in the cache using $O(\sqrt{n})$ space, requiring $O(n)$ time. Amortized, this comes to $O(1/\sqrt{n})$ space and $O(1)$ time added onto any insert or delete operation described below.

Inserting into the bottom of the tree Keys are inserted by placing them as a leaf (in key-order), then rotating them to the proper depth, based on priority. If a new key ends up at depth d in the treap, no node of depth $< d$ will have its depth altered during the rotations. The probability that a new key being inserted ends up at depth d is the probability that its priority is greater than $2^d/n$ fraction of the keys in the treap. With probability $1-O(1/\sqrt{n})$, a key will end up with a final depth $d \geq m$. The rotations to insert the key will be constrained to a subtree of the median layer node, requiring only one persistent cache insert, for the one median layer node that is an ancestor of the inserted key. In expectation, this involves $O(1)$

updates to the persistent cache, plus the constant expected costs managing the persistent treap.

Inserting into the top of the tree If the key that is inserted ends up at depth $d < m$, then the rotations during the insert will alter the depths of every descendant of the newly inserted node, including $2^m/2^d$ median layer nodes. The probability of a depth d insert is $2^d/n$, requiring $2^m/2^d$ persistent cache inserts. Summing over $d \in [0, m]$, the expected number of updates to the persistent cache is $m * 2^m/n = m/O(\sqrt{n}) \lll O(1)$, plus the constant expected costs in managing the persistent treap.

Deleting a key stored in the bottom of the tree The analysis of deletions is the same as the analysis of insertions above. The probability of a key being deleted being at depth d is $2^d/n$, and all of the rotations involved in the deletion will be constrained to that subtree. With probability $1 - O(1/\sqrt{n})$, a key will end up with a final depth $d \geq m$, and deleting will only require one median layer node to have its cache updated. In expectation, this involves $O(1)$ updates to the persistent cache, plus the constant expected costs of managing the persistent treap.

Deleting a key stored in the top of the tree If the key is deleted is at depth $d < m$, then the rotations involved will alter the depths of every descendant of the to-be-deleted node. The calculation and result as for inserts, the expected number of updates to the persistent cache is $m * 2^m/n = m/O(\sqrt{n}) \lll O(1)$, plus the constant expected costs in managing the persistent treap.

Cost of a lookup Lookup costs are $O(sqrtn)$. Their calculation was described above.

Result

Any update to the persistent authenticated dictionary incurs $O(1)$ storage cost per update for the persistent tree and $O(1)$ storage into the persistent cache. The persistent cache is soft-state, the specific layer chosen as the median layer only affects the computation complexity. In fact, all of the storage choices we discuss in this section are just different ways to store the forest of closely related snapshotted treaps, and different ways the subtree authenticator cache is managed. These choices have *no effect upon the generated authenticators and proofs*. All of the approaches we propose are *equivalent* to each other in output, and only differ in storage and performance as we can always reconstruct a missing subtree authenticator from the keys and values in the subtree.

4.3 Tuple-based PADs

Previously, we described how to design PADs based on Merkle trees, offering constant update size and logarithmic update time. In this section, we develop a novel alternative foundation. These designs are super-efficient, yielding constant-sized query response proofs instead of the $O(\log n)$ proofs from tree-based PADs. The tradeoff is that tuple PAD updates are much more expensive. In addition, these PADs offer different features, functionality, and efficiency choices.

This class of techniques uses a *tuple representation* of a dictionary. If a dictionary has keys $k_1 \ldots k_n$, with $k_i < k_{i+1}$ and corresponding values $c_1 \ldots c_n$, we subdivide the entire key-ID space into disjoint intervals $[k_0, k_1), [k_1, k_2)$, and so forth. Each interval $[k_j, k_{j+1})$ contains a single dictionary key at k_j with value c_j and indicates that there is no other key elsewhere in the interval. Let this be represented as the tuple $([k_j, k_{j+1}), c_j)$, which we can formally read as: "Key k_j has value c_j, and there are no keys in the dictionary in the

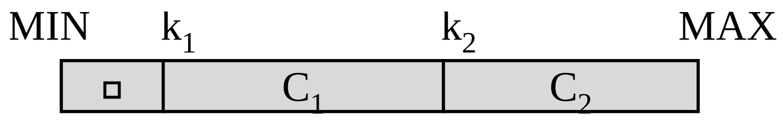

Figure 4.5 : Tuple authenticated dictionary showing 2 keys and 3 tuples. Tuple $([k_j, k_{j+1}), c_j)$ is represented as a rectangle from k_j to k_{j+1} containing c_j.

interval (k_j, k_{j+1})." Keys could be integers, strings, hash values, or any type that admits a total ordering. In order to cover the key-ID space before the first key k_1 and after the last key k_n in the dictionary, we include two sentinels, $([k_{\min}, k_1), \square)$ and $([k_n, k_{\max}), c_n)$ where $k_{\min}$ and $k_{\max}$ denote the lowest and highest key-IDs respectively. An alternative would use a circular key-ID space rather than the sentinels. Figure 4.5 illustrates the tuples composing a dictionary.

If each tuple is individually signed by an author to form an authenticated dictionary, then the server can prove the presence or absence of a key k_q from the authenticated dictionary by returning the one signed tuple $T = ([k_j, k_{j+1}), c_j)$ that *matches* k_q by being responsible for the section of the key-space containing k_q, or, more formally, having $k_q \in [k_j, k_{j+1})$. The key k_q is in the dictionary with value c_j if $k_q = k_j$ and $c_j \neq \square$ ($\square$ denotes no key). If $k_q \neq k_j$, the client may conclude k_q is absent from the dictionary. This representation offers super-efficient, $O(1)$, lookup proofs of membership and nonmembership of a key in the dictionary.

Now that we have explained the tuple representation of a single authenticated dictionary, the challenges are how to add persistence, how to efficiently store the tuples and their signatures, how to reduce the number of tuples that need to be signed, and finally how to authenticate tuples without individually signing each one.

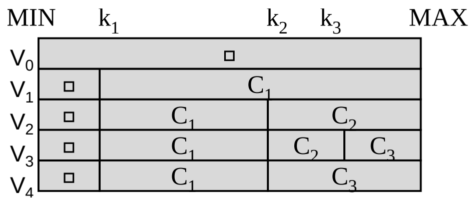

Figure 4.6 : Tuple PAD containing 5 snapshots. From top to bottom, starting with an empty PAD, inserting k_1, c_1, inserting k_2, c_2, inserting k_3, c_3, and removing k_2. Each rectangle corresponds to a signed tuple.

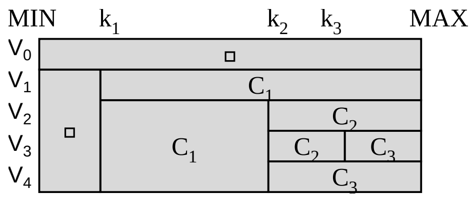

Figure 4.7 : Example of tuple-superseding representation of Fig. 4.6, showing the space savings when tuples can span many version numbers. As before, each rectangle corresponds to a signed tuple.

4.3.1 PADs based on individually signed tuples

In a solitary PAD, each tuple is individually signed by the author. The author signs $n + 1$ tuples for each snapshot. To support persistency, tuples include a version number and have the form: $(v_\alpha, [k_j, k_{j+1}), c_j)$, which can be read as "In version v_α, key k_j has contents c_j, and there is no key in the dictionary with a key between k_j and k_{j+1}." Figure 4.6 graphically shows such a PAD. The server can prove the membership or non-membership of any key k_q in snapshot v_q in the PAD by returning one signed tuple $T = (v_q, [k_j, k_{j+1}), c_j)$ that matches the lookup request by having $k_q \in [k_j, k_{j+1})$. This design is super-efficient, persistent and history independent, but does not have a root authenticator or support Merkle aggregation.

Updates are clearly expensive. The author must sign each tuple individually on each snapshot and send the signatures to the server, which must then store them. The per-snapshot computation, storage, and communications costs are $O(n)$.

Optimizing storage by coalescing tuples. If we assume that a snapshot is generated after every update, all but at most two of the signed tuples in snapshot v_α will have the same keys and values in snapshot $v_\alpha + 1$. This is because an insert into the dictionary will split the range of the prior tuple into two ranges. Removing a key will require deleting a tuple and replacing its predecessor tuple with a new one with an expanded range.

Most tuples may remain unchanged across many snapshots. Instead of storing each of the tuples, $(v_\alpha, [k_j, k_{j+1}), c_j)$, $(v_\alpha + 1, [k_j, k_{j+1}), c_j)$, $\ldots$ $(v_\alpha + \delta, [k_j, k_{j+1}), c_j)$, and signatures on each of these tuples, the server may store one *coalesced tuple* $([v_\alpha, v_\alpha + \delta], [k_j, k_{j+1}), c_j, SIGS)$ that encodes that the key space from k_j to k_{j+1} did not change from snapshot v_α to $v_\alpha + \delta$. In each coalesced tuple, *SIGS* stores the $\delta + 1$ signatures signing each individual snapshot's tuple. The coalesced tuple, itself, is never signed.

Upon a lookup query for k_q at time v_q, the server finds the tuple $T = ([v_\alpha, v_\alpha + \delta], [k_j, k_{j+1}), c_j, SIGS)$ that matches k_q and v_q by having $v_q \in [v_\alpha, v_\alpha + \delta]$ and $k_q \in [k_j, k_{j+1})$, from which it regenerates the tuple $(v_q, [k_j, k_{j+1}), c_j)$, which the author signed earlier.

Storing tuples with a persistent search tree. Our next challenge is how to store coalesced tuples and signatures so that they may be easily found during lookups. We need a data structure that can store the varying set of coalesced tuples representing each snapshot, and for any given snapshot version, we need to be able to find the tuple containing a search key. This can be easily done with a persistent search tree that supports predecessor queries, such as the $O(1)$ persistent search tree data structure described in Section 4.2.5.

Each snapshot in the PAD has a corresponding snapshot in the persistent search tree *PST* for storing the tuples representing that snapshot. Whenever an update occurs, the author will indicate which tuples are *new* (i.e., their key interval or value was not in the prior snapshot), and which tuples are to be *deleted* (i.e., their key interval or value is not in the new snapshot). The remaining tuples are *refreshed*. At most two tuples will be deleted and one tuple will be new. The author transmits signatures on every new or refreshed tuple.

When a tuple $([v_\alpha, v_\beta], [k_j, k_{j+1}), c_j, SIGS)$ is to be deleted from snapshot $v_\beta + 1$, the server removes that tuple from the next snapshot of *PST*. When a tuple is to be added to snapshot $v_\beta + 1$, the server inserts $([v_\beta + 1, v_\beta + 1], [k_j, k_{j+1}), c_j, SIG)$ into PST. If a tuple $T = ([v_\alpha, v_\beta], [k_j, k_{j+1}), c_j)$ is refreshed, the server appends the author's signature to T and updates the ending snapshot version to $v_\beta + 1$.

This data-structure requires $O(1)$ storage per update for managing the coalesced tuples representing the PAD and can find the matching coalesced tuple and signature for any key in any snapshot in logarithmic time. Unfortunately, the additional costs of $O(n)$ signatures for every snapshot must also be included in the communication and storage costs. Reducing these costs is the challenge in building tuple-based PADs.

4.3.2 Optimizing storage: Tuple superseding

We now show how to reduce storage costs on the server from $O(n)$ to $O(1)$ signatures per snapshot. Previously, authors signed tuples of the form $(v_\alpha, [k_j, k_{j+1}), c_j)$ for each snapshot. With tuple superseding, the author signs a coalesced tuple of the form $([v_\alpha, v_\beta], [k_j, k_{j+1}), c_j)$ attesting that for all snapshots in $[v_\alpha, v_\beta]$, key k_j has value c_j and there is no key in the interval (k_j, k_{j+1}). Figure 4.7 shows the benefits of tuple superseding, when a signature can span many version numbers. Clients authenticating a response to a query k_q in snapshot v_q will receive a tuple of the form $([v_\alpha, v_\beta], [k_j, k_{j+1}), c_j)$. They will verify that its signature is valid and that $k_q \in [k_j, k_{j+1})$ and $v_q \in [v_\alpha, v_\beta]$.

For tuples that are refreshed, the server will receive a tuple $([v_\alpha, v_\beta + 1], [k_j, k_{j+1}), c_j)$, signed by the author. This newly signed tuple supersedes the signed tuple $([v_\alpha, v_\beta], [k_j, k_{j+1}), c_j)$ already possessed by the server and can transparently replace it. Although the author must sign $O(n)$ tuples and send them to the server for each snapshot, all but $O(1)$ of them refresh existing tuples. Only the $O(1)$ new tuples and their signatures add to storage on the server. When tuple superseding is used, the PAD is no longer history independent because the signed tuples describe keys in earlier snapshots.

Iterated hash functions.

Public key signatures are notably slow to generate and verify. In contrast, cryptographic hash functions are very fast. With a light-weight signature [98] implemented by iterated hash functions, we can indicate that a tuple is refreshed. Rather than signing each superseded tuple, the author now only signs the tuple: $(v_\alpha, H^m(R), [k_j, k_{j+1}), c_j)$ where $H^m(R)$ represent the result of iterating a hash function m times on a random nonce R. The author can indicate that a tuple is refreshed in successive snapshots by releasing successive preimages of $H^m(R)$ which it can incrementally generate in $O(1)$ time and $O(\log m)$ space. A client will need to verify at most m hashes, which will still be significantly cheaper than the cost of verifying the digital signature for reasonable values of m.

4.3.3 Optimizing signatures via speculation

Speculation has been previously used to optimize byzantine fault tolerance [66]. In this section we show how a novel application of *speculation* can be used to significantly reduce the number of needed signatures by exploiting redundancy between snapshots. In our original design, the author was required to sign every tuple to refresh it for a new snapshot, at a cost proportional to the number of keys in that snapshot. We can improve on this by dividing the PAD P into two generations: a young generation G_0 that contains keys that are recently modified, and an old generation G_1 that contains all other keys. Tuples in the old generation G_1 are speculatively signed with version intervals that stretch into the future, but are only considered when there is a proof that the key is not set in the younger generation. (Section 4.3.1 noted that it's trivial to prove the absence of a key by returning the signed tuple for the interval containing that key.) Effectively, G_0 contains "patch" tuples that can correct erroneous speculations in G_1. Tuples now include generation markers, g_0 or g_1, to indicate which generation they're in. In Fig. 4.8 we present such a speculative PAD with an epoch of 3 snapshots.

A snapshot of G_0 must be taken every time a snapshot is taken of P, which requires signing every new or refreshed tuple in G_0. To reduce these costs, we keep the size of G_0 small by dividing time into *epochs*. Every E_1 times a snapshot is taken of P, we migrate all of the entries from G_0 into G_1, take a snapshot of G_1, and erase G_0. With a snapshot taken after every update, this ensures that G_0 contains at most $E_1 + 1$ tuples.

When an insert into P is requested, the author inserts the tuple representing the key and value into G_0. When a removal of k_j from P is requested, G_0 is updated to store the tuple $(g_0, [v_\beta, v_\beta], [k_j, k_{j+1}), \square)$, indicating that key k_j is not in the PAD in version v_β.

Tuples in G_0 have the form $(g_0, [v_\beta, v_\beta], [k_j, k_{j+1}), \square)$, indicating the one version that they are valid for, while tuples in G_1 have the form, $(g_1, [v_\gamma, v_\gamma + E_1 - 1], [k_j, k'_{j+1}), c'_j)$, indicating that they are valid for the duration of an epoch. At the start of every epoch, the author enumerates every key-value pair in the current snapshot in G_0, and inserts them into G_1. During this process, the author may find opportunities to merge tuples representing deleted keys. If a tuple $(g_0, [v_\beta - 1, v_\beta - 1], [k_j, k_{j+1}), \square)$ representing a removed key is migrated, it may force the deletion of a tuple, $(g_1, [v_\beta - E_1, v_\beta - 1], [k_j, k'_{j+1}), c'_j)$, in G_1 from the next epoch. After migrating keys into G_1, the author speculatively signs each tuple in G_1 as valid for the entire duration of the future epoch.

On a lookup of key k_q in snapshot v_q, the server returns two signed tuples: $(g_0, v_\beta, [k_j, k_{j+1}), c_j)$ with $v_q = v_\beta$ and $k_q \in [k_j, k_{j+1})$ and $(g_1, [v_\gamma, v_\gamma + E_1 - 1], [k'_j, k'_{j+1}), c'_j)$ with $v_q \in [v_\gamma, v_\gamma + E_1 - 1]$ and $k_q \in [k'_j, k'_{j+1})$. There are two cases. If $k_q = k_j$, then the key is in G_0 with value c_j, with $c_j = \square$ denoting a deleted key. Otherwise, if $k_q \in (k_j, k_{j+1})$, we must examine G_1. If $k_q = k'_j$, then the key is in G_1 with value c'_j. Otherwise, if $k_q \in (k'_j, k'_{j+1})$ then the lookup key is not in the snapshot.

Speculation can reduce the number of signatures required by the author from $O(n)$ to

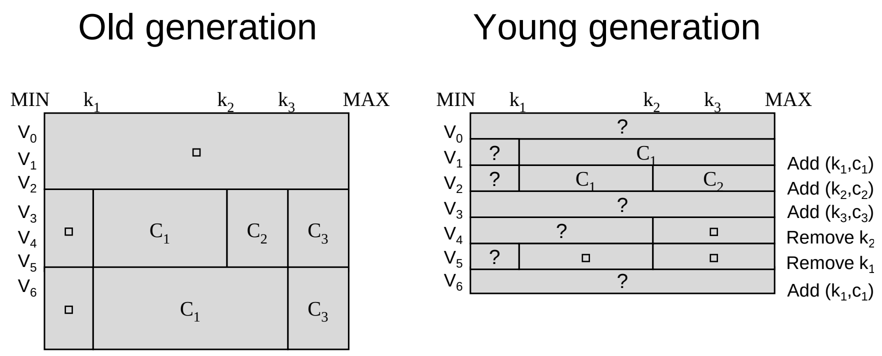

Figure 4.8 : Example of a PAD using speculation with an epoch of 3 snapshots. Lookups examine the young generation first. Because we did not use a circular ID-space the sentinal tuple in the young generation uses a key of ? to indicate that the older generation must be examined for $k_q = k_{MIN}$.

$O(\sqrt{n})$ amortized for each update if we assume a snapshot is taken after every update. The author must sign $E_1 + 1$ tuples in G_0 each time P has a snapshot taken, and, once every E_1 snapshots, the author must sign all $n + 1$ tuples in G_1. The amortized number of signatures per update is $O(E_1 + n/E_1)$, with a minimum when $E_1 = \sqrt{n}$. If DSA signatures are used, latency can be reduced at the start of an epoch by partially precomputing signatures [99]. This creates a super-efficient, history-independent PAD with $O(\sqrt{n})$ amortized signatures and $O(\sqrt{n})$ storage per update. Note that speculation makes a PAD no longer history independent because the tuples in G_1 describe keys contained in the PAD at the start of the epoch.

More than two generations. Speculative PADs can be extended to more than two generations. As before, generation G_0 is definitive, and later generations are progressively more speculative. Lookup proofs will include one tuple per generation.

In the case of 3 generations, we have epochs every E_1 snapshots, when keys are migrated from G_0 to G_1, and every E_2 snapshots, when keys are migrated from G_1 to G_2. If we assume a snapshot after every update, the author must sign an amortized $O\left(\frac{n}{E2} + \frac{E2}{E_1} + E_1\right)$

tuples per update. This is minimized to $O(\sqrt[3]{n})$ when $E_2 = n^{\frac{2}{3}}$ and $E_1 = n^{\frac{1}{3}}$. More generally, if there are C generations, lookup proofs contain C signatures, the author must sign a $O(C \sqrt[C]{n})$ tuples, and the storage per update is $O(C \sqrt[C]{n})$ if tuple superseding is not used.

Speculation and tuple superseding. Speculation reduces the total number of signatures by the author and thus reduces the space required on the server to store them. It can be naturally combined with tuple-superseding (with our without using iterated hashes) to reduce the number of tuples the server must save to $O(C)$ per update.

4.3.4 Tuple PADs based on RSA accumulators

RSA accumulators [36] are a useful way to authenticate a set with a concise $O(1)$ summary, which can be signed using digital signatures. Membership of an element in the set is proved with a constant-sized *witness*, which may be computed by the untrusted server. Recent developments include an accumulator supporting efficient non-membership proofs [100] or batch update of witnesses [101, 102].

By storing tuples in a signed accumulator, the update size for a snapshot can be reduced to $O(1)$ while supporting a root authenticator. In this section we design such a PAD offering constant update size, constant storage per update, constant proof size, and sublinear computation per update, all by using accumulator techniques.

Background Accumulators use RSA exponentiation to generate an integer that authenticates a set. The server proves that an element is in the set by sending the item in question, the accumulator as signed by the author, and the witness.

Consider storing a set of e r-bit prime numbers $p_1 \ldots p_e$. The accumulator storing these keys works as follows: The author selects an s-bit modulus $N = pq$ and a generator g with $s > 3r$. p and q are strong primes, and g is a quadratic residue mod N. p and q are kept secret. The RSA accumulator A over this set is $g^{p_1 \cdots p_e}$. The accumulator A is then signed. To prove that a key k_i is in the set, the server supplies a witness $W_i = g^{p_1 p_2 \cdots p_{i-1} p_{i+1} \cdots p_e}$. (To prevent keys from having a mathematical relationship with one other, prime numbers must be used to represent the set members.)

The author, with its knowledge of the factorization of N, may insert or remove keys from the accumulator with $O(1)$ exponentiations per update. Witnesses can be computed by an untrusted server without the knowledge of any secrets. The witness for any single key can be computed with $O(e)$ exponentiations and the set of all witnesses can be computed with an $O(e \log e)$ algorithm [103].

A membership proof that prime p_i is in the set, consists of (A, W_i, p_i), and the author's signature on A. The proof is verified by checking the signature on A and that $A = (W_i)^{p_i}$. By the Strong RSA Assumption [103], it is hard for a computationally bounded adversary to find $y > 1$ such that $g^y = A \mod N$ without knowing the factorization of N.

Baric and Pfitzmann [103] observed that we can generate *prime representatives* for arbitrary keys in the random oracle model by cryptographically hashing the key and then appending a fixed number t of extra bits. t is chosen such that there is a prime number in $[2^t(X), 2^t(X + 1))$ with high probability. The value of those extra bits is chosen such that the concatenation is a prime number. Inputs for which this is not possible cannot be stored in the RSA accumulator. Papamanthou et al. [39] recently implemented an authenticated hash table following this design.

In our design, we require that the conversion from a hash value into a prime representative is deterministic. This ensures that the RSA accumulator for a given set is uniquely defined by the inputs to the set and can be recomputed from the keys being inserted. To do this, we follow Baric and Pfitzmann [103], testing successive integers until we find a prime number.

Design By cryptographically hashing tuples and then converting them into prime representatives, we can use RSA accumulators to authenticate a set of tuples as a single $O(1)$ accumulator that can then be bound to the version number and signed by the author. Define $A(v_q)$ to be the accumulator value for version v_q. $A(v_q)$ authenticates tuples of the form $([k_j, k_{j+1}), c_j)$ containing a key range and a contents. These tuples can omit the version number v_q because it is in the signature over the accumulator.

Each update to a PAD now only requires adding or removing at most $O(1)$ tuples. The accumulator for the next snapshot, $A(v_{q+1})$, can be computed by incrementally modifying $A(v_q)$ at a cost of $O(1)$ exponentiations per dictionary update to add or remove tuples. Updates require $O(1)$ communication; the author sends the keys being inserted or removed from the PAD, the new accumulator, and the signature. Storage increases by only $O(1)$ per update for storing the updated key. The server could compute witnesses lazily upon lookup requests at a cost of $O(n)$ exponentiations, using no additional storage. Alternatively the server can expend $O(n)$ additional storage per-snapshot for precomputed witnesses. The server can precompute witnesses by itself with $n \log_2 n$ exponentiations or the author can incrementally update the n witnesses in $O(n)$ time and send them along with the update.

When a server receives a lookup request from a client for key k_q in snapshot v_q, the server returns the accumulator $A(v_q)$, bound to the version number v_q and signed by the author, a tuple $T = ([k_j, k_{j+1}), c_j)$ with $k_q \in [k_j, k_{j+1})$, prime representative p_i, and a witness for tuple i in snapshot (v_q, W_{i,v_q}). The client verifies that the prime representative corresponds to the returned tuple, $\left\lfloor \frac{p_i}{2^t} \right\rfloor = H(T)$, that the accumulator authenticates the tuple, $(W_{i,v_q})^{p_i} = A(v_q)$, and that the signature on the accumulator is valid.

Unlike standard accumulator schemes, this representation offers super-efficient proofs of non-membership. The tuple $T = ([k_j, k_{j+1}), c_j)$ attests that there is no key in the interval (k_j, k_{j+1}) is in the set.

Speculation and witness computation. Accumulator-based tuple PADs can be combined with speculation, as described in Section 4.3.3. This increases the size of a lookup proof to $O(C)$ but reduces the costs of witness computation from $O(n \log n)$ to $O\left((C+1)\sqrt[C]{n}\right)$ exponentiations per update.

Rather than individually sign each generation's accumulator $A(G_0, v), A(G_1, v)$ and so forth, we could instead collect these accumulators into a short hash chain $B(v) = H(A(G_0, v), H(A(G_1, v), H(A(G_2, v) \ldots)))$, and then bind the root of this hash chain, $B(v)$, to its version number and sign it. However, signing each generation individually only uses $1 + \frac{1}{\sqrt[C]{n}}$ times more signatures than using a hash chain.

On each update to the PAD, the author performs $O(C)$ amortized exponentiations, one to update the accumulator for G_0, and the remaining exponentiations account for the amortized costs of updating the accumulators for the other generations. The author then transmits the update and the new signed $B(v+1)$ to the server, who can deterministically update its copy of the PAD.

When using speculation, only G_0, containing $O(\sqrt[C]{n})$ tuples, is updated on every snapshot. The amortized cost for computing witnesses over all generations using the $O(e \log e)$ algorithm is $O((C+1)\sqrt[C]{n} * \log n)$. The server must store these witnesses at an amortized cost of $O(C\sqrt[C]{n})$ per update to the PAD.

Accumulators and tuple superseding. When we first discussed tuple superseding, in Section 4.3.2, it was used to reduce the signature storage on the server. This same principal may be applied to witness storage on the server for accumulators.

We alter the tuples stored in the accumulator to include the version number when they are created, e.g., $(v_q, [k_j, k_{j+1}), c_j)$. If the accumulator $A(v_{q+\delta})$ contains that tuple and is signed by the author, we consider the tuple to be valid for all versions $v \in [v_q, v_{q+\delta}]$. Thus, when a client queries for a key k in snapshot $v_{q'}$ (where $k \in [k_j, k_{j+1})$), the server may send as a proof a signed $A(v_q)$, the tuple $T = (v_q, [k_j, k_{j+1}), c_j)$ with $k \in [k_j, k_{j+1})$ and $v_{q'} \in [v_q, v_{q+\delta}]$, and a witness proving that $T \in A(v_{q+\delta})$. The same response can authenticate any version $v_{q'} \in [v_q, v_{q+\delta}]$. Instead of storing one witness for each snapshot, the server now can store only one witness, the one in $A(v_{q+\delta})$ that authenticates T.

As before, we assume a snapshot is taken after every update. Just as the situation described in Section 4.3.2, each time a snapshot occurs the server must generate a full set of witnesses. At most two of those witnesses will be for newly created tuples. The remaining witnesses are for refreshed tuples and can be superseded and replace the witnesses previously stored. Computation cost is the same, but the per-update storage costs drop to $O(1)$.

Accumulators, tuple superseding, and speculation can be combined to form our penultimate PAD design, offering constant time on the author per update, constant communication per update, constant storage per update on the server and constant lookup proof size. Computing a new set of witnesses is sublinear in the number n of keys in the pad at $O((C + 1)\sqrt[C]{n})$ exponentiations per update. Unlike before, we individually sign each generation's accumulator in order to independently choose witnesses from different snapshots for each generation.

4.4 Scalability

We expect that the server may well be called upon to scale to run on large clusters and support much higher insertion and query rates. This section considers scalability issues for such environments and how our algorithms could be modified to run faster in such environments.

Faster server insertion rates. Keys exist in a large key space. We can partition that key space across a large cluster of machines, with each server responsible for only a fraction of the key space (much as is standard practice in distributed hash table implementations). Each server then maintains that fraction of the PAD. Assuming keys are uniformly distributed across the key space, each server should see a uniform fraction of the load. To guarantee this uniformity, keys could be hashed before being stored in the PAD.

For any tuple PAD implementation, without RSA accumulators, this split is quite natural. Different servers can be responsible for different key ranges, allowing for excellent scalability. For tree PADs, each server would be responsible for a different subtree, but coordination would be required for changes to the shared top levels of the tree.

Faster client query rates. Client queries require no mutation of state on the server. As such, server state may be arbitrarily replicated to support larger client query rates. This would require inbound mutation operations from authors to be distributed to each replica responsible for any given key.

Lots of snapshots. While some measure of coordination is required, as above, to handle the most current version of a PAD, older versions are static. In a large server cluster, older snapshots can be replicated onto dedicated machines. Any given range of keys from any given snapshot can be stored on multiple, different servers, allowing for excellent scalability both in terms of storage capacity and supported client query rates.

Faster authors. Presently, we assume that the "author" is running on a single computer, but we could imagine a large number of machines, sharing the author's crypto key material, concurrently authoring a PAD. Assuming the server is ready to support the higher insertion rate, as above, the challenge is to coordinate all the author nodes. For modest scalability, a single-threaded author can control the tree or tuple layout, delegating expensive cryptographic computations to other nodes in its cluster. If DSA signatures are used, latency can be further reduced by having author nodes partially precompute signatures [99].

For broader scaling, the author nodes could follow a partitioning strategy, similar to that described for the server. Again, this partitioning is quite natural with tuple PADs and will require coordination of the higher layers in the tree for tree PADs.

4.5 Future work, applications, and extensions

PADs are suitable for a variety of problems, such as in a public key infrastructure where they can efficiently store a constantly-changing set of valid certificates. If a PAD supporting a root authenticator is used, the root authenticator may be stored in a tamper-evident log such as the one described in Chapter 3 and the author cannot later modify it without detection. Similarly, the root authenticator could be submitted to a time-stamping service [54,57] every time a snapshot is taken to prove its existence. PADs can be used to implement many forms of outsourced databases. Using Merkle aggregation, PADs can be used to implement flexible query languages, or in the case of Pari-mutuel gambling, as used in horse racing, to count wagers. With a canonical or history independent representation, PADs can make distributed algorithms more robust.

In this work we developed several new ways of implementing PADs. We presented designs offering constant-sized proofs and lower storage overheads. We also developed speculation as a new technique for designing authenticated data structures. Future work in PAD designs could include creating fully persistent authenticated dictionaries based on fully persistent data structures [86].

In the next chapter, we will perform an evaluation of each of our algorithms and of their respective costs for each operation in order to guide which algorithm is right for which situation. We will also compare our designs to alternative PAD algorithms [7] and evaluate PADs based on RSA accumulators and other cryptographic techniques.

Chapter 5

Performance analysis of PADs

In this chapter, we describe our implementation of 21 different PAD algorithms, including prior designs based on Merkle trees [7] and the designs described in Chapter 4. Our evaluation includes both a big-O analysis and includes benchmarks of an implementation of each algorithm. For each algorithm we measure the time, space and communication overheads, determining real-world performance including the constant factors of digital signature generation, modular exponentiation, primality testing, serialization, and so forth. In this evaluation, for simplicity, we assume a snapshot is taken of the dictionary after every update.

The PAD algorithms we built make different tradeoffs of CPU, bandwidth, and storage requirements. The ideal algorithm for any given workload will thus depend on the relative costs of these resources. Rather than guess at these tradeoffs, we instead normalize them using contemporary costs, in U.S. Dollars, charged by Google and Amazon for bandwidth, CPU time, and storage on their EC2 and AppEngine services. If we assume that Google and Amazon are offering these resources at their marginal cost, i.e., that their rates charged for bandwidth, CPU time, and storage are close to the actual costs to any provider delivering large quantities of these resources, then our evaluation strategy should generalize to other vendors as well.

In Section 5.1, we compare the big-O times for the different algorithms against each other. In Section 5.2, we describe our PAD implementations and evaluation methodology. Section 5.3 presents benchmark results for our tree PAD implementations. Section 5.4

presents benchmark results for our tuple-based PAD implementations, including the RSA accumulator variation. Section 5.5 presents realistic benchmark results against real-world traces. Finally, conclusions and future work are discussed in Section 5.6. Appendix A presents further detailed performance measurements on RSA accumulators.

5.1 Big-O evaluation of the different PAD designs

In Chapter 4, we presented a variety of algorithms for implementing a PAD. In this section, we do a big-O comparison across the different algorithms. In Table 5.1 we compare our designs to the existing related work and present a comparison of the space usage and amortized expected running time of each algorithm in terms of the number of keys n and number of snapshots v. We assume that a snapshot is taken after every update. For tree-based PADs, query times include the $O(\log v)$ cost to binary search in the authenticator cache. For tuple-based PADs, query times include searching the persistent tree for the tuple.

A modular exponentation, used in signatures, is much more expensive than many cryptographic hashes. A standard big-O bound would not capture these effects. To enable a more accurate comparison, we account for exponentiations used in verifying signatures by using β to denote its cost. Table 5.1 then describes:

1. **Server storage (per-update).** Storage, per update, on the server.

2. **Lookup proof size.** Size of a lookup proof sent to a client.

3. **Query time (historical).** Time to make a lookup proof for old snapshots.

4. **Query time (current).** Time to make a lookup proof for the current snapshot.

5. **Verify time.** Time to verify a lookup proof by a client.

6. **Update info.** The size of an update, sent to the server.

7. **Author update time.** Time on the author required to generate an update.

8. **Server update time.** Time on the server required to process an update.

5.2 Implementation and methodology

5.2.1 Implementation

Our implementation is a hybrid of C++ and Python, connected with SWIG-generated interface wrappers. Because of the complexity of implementing all 21 different configurations, our initial implementation was in Python. Python made it much easier to design correct algorithms, debug our implementation, and cleanly modularize the code. We could then progressively and quickly port the debugged algorithms to C++, function by function and module by module while fully preserving both the original Python implementation, and the equivalence between the C++ and Python implementations, applying our Python test cases against our C++ implementation. In this paper, we present the results after porting many of the slowest modules to C++.

We ported persistent search trees first, because of their use in tree-based PADs, and their use for storing the tuple repository. We achieved a 10x-20x speed up from converting the code. We used profile-based analysis for our porting effort, only porting modules and functions that were not bottlenecked in cryptographic or existing C++ code. To guide these choices, we separately measured the time spent in signatures, serialization, and modular exponentiations.

As public-key cryptographic operations like RSA can be done with variable key lengths, trading off speed for cryptographic strength, we selected parameters at the “112-bit security

Reference	Storage Size	Query Time (historical)	Query Time (current)	Proof Size	Verify Time	Update Time (author)	Update Time (server)	Update Size
Tree PAD (Path Copy) [7]	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$\beta + O(\log n)$	$\beta + O(\log n)$	$O(\log n)$	$O(1)$
Tree PAD (Versioned Node) (No Cache)	$O(1)$	$O(n)$	$O(\log n)$	$O(\log n)$	$\beta + O(\log n)$	$\beta + O(\log n)$	$O(\log n)$	$O(1)$
Tree PAD (Versioned Node) (Cache Everywhere)	$O(\log n)$	$O(\log v \cdot \log n)$	$O(\log n)$	$O(\log n)$	$\beta + O(\log n)$	$\beta + O(\log n)$	$O(\log n)$	$O(1)$
Tree PAD (Versioned Node) (Median Cache)	$O(1)$	$O(\sqrt{n} \log v)$	$O(\log n)$	$O(\log n)$	$\beta + O(\log n)$	$\beta + O(\log n)$	$O(\log n)$	$O(1)$
Tuple PAD	$O(n)$	$O(\log n)$	$O(\log n)$	$O(1)$	$\beta + O(1)$	$O(\beta n)$	$O(n)$	$O(n)$
Tuple PAD (Speculating)	$O(C \sqrt[C]{n})$	$O(C \log n)$	$O(C \log n)$	$O(C)$	βC	$O(\beta C \cdot \sqrt[C]{n})$	$O(C \sqrt[C]{n})$	$O(C \sqrt[C]{n})$
Tuple PAD (Speculating) (+Superseding)	$O(C)$	$O(C \log n)$	$O(C \log n)$	$O(C)$	βC	$O(\beta C \cdot \sqrt[C]{n})$	$O(C \sqrt[C]{n})$	$O(C \sqrt[C]{n})$
Tuple PAD (Speculating) (+Superseding+IterHash)	$O(C)$	$O(C \log n)$	$O(C \log n)$	$O(C)$	$(\beta + D)C$	$O(C \sqrt[C]{n} (\frac{\beta}{D} + D))$	$O(C \sqrt[C]{n})$	$O(C \sqrt[C]{n})$
Accum PAD (Author precomputes witnesses)	$O(n)$	$O(\log n)$	$O(\log n)$	$O(1)$	2β	$O(\beta n)$	$O(n)$	$O(n)$
Accum PAD	$O(n)$	$O(\log n)$	$O(\log n)$	$O(1)$	2β	$O(\beta)$	$O(\beta n \log n)$	$O(1)$
Accum PAD (Speculating)	$O(C \sqrt[C]{n})$	$O(C \log n)$	$O(C \log n)$	$O(C)$	$(C + 1)\beta$	$O(\beta C)$	$O(\beta(C + 1) \sqrt[C]{n} \log n)$	$O(C)$
Accum PAD (Speculating+Superseding)	$O(C)$	$O(C \log n)$	$O(C \log n)$	$O(C)$	$2C\beta$	$O(\beta C)$	$O(\beta(C + 1) \sqrt[C]{n} \log n)$	$O(C)$

Table 5.1 : Persistent authenticated dictionaries, comparing techniques assuming a snapshot is taken after every update. Storage sizes are measured per-update. β denotes the cost of an exponentiation used during signature generation. C denotes the number of generations in a speculative PAD and D denotes the maximum hash-chain length. In this table, we report the amortized expected time or space usage. "Accum PAD" refers to tuple PADs based around accumulators. Except when stated otherwise, tuple PADs using accumulators are assumed to precompute witnesses on the server.

level" [104]. Keys and values are assumed to be 28-byte hashes and modular operations are done with a 2048-bit modulus.

All of our benchmarks were run on an Intel Core 2 Duo 2.4 GHz Linux machine with 4GB of RAM running in 64-bit mode. We used Python version 2.5.4 and compiled our C++ code with Gcc 4.3.4.

5.2.2 Serialization

For completeness, our evaluation includes the actual sizes of messages used in our PAD system. To this end, we serialized each update from the author, each request from clients, and each reply from the server. Rather than error-prone manual construction of mutually compatible serialization code in both C++ and Python, we used the Google protocol-buffer library to automatically generate interoperable serialization code. Protocol buffers support nested message types and very low space overhead. In our messages, each message field has a field header of one byte. Integers use a variable-length encoding. Blobs and encapsulated messages require a field header, length, and the binary contents.

Protocol buffers generate very fast C++ code. Unfortunately, the current Python implementation is unoptimized, often dominating the CPU time. The not-yet-released next version of protocol buffers for Python is reported to be significantly faster.

5.2.3 Tree-based PADs

There are many types of balanced tree-like data-structures from which Merkle trees can be built. We implemented treaps [91], red-black trees [93], and skiplists [79].

When implementing a PAD, the author only needs to manage one search tree, that of the latest snapshot. On the server, each snapshot is a logically distinct Merkle tree with a different signed root hash. Rather than storing each snapshot as a distinct tree,

Caching strategies	Storage (per update)	Lookup proof (time)
No cache	$O(1)$	$O(n)$
Cache everywhere	$O(\log n)$	$O(\log n)$
Median layer	$O(1)$	$O(\sqrt{n})$

Table 5.2 : Caching strategies for subtree authenticators in a Sarnak-Tarjan tree.

we can exploit the similarity between trees across snapshots to implement a more space-efficient repository on the server. The classic approach for this is *path copying*, which uses a standard applicative tree to avoid the redundant storage of subtrees that are unchanged across snapshots. Under our assumption that we take a snapshot after every update, path copying will require $O(\log n)$ storage per update and lookup proofs can be generated in logarithmic time.

For the server's repository of persistent trees, we implement path copying and the three variations of storing/recomputing subtree authenticators for Sarnak-Tarjan trees discussed in Section 4.2.6 and summarized in Table 5.2. We implemented 12 different tree-based PAD variations in Python and C++ consisting of 3 kinds of tree data structures and 4 different repository designs. We only present performance data for our C++ implementations. PADs based on two of these 12 variations were proposed by Anagnostopoulos et al. [7], red-black trees and skiplists both using path copying.

Because we are supporting different types of applicative representations, our red-black, skiplist and treap implementations are *only* allowed to "mutate" the children of a node through an abstract interface which, given a node and a pair of new left and right children, returns a node representing the result of applying those changes. The result depends on the underlying repository implementation. With path copying, it will always be a clone. With Sarnak-Tarjan trees, it may or may not be a clone. This requires that the implementations of these algorithms be *bottom-up* and *mutation-free*. In addition, because nodes store keys and values, we must preserve node identity during rotations and other operations, reusing nodes that already store the needed key and value, updating their children through our abstract interface, rather than needlessly cloning those nodes.

5.2.4 Tuple-based PADs

Tuple PADs offer a more complex parameterized set of design choices, including several optimizations described in Section 4.3. Apart from signing each tuple individually, tuple-superseding may be used alone, or in combination with lightweight signatures. Any of these three designs may be combined with speculation. In addition to this, there are the three RSA accumulator-based designs described in Section 4.3.4.

Except for lightweight signatures, our implementation is purely in Python. Despite the overheads of the Python interpreter, many of the design variations bottleneck in unavoidable cryptographic operations that already run at native speed.

5.2.5 Accumulators

We used the GMP library for all modular operations. Our accumulators use 184-bit prime representatives*. The prime representative of a tuple must be found deterministically. The SHA-1 hash of a tuple is concatenated with 24 zero bits and treated as an integer. The prime representative is chosen as the numerically smallest prime number greater than that integer, found by performing 82 Miller-Rabin [105] primality tests (as advised by NIST [104]) to confirm a candidate representative. Due to the expense of finding a prime representative,

*Implementing the 112-bit security level would properly require 248-bit prime representatives based around SHA-224. Our current crypto library limited us to SHA-1 hashes. Our results therefore underestimate the costs of RSA accumulators.

	Amazon	Google
CPU time (cents/hour)	8.5	10
Storage (cents/GB*month)	15	15
Bandwidth (cents/GB)	10–17	10-12

Table 5.3 : Costs charged by Amazon EC2 and Google AppEngine for cloud-computing and storage.

the author sends the offset to the prime representative along as a hint. In our implementation, we perform all witness computation on the server.

5.2.6 Cloud provider economics

In Table 5.3 we present the current costs of two cloud providers: Amazon EC2 and Google AppEngine. While the absolute prices we measure may vary in the future, what matters in our evaluation is the relative prices between storage, bandwidth, and CPU cycles. We observe that both providers charge very similar prices. In our analysis, we will assume that the relative costs in this table indicate tradeoffs that apply with any large service provider; we will also assume that the author is spending the money, and will attempt to minimize the total costs for the author and server. For simplicity in our evaluation of algorithms we will assume that cloud providers charge by CPU time, while the task is executing. Or, if a cloud provider charges by wall-clock time, the CPU utilization is 100%.

We observe that transmitting an extra kilobyte of data costs just as much as computing for $1/200^{\text{th}}$ of a second. This defines the *provider equilibrium rate*, measured in KB/sec. An algorithm need not be perfectly balanced to be optimal, of course, but this demonstrates that an optimal algorithm may well trade-off somewhat more communication for a greater savings in computation or vice versa.

5.2.7 Methodology

Our analysis has too many algorithms for us to directly compare. We reduce the complexity of our evaluation by first performing microbenchmarks to determine optimal parameters for each algorithm. We then make comparisons across algorithms with longer traces.

In our *growing microbenchmark*, we evaluate the performance when inserting one key in each snapshot, then performing random queries against each snapshot. In Section 5.5, we present our results of running a macro-benchmark of the different PAD algorithms' performance when used to store a constantly changing set of values taken from a trace of e-commerce prices.

For each benchmark we evaluate its raw performance on the author, publisher and client. We then evaluate the algorithms' effectiveness in the context of a cloud-computing environment, based on the charges made by Amazon and Google for their online services. For each algorithm, we can evaluate the relative contribution of bandwidth or CPU time to the monetary costs of an update or a lookup. We define the *update bandwidth ratio* as the result of the dividing the update size (in kilobytes) by the time to perform an update, in seconds[†]. We define the *lookup bandwidth ratio* similarly. Both are measured in kilobytes per second. For updates, we include time spent on the author and server. For lookup proofs, we only count costs on the server.

We can compare the bandwidth ratio of an algorithm to the provider equilibrium rate to determine whether bandwidth or CPU time is responsible for the majority of the monetary costs of an algorithm. When the bandwidth ratio of an algorithm exceeds the provider equilibrium rate, the bandwidth is responsible for the majority of the costs.

[†]Equivalently, we could multiply the size of a message by the rate (in messages/sec) at which the algorithm generates updates.

While bandwidth ratios for updates and lookups are a useful mechanism for comparing the relative contribution of bandwidth or CPU time to the costs of an update or lookup, the absolute costs, both per update, and totaled over all updates are also important. Some of the algorithms we present differ in cost by less than a millionth of a dollar per lookup. Optimizing algorithms to this degree is only important when there are billions or trillions of lookups. In addition, we assume that costs on the author and server are equal and the goal is to minimize the total monetary cost of the implementation. For systems under other constraints, or built under a different pricing structure, the analysis would be different.

This evaluation methodology also measures the update costs, verification costs, and proof sizes of dynamic authenticated dictionaries based on these designs. Recall that the only difference between a PAD and DAD is that the server for a DAD will purge data from older versions[‡].

5.3 Tree PAD microbenchmarks

We first consider the relative performance of treaps, red-black trees, and skiplists against microbenchmark loads. We also consider how efficiently these tree-like structures reuse state across versions, comparing path copying and three Sarnak-Tarjan variations.

5.3.1 Comparing tree structures

Our first evaluation considers which type of tree-like data structure runs fastest. We performed a growing microbenchmark with 100,000 keys. In general, all three tree algorithms performed similarly with 730-750 inserts per second, and 480-600 lookup proof verifications per second. All three tree algorithms spent 80%-90% of their time computing cryp-

[‡]It might be tempting to remove version numbers entirely, particularly the version number ranges from tuple PADs. This could enable version rollback attacks, so we leave this information in the DAD.

	Proof size (kB)	Lookup rate (keys/sec)	RAM used (MB)
Treap	1.98	7649	1079
Red-black	1.53	7756	843
Skiplist	2.67	4346	1587

Table 5.4 : Performance across different tree types, inserting 100k keys, and using path-copying to implement the repository.

	Queries (per sec)	RAM used (MB)
Path Copying	7756.	843
Cache Nowhere	1.5	182
Cache Everywhere	7423.	358
Cache Median	196.	205

Table 5.5 : Memory usage and lookup proof performance across different persistency approaches storing red-black trees containing 100k keys.

tographic signatures, implying that additional performance tuning on our part would yield limited gains. All three algorithms had an update size of 150 bytes.

There are differences between the algorithms that can be seen in Table 5.4. Red-black generates the shallowest trees, causing it to have the smallest lookup proofs, the fastest performance, and the least RAM usage. Although red-black trees make the most efficient trees to use for authenticated dictionaries, they are the most complex; their implementation requires 38 distinct rules[§]. Treaps and our skiplists are much simpler, requiring 13 rules. In addition they are *history independent* [88, 89], meaning that the root hash does not depend on the insertion order. For some uses of a PAD, history independence may be desirable.

[§]The authors wish to thank Stefan Kahrs at the University of Kent for making a Haskell implementation of red-black trees that correctly handles deletion available on the Internet. We ported his code to Python and then C++.

5.3.2 Comparing tree PAD repositories

Our second evaluation of tree PADs considers the different strategies for representing the repository for their efficiency at storing the forest of trees that represents the individual snapshots. In our implementation, each Sarnak-Tarjan node always caches the subtree authenticator for the latest snapshot, and lookup proof generation performance on that snapshot is between 4,300-7,600 proofs per second, depending on the tree used, as discussed in Section 5.3.1.

In Table 5.5 we present the RAM usage and the lookup rate for the four type of repositories when querying for historical snapshots. As expected, the Sarnak-Tarjan trees use much less memory than path copying trees and the different caching strategies follow the asymptotic memory usage and performance that we would have expected (see Table 5.2). Even though Sarnak-Tarjan trees that cache everywhere have the same logarithmic space and CPU costs as path copying trees, they use less memory because adding to the authenticator cache is much cheaper than cloning nodes.

To better understand the scaling behavior of tree PADs, we ran a *steady-state microbenchmark*. We fill the PAD to some capacity, and then add one key and remove one key in each snapshot. Figure 5.1 show how the performance of a red-black tree varies for different keycounts in the dictionary with all four of our tree repository strategies. As expected, the penalty for cache-nowhere and cache-median layer increases as the dictionary gets more keys, with cache-median degrading more slowly.

5.3.3 Tree PADs in a cloud-computing environment

In this section we will evaluate the tradeoffs between the two tree versioning strategies in the last section with the best time/space tradeoffs, cache everywhere and cache median, in a cloud computing environment. We will evaluate red-black trees containing 10k and 100k

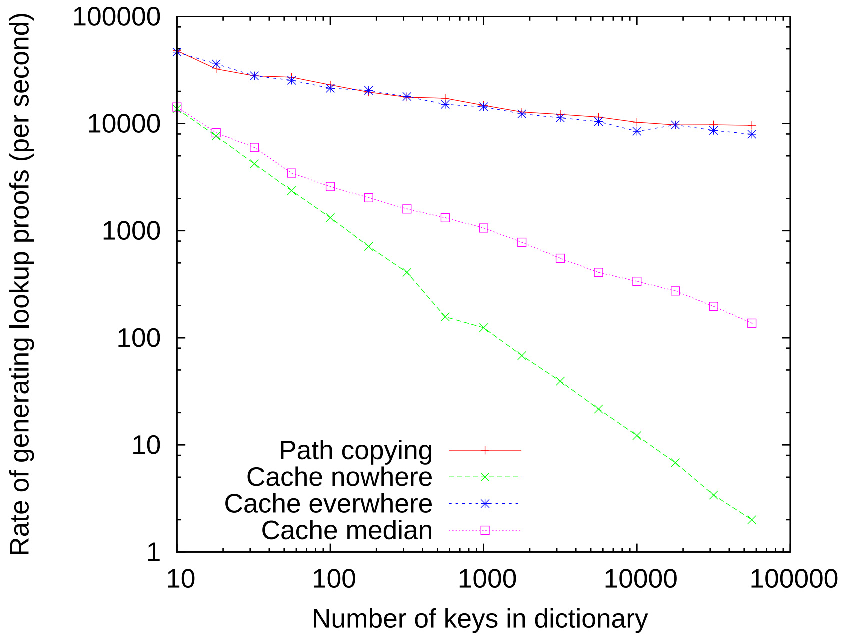

Figure 5.1 : Steady-state lookup proof generation performance for red-black trees.

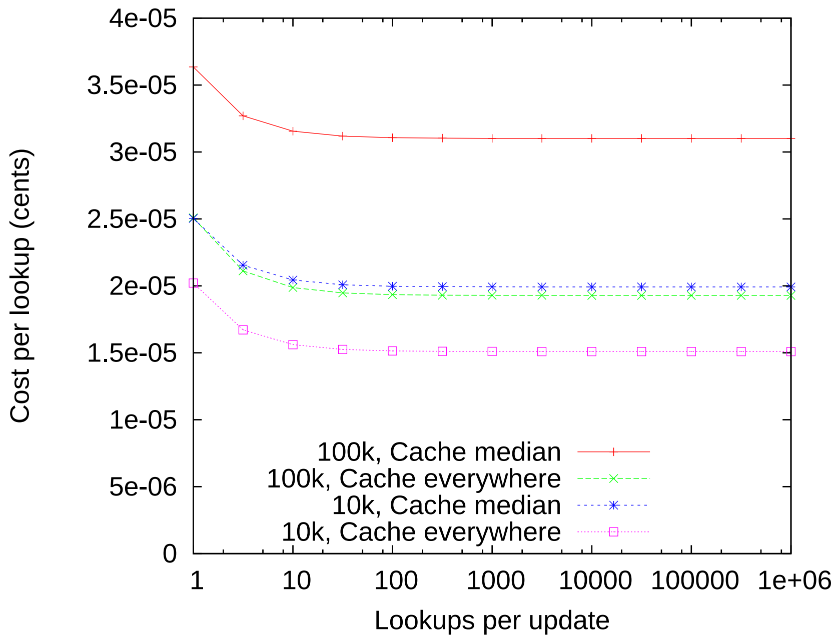

Figure 5.2 : Amortized cost per lookup for red-black tree PADs with two different hash caching strategies.

	Bandwidth Ratio	
	Updates	Lookups
Cache everywhere, 10k keys	109	13190
Cache median, 10k keys	109	562
Cache everywhere, 100k keys	90	11357
Cache median, 100k keys	102	300

Table 5.6 : Bandwidth ratios for each red-black tree PAD algorithms summarizing the relative monetary costs of bandwidth and CPU time. For ratios over the provider equilibrium ratio (200kB/sec), proof size dominates the monetary costs. For smaller ratios, computation time dominates.

keys.

In Table 5.6 we present our results. Surprisingly, even though cache median has lookups almost 40 times *slower* than cache everywhere, both algorithms are fast enough that the bandwidth of the reply message is the majority of the monetary cost of deployment.

The average per-lookup monetary cost of a PAD algorithm can vary depending on the ratio between lookups and updates. In Figure 5.2 we plot the costs per update across different lookup to update ratios for the different configurations of red-black tree PADs. Cache median is only 50% more expensive than cache everywhere, but required 40% less memory usage.

5.3.4 Summary of results

We conclude that tree-based PADs should use Sarnak-Tarjan nodes with the cache-everywhere versioning strategy. In the case where very few queries are made for historical snapshots or where available memory is low, caching on the median layer may have sufficient query throughput. We also conclude that red-black trees dominate treaps and skiplists, running faster, having smaller lookup-proof sizes, and using less storage. Treaps enable other useful semantics which we have not discussed in this paper (see Crosby and Wallach [8] for details), but there is no reason to ever use a skiplist.

Base+SS	No speculation. Optimized with superseding.
Base+LW	No speculation. Optimized with lightweight signatures.
Spec+SS	Speculation with 2 generations. Optimized with superseding.
Spec+LW	Speculation with 2 generations. Optimized with lightweight signatures.
Accumulators	Speculation with 2 generations. Uses accumulators.
Chain Accum.	Speculation with 2 generations. Uses accumulators in a hash chain.

Table 5.7 : Abbreviations used to denote the different tuple-based algorithms.

5.4 Tuple PAD microbenchmarks

In this section, we will evaluate the various tuple PAD designs described in Section 4.3. Table 5.7 describes the abbreviations we will use for the different algorithms. We only present results with tuple superseding; not using superseding has a trivial impact on CPU time, update sizes, and lookup proof sizes. Superseding only saves storage on the server. For comparison, we also report results for red-black trees using the cache-everywhere strategy. Because of the slower performance of tuple PADs, we only benchmarked 10,000 keys.

5.4.1 Tuple PAD author costs

In Table 5.8, we present the performance of each tuple PAD algorithm we analyzed. We also present red-black results for comparison. Note that due to poor insert performance, we only ran Base+Supersede for 2915 inserts, instead of 10,000. If we extrapolated its performance at 10,000 inserts, we would expect .10 updates per second and a 250kB update size.

Table 5.8 demonstrates several of the tradeoffs between the PAD algorithms. It shows the benefits of speculation increasing performance by a factor of 30 and reducing update sizes by a factor of 50. Lightweight signatures have a similarly strong impact on performance. We also observe that lightweight signatures are sufficiently cheap that crypto costs are no longer the dominant limiting factor in PAD insertion performance. Since much of the remaining code, in these cases, is written in Python, we expect that significant speedups could still be available from performance tuning. Lightweight signatures have a small negative performance interaction with speculation. Whenever the length of an epoch changes, every tuple must be re-signed with a public key signature.

This table also shows the poor update performance of tuple PAD algorithms. Even if we assume non-crypto overheads on the author are zero, the fastest tuple PAD is still four times slower than a simple red-black tree PAD. The network communication needed for updating the red-black tree PAD is similarly as small as the very best accumulator-enhanced tuple PAD.

We implemented the hash chain optimiziation described in Section 4.3.4 and observe the essentially identical CPU performance as signing each generation's accumulator individually because because primality computation and exponentiation operations dominate, compared to computing $1 + \frac{1}{\sqrt{n}}$ times as many signatures. Unexpectedly, accumulators, although being "constant size," are surprisingly large and generate lookup proofs no smaller than red-black trees storing 10k keys.

Accumulator tuple PAD costs In Table 5.9 we break down the unavoidable cryptographic costs of accumulator operations on the author and server. These results show that 99% of CPU time spent on the publisher handling accumulator updates is in the underlying costs of the accumulator. In addition, of the 67% of the CPU time spent on cryptography by the author, 91% is spent on underlying costs of the accumulator. We discuss accumulator overheads more extensively in Appendix A.

	Inserts		Size (kB)		Number
	(per sec)	% in crypto	Update	Proof	Inserted
Base+SS	.35	61%	86.95	.15	2915
Base+LW	.94	5%	156.72	.21	10000
Spec+SS	4.5	59%	6.42	.30	10000
Spec+LW	30.	24%	3.76	.42	10000
Chain Accum.	54.7	67%	.14	1.23	10000
Accumulators	53.9	67%	.14	1.29	10000
Red-black	776.	92%	.15	1.24	10000

Table 5.8 : Comparing author performance, update sizes and proof sizes across different PAD designs. Crypto costs include digital signatures, finding prime representatives, lightweight signatures, and exponentiations. Except for "Base+Supersede," where 2915 keys were inserted, we ran each algorithm with 10,000 keys.

Author	
Update RSA accumulator	13%
Find prime representative	48%
Digital signature	6%

Server receiving update	
Compute witnesses	91%
Find accumulator value	8%

Server generating reply	
–	

Client verifying proof	
Verify signature	44%
Verify accumulator	34%

Table 5.9 : Breakdown of accumulator update and witness computation and verification costs for tuple PADs using non-hash-chain accumulators with 10,000 keys.

	Updates		Server response	Client response verification	
	(per sec)	% in crypto	generation (per sec)	(per sec)	% in crypto
Base+SS	1.7	—	1182	486	68
Base+LW	1.03	—	1088	441	65
Spec+SS	20.8	—	620	240	67
Spec+LW	45.4	—	571	227	64
Chain Accum.	.92	99%	522	201	62
Accumulators	.92	99%	480	157	70
Red-black	10869.	—	10992	642	90

Table 5.10 : Comparing server and client performance across different PAD designs. Cryptographic costs include digital signatures, finding prime representatives, lightweight signatures, and exponentiations.

5.4.2 Tuple PAD server costs

In Table 5.10, we present the server's costs for the different PAD algorithms. On each update, most algorithms do nothing other than store tuples and signatures into the repository, taking time proportional to the update size. Accumulator algorithms, however, also have to compute witnesses for each tuple and are extremely slow. (See Appendix A for performance details on this.) In this table we can see the extreme benefits of speculation, which improves performance on the server by reducing the number of tuples the server must process for each snapshot from $O(n)$ to $O(\sqrt{n})$.

Except for witness computation overheads in tuple PADs using accumulators, servers do not perform any cryptography when handling an update. Servers also never perform cryptography when generating replies to clients. From our experience with optimizing tree PADs in C++, this non-crypto code is significantly slowed down due to Python overheads. We expect that the non-crypto server performance would improve by a factor of 10-50 if the tuple PAD code was rewritten in C++.

The time for a client to verify a lookup proof varies across the different algorithms. Except for tuple PADs using accumulators, the cost of verifying is dominated by signature verification. Designs using speculation usually require verifying two signatures, one in each generation, and thus take twice as long.

Accumulator PADs using hash chains do not have an appreciably smaller lookup proof. The size of a lookup proof is dominated by the 2048 bit accumulator value and the 2048-bit witness, required for each generation. These overheads are large compared to the 240-bit cost of an extra signature. Hash chains somewhat improve lookup proof verification performance. When a hash chain is used, only one signature need be checked. This can be seen in Table 5.10 in the increased performance verifying a hash chain accumulator lookup proof.

	Bandwidth Ratio	
	Updates	Lookups
Base+SS	25.	177
Base+LW	77.	228
Spec+SS	24.	186
Spec+LW	68.	240
Chain Accum.	.126	624
Accumulators	.126	619
Red-black	109.	13630

Table 5.11 : Bandwidth ratios for each PAD algorithm summarizing the relative monetary costs of bandwidth and CPU time. For ratios over the provider equilibrium ratio (200kB/sec), proof size dominates the monetary costs. For smaller ratios, computation time dominates.

5.4.3 Tuple PADs in a cloud-computing environment

In this section, we will evaluate the tradeoffs between the various PAD designs in the context of a cloud-computing environment and perform the analysis described in Section 5.2.7. In Table 5.11 we present the bandwidth ratio for each algorithm. Whenever the ratio exceeds 200 kB/sec, the monetary cost of transmitting the message exceeds the monetary cost of computing the message. Every implementation has a bandwidth ratio over 177 for lookups, meaning that at least 45% of the monetary costs of these algorithms will come from bandwidth of the reply, not the CPU time, despite slow Python implementations.

The overall monetary cost of each algorithm depends on the relative ratio between updates and lookups. In Figure 5.3 we plot the costs per lookup across different lookup to update ratios for several algorithms. This plot concisely illustrates the tradeoffs between the different algorithms. Except for the algorithms using accumulators, *each* other algorithm is the cheapest at some ratio of lookups to updates. This plot also demonstrates that accumulators are more expensive than red-black trees at all ratios when the PAD contains 10k keys.

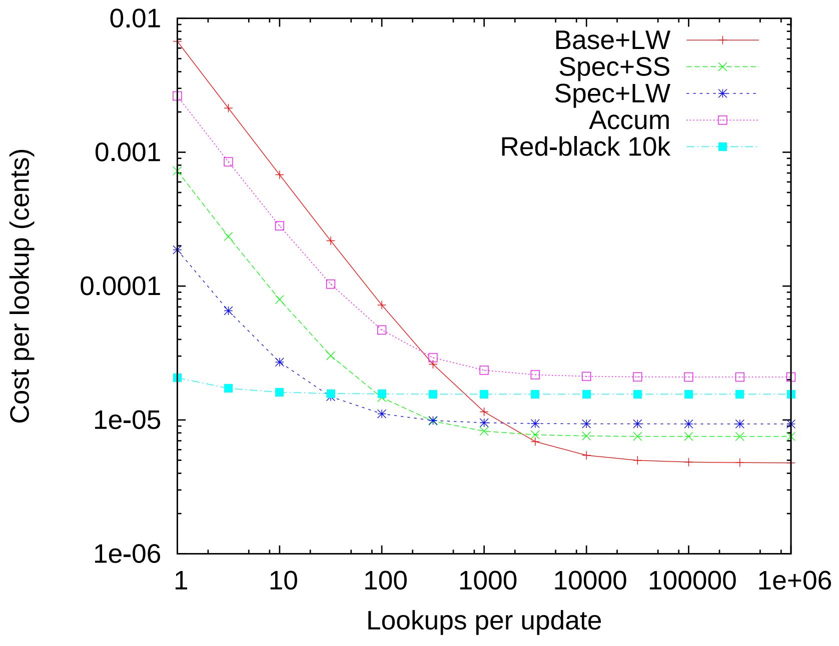

Figure 5.3 : Amortized cost per lookup for different PAD algorithms.

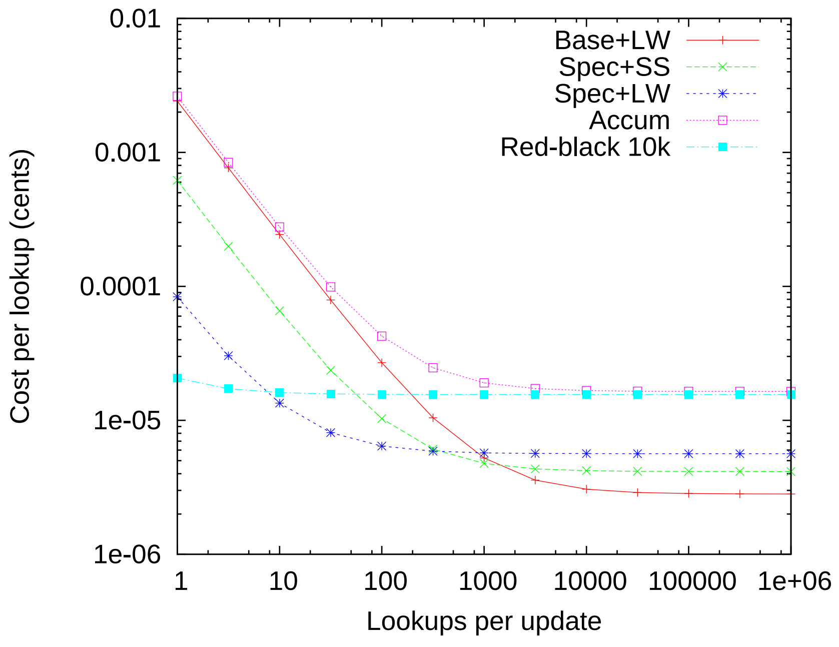

Figure 5.4 : Amortized cost per lookup for different PAD algorithms, correcting for Python overheads.

Correcting for Python overheads In this analysis, so far, we used our measured CPU performance, despite the tuple PAD algorithms not having an optimized C++ implementation. We correct for these performance anomalies by assuming that a C++ implementation of Spec+LW will have an update performance on the author three times faster and that Base+LW will also have its performance increase by a factor of 10. We will also assume that C++ versions of the non-accumulator tuple PAD algorithms can process updates ten times faster.

Finally, we will assume that the tuple repository can return the tuple matching a lookup query in .1ms. When the server responds to a lookup request, it only needs to find the matching tuple in the repository, implemented using a persistent red-black tree, and serialize it. In Table 5.10, we benchmarked our C++ red-black tree PAD at generating over 10,000 lookups proofs per second, with each proof requiring serializing the entire search path to the lookup key, clearly a more expensive operation.For tuple PADs that use speculation, multiple tuples are required in a proof and we multiply the lookup time by the number of generations.

Under these assumptions, Figure 5.4 presents the cost per lookup for the different PAD algorithms. The cost per lookup across the algorithms does not change much. From the bandwidth ratios reported in Table 5.11, bandwidth costs were already responsible for 50%-75% of the monetary costs of most of the PAD algorithms and reducing the CPU consumption has a small effect on the total cost. The bandwidth ratios in Table 5.11 do show that 66%-99.9% of the monetary costs of updates for are from CPU consumption, which are affected by our assumed performance increases of a C++ implementation.

5.4.4 Summary of results

Whether we use the corrected or uncorrected performance numbers, we can reach several conclusions. RSA accumulators are so expensive, from a CPU and bandwidth perspective, that we will never recover these costs for any realistic problem set. For PADs which are updated very frequently, red-black tree PADs clearly win. However, for more stable PADs with higher query rates, the tuple-based PAD structures, sans the RSA accumulator, become the preferable strategy. For workloads where a widely varying range of lookups per update might be expected, the full set of optimizations, including speculation, lightweight signatures, and superseding, seems to be an excellent strategy. For workloads where over 1000 lookups might be expected per update, the non-speculative tuple PAD, but with lightweight signatures, would seem to be the appropriate algorithm.

5.5 Macrobenchmark

Now that we have done many microbenchmarks of the different PAD designs, we now analzye the performance and monetary costs of the different PAD algorithms when used to store a constantly changing set of values taken from a trace of e-commerce prices.

Our data set represents the selling prices of different products for three brands of high-end luxury goods as offered by a number of vendors on the Internet. All price observations were made between January 1, 2009 and June 30, 2009 inclusive, representing 27 distinct dates. In total, 1,272 different luxury items were found online for the three brands, on a total of 544 different web sites. In total, there are 38,391 different observations in the data set. Our data tracked the price of each good on each web site, forming 14,374 distinct keys in the PAD.[¶]

[¶]Data provided by Glenn Kramer Consulting, LLC, representing actual brands and products monitored

	Insert All Keys (sec)	Process All Updates (sec)	Size (kb) Update	Size (kb) Proof	Lookups (per sec)
Base+SS	565.	154.	711	.18	1067
Base+LW	281.	135.	550	.24	710
Spec+LW	215.	101.	460	.42	614
Red-black	1.75	1.48	149	1.59	10012

Table 5.12 : Performance of different PAD algorithms on the macrobenchmark, including the total time on the author and server to insert six months of price data, the average size of an update and lookup proof, and the lookup rate.

Table 5.12 presents the performance of the different algorithms on this benchmark. This dataset is very different than our microbenchmarks. It has a course granularity. 38k updates are contained in only 27 snapshots leading to large update messages. Lookup performance is as fast as we saw in our earlier microbenchmarks.

This dataset also demonstrates that the strengths of speculation occur when there are many snapshots and relatively few keys are modified in any one snapshot. In this dataset, with the default epoch size, speculation only reduces the number of signatures needed by 7%. However, when we reduced the epoch-size to 6, the ideal epoch size for this data set, speculation reduced the needed signatures by 48%. We present results with an epoch size of 6. Lightweight signatures were also very beneficial, reducing the number of public-key signatures by over 80%.

We also performed a cloud-computing analysis of the monetary costs of different PAD algorithms over this data set. Bandwidth ratios are reported in Table 5.13 and the bandwidth ratios for lookup messages are within 20% of what we observed earlier in Tables 5.6 and 5.11 when running the growing benchmark. The update message bandwidth ratio for red-black trees is much larger than we saw in Table 5.6 because the message size has grown to include all of the updated keys, while the number of CPU-time-expensive

for an anonymous client, blinded and provided with client's permission.

	Bandwidth Ratio	
	Updates	Lookups
Base+SS	27	192
Base+LW	36	170
Spec+LW	45	261
Red-black	1244	15919

Table 5.13 : Bandwidth ratios for each algorithm, processing the luxury-goods macrobenchmark, summarizing the relative monetary costs of bandwidth and CPU time. For ratios over the provider equilibrium ratio (200kB/sec), proof size dominates the monetary costs. For smaller ratios, computation time dominates.

digital signatures remains at one per snapshot.

In Figure 5.5 we plot the cost per lookup. In this dataset, the large number of changes per snapshot results in large per-update monetary costs which must be amortized over many messages before the smaller response sizes of tuple PADs reduces the overall costs.

From this, we can conclude that the red-black tree PAD (Sarnak-Tarjan, cache-everywhere) is the preferred PAD algorithm until the query rate exceeds roughly 5000 lookups per update. Only then do the tuple PAD structures become more cost effective, with the simpler "Base+SS" strategy (no speculation or lightweight signatures; just superseding) ultimately winning only when the query rate exceeds 25k lookups/update.

5.6 Summary of PAD performance results

Our analysis considered two very different structures for implementing persistent authenticated dictionaries: those based on Merkle tree-like data structures and those based on independently signed "tuples." We implemented Merkle trees based on skiplists, treaps, and red-black trees, along with four different strategies for how to share related state across different versions of the trees. We implemented several tuple-based PAD designs, both using accumulators and including a variety of optimizations.

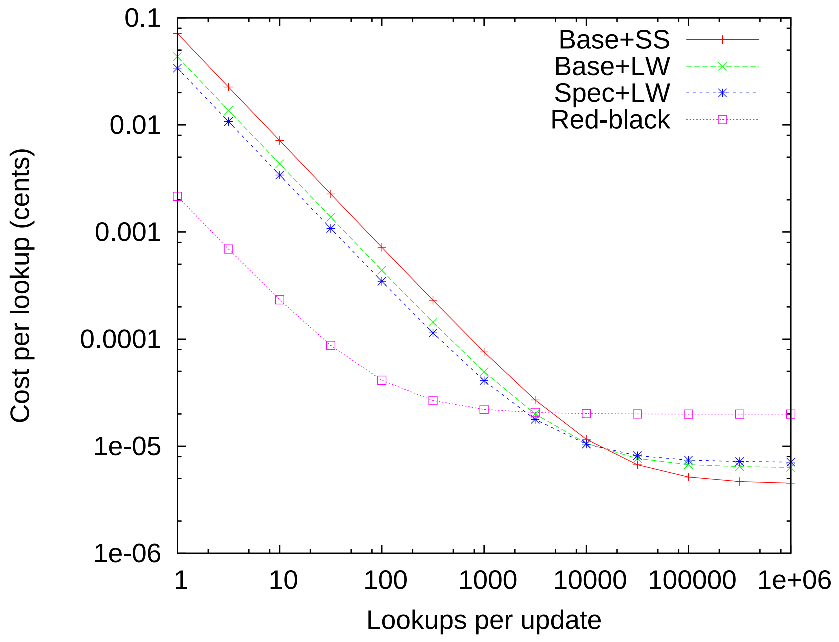

Figure 5.5 : Amortized cost per lookup for different PAD algorithms processing the luxury-goods macrobenchmark.

These algorithms make a variety of different tradeoffs between computation, bandwidth, and storage. Our strategy of converting all of these units into monetary currency, based on commodity pricing from Amazon and Google, offered us a very useful insight into which algorithms are preferable under which conditions. Most notably, we conclude that the fixed costs of RSA accumulators dwarf their asymptotic benefits, making them unsuitable for production use. We conclude that red-black trees, implemented with Sarnak and Tarjan's versioning strategy, and caching subtree authenticators at every node for every version, is the optimal strategy for PADs experiencing frequent updates. However, when the query rate grows much larger than the update rate, our tuple PAD strategies, with the full suite of optimizations, seem to be the preferable strategy.

Chapter 6

Conclusions and future work

In this thesis I have presented two classes of tamper-evident data structures, tamper-evident logs and persistent authenticated dictionaries. These data structures are designed to run on untrusted servers. As an untrusted machine can nominally do anything, tamper-evidence is the strongest guarantee that can be offered. To this end, we have presented the history tree, several persistent authenticated dictionary algorithms, and an evaluation of these designs including both big-O and the measured performance of working implementations.

The essence of our history tree is its ability to detect unauthorized changes between different versions, without sending intermediate events. I would like to extend this property to data structures more complex than an append-only log.

There are several avenues of future work in persistent authenticated dictionaries. There are a number of properties I would like to formally prove, including big-O bounds on the storage costs and tighter bounds on lookup time, as well as formally proving that my PAD designs always detect failure or return the correct answer for various threat models. Future work includes expanding our trust model for PADs to better support multiple, mutually-untrusting authors, offer stronger privacy features, as well as extending our tuple PAD designs to support out-sourced storage where a trusted device uses a small amount of trusted storage to detect faults in a larger untrusted storage [106, 107].

My algorithms and my evaluation generalize to the case of a dynamic authenticated dictionary, when persistence is unnecessary, but tamper-evidence is, by simplifying the techniques to only preserve the data necessary to authenticate the latest snapshot. I plan to adapt speculation and lightweight signatures to create a dynamic super-efficient authenticated dictionary.

Given my flexible software implementation of so many different PAD variations, I intend to pursue applications of my data structures toward more concrete problems, such as building robust file storage above potentially untrusted storage like Amazon's S3 service. I also intend to release my code under a suitable open-source license.

6.1 Contributions

My research makes the following contributions:

- Recognizing that auditing is a critical and frequent operation in designing tamper-evident data structures.
- Designing, implementing, and evaluating the history tree tamper-evident log.
- Merkle aggregation, a generic technique of aggregating annotations up a Merkle tree.
- Improvements on existing tree-based persistent authenticated dictionaries and presenting tuple PADs, a new paradigm for PADs, offering constant sized lookup results.
- A new way of evaluating algorithms that use network bandwidth and CPU time in terms of their monetary costs by using the prices charged by cloud-computing providers.
- An implementation and evaluation of all current PAD algorithms that generalizes to non-persistent authenticated dictionaries.

Tamper-evident data structures are widely applicable. They can detect malicious insiders and increase the trust in software services and "cloud computing". Along with presenting and evaluating specific designs for new and improved designs for tamper-evident logs and dictionaries, this thesis also presents design principals for designing new tamper-evident data structures along with several optimizations usable by tamper-evident systems.

Appendix A

Accumulators in practice

Our research showed that RSA accumulators, when applied to tuple PADs, introduced significant overheads both in terms of CPU and bandwidth costs, leading us to conclude that RSA accumulators, despite their asymptotic benefits, were unsuitable for production use in PADs.

In this appendix, we take a closer look at RSA accumulators as a stand-alone entity, and their costs on the author, server, and clients. This performance evaluation assumes the "112 bit security level," requiring 224-bit hashes, 240-bit prime representatives, and 2048-bit modulus operations.

In Figure A.1, we graph the cost of updating an accumulator as a function of the number of keys in the accumulator. For each accumulator set size, we estimate the runtime by combining microbenchmarks of the costs of primality tests and exponentiations of different sizes, and by counting the exact number of exponentiations and primality tests required to update that accumulator.

Generating a prime representative takes 7.68ms if 120 Miller-Rabin primality tests are performed. 120 primality tests are needed to attain a mathematical security factor of 2^{-120}. A less conservative design could test a tentative prime representative with 5 Miller-Rabin tests, taking 1.50ms. The author, knowing the factorization of n can incrementally update precomputed witnesses for the next snapshot at a cost of $O(n)$ 2048-bit exponentiations per snapshot, each costing 2.42 ms by using the Chinese remainder theorem. Alternatively the publisher may batch compute all witnesses with $O(n \log n)$ 240-bit exponentiations, each

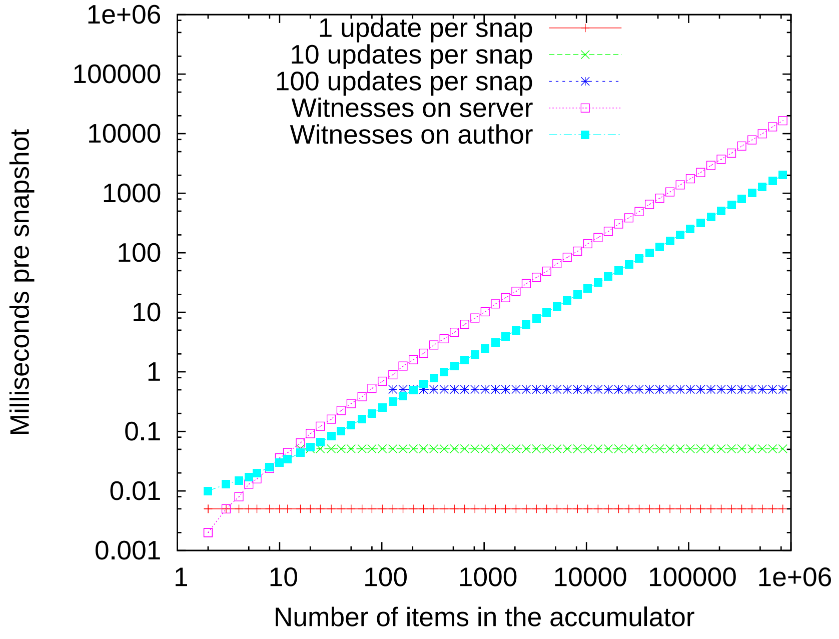

Figure A.1 : Calculated CPU time per-update for accumulators.

requiring .99ms. Verifying an accumulator requires one 2048-bit exponentiation, costing 8.59ms.

A membership proof in an RSA accumulator requires 2048 bits to send the accumulator value and 2048 bits to send the witness. In addition, the signature over the accumulator and the item in the accumulator must also be included.

The advantage of RSA accumulators is in saving the bandwidth required for an update. If witnesses are computed on the author and sent, no bandwidth is saved unless witnesses are smaller than signatures. If witnesses are computed on the server, then an accumulator only makes sense when the cost of the time to compute witnesses is cheaper than the cost of the time required to sign each item, as in the tuple PAD designs, and the cost of the bandwidth to send the signatures to the server. With Amazon and Google's prices for bandwidth and computation, it is far cheaper to simply use a tuple PAD and avoid accumulators.

Future accumulator designs may solve these problems. Alternative accumulator designs have been proposed around elliptic curve cryptography, potentially offering smaller accumulator sizes. However, the designs we examined require a fixed bound on the number of keys in the accumulator [108] or have quadratic overhead for computing witnesses [109].

Bibliography

[1] D. Dolev and A. C. Yao, "On the security of public key protocols," *Annual IEEE Symposium on Foundations of Computer Science*, vol. 0, pp. 350–357, 1981.

[2] B. Cohen, "Incentives build robustness in BitTorrent," tech. rep., bittorrent.org, 2003.

[3] P. A. Gerr, B. Babineau, and P. C. Gordon, "Compliance: The effect on information management and the storage industry." The Enterprise Storage Group, May 2003. `http://searchstorage.techtarget.com/tip/0,289483,sid5_gci906152,00.html`.

[4] R. Sion, "Strong WORM," in *International Conference on Distributed Computing Systems*, (Beijing, China), pp. 69–76, May 2008.

[5] M. Naor and K. Nissim, "Certificate revocation and certificate update," in *USENIX Security Symposium*, (San Antonio, TX), Jan. 1998.

[6] P. C. Kocher, "On certificate revocation and validation," in *International Conference on Financial Cryptography (FC '98)*, (Anguilla, British West Indies), pp. 172–177, Feb. 1998.

[7] A. Anagnostopoulos, M. T. Goodrich, and R. Tamassia, "Persistent authenticated dictionaries and their applications," in *International Conference on Information Security (ISC)*, (Seoul, Korea), pp. 379–393, Dec. 2001.

[8] S. A. Crosby and D. S. Wallach, “Super-efficient aggregating history-independent persistent authenticated dictionaries,” in *Proceedings of ESORICS 2009*, (Saint Malo, France), pp. 671–688, Sept. 2009.

[9] J. S. Shapiro and J. Vanderburgh, “Access and integrity control in a public-access, high-assurance configuration management system,” in *USENIX Security Symposium*, (San Francisco, CA), pp. 109–120, Aug. 2002.

[10] N. Sarnak and R. E. Tarjan, “Planar point location using persistent search trees,” *Communications of the ACM*, vol. 29, no. 7, pp. 669–679, 1986.

[11] R. C. Merkle, “A digital signature based on a conventional encryption function,” in *CRYPTO ’88*, pp. 369–378, 1988.

[12] M. Goodrich, R. Tamassia, and A. Schwerin, “Implementation of an authenticated dictionary with skip lists and commutative hashing,” in *DARPA Information Survivability Conference & Exposition II (DISCEX II)*, (Anaheim, CA), pp. 68–82, June 2001.

[13] B. Gassend, G. Suh, D. Clarke, M. Dijk, and S. Devadas, “Caches and hash trees for efficient memory integrity verification,” in *The 9th International Symposium on High Performance Computer Architecture (HPCA)*, (Anaheim, CA), Feb. 2003.

[14] P. Williams, R. Sion, and D. Shasha, “The blind stone tablet: Outsourcing durability,” in *Sixteenth Annual Network and Distributed Systems Security Symposium (NDSS)*, (San Diego, CA), Feb. 2009.

[15] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen, “Ivy: A read/write peer-to-peer file system,” in *USENIX Symposium on Operating Systems Design and Implementation (OSDI ’02)*, (Boston, MA), Dec. 2002.

[16] J. Li, M. Krohn, D. Mazières, and D. Shasha, "Secure untrusted data repository (SUNDR)," in *Operating Systems Design & Implementation (OSDI)*, (San Francisco, CA), Dec. 2004.

[17] Z. N. J. Peterson, R. Burns, G. Ateniese, and S. Bono, "Design and implementation of verifiable audit trails for a versioning file system," in *USENIX Conference on File and Storage Technologies*, (San Jose, CA), Feb. 2007.

[18] K. Fu, M. F. Kaashoek, and D. Mazières, "Fast and secure distributed read-only file system," *ACM Transactions on Compututer Systems*, vol. 20, no. 1, pp. 1–24, 2002.

[19] M. T. Goodrich, R. Tamassia, N. Triandopoulos, and R. F. Cohen, "Authenticated data structures for graph and geometric searching," in *Topics in Cryptology, The Cryptographers' Track at the RSA Conference (CT-RSA)*, (San Francisco, CA), pp. 295–313, Apr. 2003.

[20] P. Devanbu, M. Gertz, A. Kwong, C. Martel, G. Nuckolls, and S. G. Stubblebine, "Flexible authentication of XML documents," *Journal of Computer Security*, vol. 12, no. 6, pp. 841–864, 2004.

[21] S. A. Crosby and D. S. Wallach, "Efficient data structures for tamper-evident logging," in *Proceedings of the 18th USENIX Security Symposium*, (Montreal, Canada), Aug. 2009.

[22] D. Davis, F. Monrose, and M. K. Reiter, "Time-scoped searching of encrypted audit logs," in *Information and Communications Security Conference*, (Malaga, Spain), pp. 532–545, Oct. 2004.

[23] B. Schneier and J. Kelsey, "Secure audit logs to support computer forensics," *ACM Transactions on Information and System Security*, vol. 1, no. 3, 1999.

[24] R. Ostrovsky, A. Sahai, and B. Waters, “Attribute-based encryption with non-monotonic access structures,” in *ACM Conference on Computer and Communications Security (CCS ’07)*, (Alexandria, VA), pp. 195–203, Oct. 2007.

[25] V. Goyal, O. Pandey, A. Sahai, and B. Waters, “Attribute-based encryption for fine-grained access control of encrypted data,” in *ACM Conference on Computer and Communications Security (CCS ’06)*, (Alexandria, VA), pp. 89–98, Oct. 2006.

[26] A. Sahai and B. Waters, “Fuzzy identity based encryption,” in *Workshop on the Theory and Application of Cryptographic Techniques on Advances in Cryptology (EuroCrypt ’05)*, vol. 3494, pp. 457 – 473, May 2005.

[27] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan, “Private information retrieval,” in *Annual Symposium on Foundations of Computer Science*, (Milwaukee, WI), pp. 41–50, Oct. 1995.

[28] P. Williams and R. Sion, “Usable PIR,” in *Network and Distributed System Security Symposium (NDSS)*, (San Diego, CA), The Internet Society, Feb. 2008.

[29] I. Goldberg, “Improving the robustness of private information retrieval,” in *IEEE Symposium on Security and Privacy*, (Oakland, CA), May 2007.

[30] B. Chor and N. Gilboa, “Computationally private information retrieval,” in *ACM symposium on Theory of computing (STOC97)*, (El Paso, Texas, United States), pp. 304–313, May 1997.

[31] O. Goldreich and R. Ostrovsky, “Software protection and simulation on oblivious RAMs,” *Journal of the ACM*, vol. 43, no. 3, pp. 431–473, 1996.

[32] P. Williams, R. Sion, and B. Carbunar, “Building castles out of mud: Practical access pattern privacy and correctness on untrusted storage,” in *ACM Conference on Computer and Communications Security (CCS ’08)*, (Alexandria, VA), pp. 139–148, Oct. 2008.

[33] H. Weatherspoon, C. Wells, and J. Kubiatowicz, “Naming and integrity: Self-verifying data in peer-to-peer systems.,” in *Future Directions in Distributed Computing (FuDiCo)*, Lecture Notes in Computer Science, (Bologna, Italy), pp. 142–147, June 2002.

[34] P. Maniatis and M. Baker, “Enabling the archival storage of signed documents,” in *FAST ’02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, (Monterey, CA), 2002.

[35] P. Devanbu, M. Gertz, C. Martel, and S. G. Stubblebine, “Authentic data publication over the Internet,” *Journal Computer Security*, vol. 11, no. 3, pp. 291–314, 2003.

[36] J. Benaloh and M. de Mare, “One-way accumulators: A decentralized alternative to digital signatures,” in *Workshop on the Theory and Application of Cryptographic Techniques on Advances in Cryptology (EuroCrypt ’93)*, (Lofthus, Norway), pp. 274–285, May 1993.

[37] J. Camenisch and A. Lysyanskaya, “Dynamic accumulators and application to efficient revocation of anonymous credentials,” in *CRYPTO ’02*, (Santa Barbara, CA), pp. 61–76, Aug. 2002.

[38] M. T. Goodrich, R. Tamassia, and J. Hasic, “An efficient dynamic and distributed cryptographic accumulator,” in *Proceedings of the 5th International Conference on Information Security (ISC)*, (Sao Paulo, Brazil), pp. 372–388, Sept. 2002.

[39] C. Papamanthou, R. Tamassia, and N. Triandopoulos, “Authenticated hash tables,” in *ACM Conference on Computer and Communications Security (CCS ’08)*, (Alexandria, VA), pp. 437–448, Oct. 2008.

[40] K. Pavlou and R. T. Snodgrass, “Forensic analysis of database tampering,” in *ACM SIGMOD International Conference on Management of Data*, (Chicago, IL), pp. 109–120, June 2006.

[41] Q. Zhu and W. W. Hsu, “Fossilized index: The linchpin of trustworthy non-alterable electronic records,” in *ACM SIGMOD International Conference on Management of Data*, (Baltimore, MD), pp. 395–406, June 2005.

[42] S. Mitra, W. W. Hsu, and M. Winslett, “Trustworthy keyword search for regulatory-compliant records retention,” in *International Conference on Very Large Databases (VLDB)*, (Seoul, Korea), pp. 1001–1012, Sept. 2006.

[43] M. Bellare and S. K. Miner, “A forward-secure digital signature scheme,” in *CRYPTO ’99*, (Santa Barbara, CA), pp. 431–448, Aug. 1999.

[44] R. Gennaro and P. Rohatgi, “How to sign digital streams,” in *CRYPTO ’97*, (Santa Barbara, CA), pp. 180–197, Aug. 1997.

[45] J. E. Holt, “Logcrypt: Forward security and public verification for secure audit logs,” in *Australasian Workshops on Grid Computing and E-research*, (Hobart, Tasmania, Australia), 2006.

[46] D. Ma and G. Tsudik, “A new approach to secure logging,” *Transactions on Storage*, vol. 5, no. 1, pp. 1–21, 2009.

[47] D. Ma, “Practical forward secure sequential aggregate signatures,” in *Proceedings of the 2008 ACM symposium on Information, computer and communications security (ASIACCS’08)*, (Tokyo, Japan), pp. 341–352, Mar. 2008.

[48] D. Ma and G. Tsudik, “Forward-secure sequential aggregate authentication,” in *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, (Oakland, CA), pp. 86–91, IEEE Computer Society, May 2007.

[49] D. X. Song, D. Wagner, and A. Perrig, “Practical techniques for searches on encrypted data,” in *IEEE Symposium on Security and Privacy*, (Berkeley, CA), pp. 44–55, May 2000.

[50] B. R. Waters, D. Balfanz, G. Durfee, and D. K. Smetters, “Building an encrypted and searchable audit log,” in *Network and Distributed System Security Symposium (NDSS)*, (San Diego, CA), Feb. 2004.

[51] E. Kiltz, A. Mityagin, S. Panjwani, and B. Raghavan, “Append-only signatures,” in *International Colloquium on Automata, Languages and Programming*, (Lisboa, Portugal), July 2005.

[52] J. Bethencourt, D. Boneh, and B. Waters, “Cryptographic methods for storing ballots on a voting machine,” in *Network and Distributed System Security Symposium (NDSS)*, (San Diego, CA), Feb. 2007.

[53] D. Molnar, T. Kohno, N. Sastry, and D. Wagner, “Tamper-evident, history-independent, subliminal-free data structures on PROM storage -or- How to store ballots on a voting machine (extended abstract),” in *IEEE Symposium on Security and Privacy*, (Oakland, CA), May 2006.

[54] S. Haber and W. S. Stornetta, "How to time-stamp a digital document," in *CRYPTO '98*, (Santa Barbara, CA), pp. 437–455, 1990.

[55] K. Blibech and A. Gabillon, "CHRONOS: An authenticated dictionary based on skip lists for timestamping systems," in *Workshop on Secure Web Services*, (Fairfax, VA), pp. 84–90, Nov. 2005.

[56] A. Buldas, P. Laud, H. Lipmaa, and J. Willemson, "Time-stamping with binary linking schemes," in *CRYPTO '98*, (Santa Barbara, CA), pp. 486–501, Aug. 1998.

[57] A. Buldas, H. Lipmaa, and B. Schoenmakers, "Optimally efficient accountable time-stamping," in *International Workshop on Practice and Theory in Public Key Cryptography (PKC)*, (Melbourne, Victoria, Australia), pp. 293–305, Jan. 2000.

[58] H. Lipmaa, "On optimal hash tree traversal for interval time-stamping," in *Proceedings of the 5th International Conference on Information Security (ISC02)*, (Seoul, Korea), pp. 357–371, Nov. 2002.

[59] P. Maniatis and M. Baker, "Secure history preservation through timeline entanglement," in *USENIX Security Symposium*, (San Francisco, CA), Aug. 2002.

[60] D. Sandler and D. S. Wallach, "Casting votes in the Auditorium," in *USENIX/ACCURATE Electronic Voting Technology Workshop (EVT'07)*, (Boston, MA), Aug. 2007.

[61] H. Chan, A. Perrig, B. Przydatek, and D. Song, "SIA: Secure information aggregation in sensor networks," *Journal Computer Security*, vol. 15, no. 1, pp. 69–102, 2007.

[62] L. Hu and D. Evans, "Secure aggregation for wireless networks," in *Symposium on Applications and the Internet Workshops (SAINT)*, (Orlando, FL), p. 384, July 2003.

[63] H. Chan, A. Perrig, and D. Song, "Secure hierarchical in-network aggregation in sensor networks," in *ACM Conference on Computer and Communications Security (CCS '06)*, (Alexandria, VA), pp. 278–287, Oct. 2006.

[64] M. Manulis and J. Schwenk, "Provably secure framework for information aggregation in sensor networks," in *Computational Science and Its Applications (ICCSA)*, (Kuala Lumpur, Malaysia), pp. 603–621, Aug. 2007.

[65] R. T. Snodgrass, S. S. Yao, and C. Collberg, "Tamper detection in audit logs," in *Conference on Very Large Data Bases (VLDB)*, (Toronto, Canada), pp. 504–515, Aug. 2004.

[66] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative byzantine fault tolerance," in *SOSP '07*, (Stevenson, WA), pp. 45–58, Oct. 2007.

[67] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz, "Attested append-only memory: Making adversaries stick to their word," in *SOSP '07*, (Stevenson, WA), pp. 189–204, Oct. 2007.

[68] A. Haeberlen, P. Kouznetsov, and P. Druschel, "PeerReview: Practical accountability for distributed systems," in *SOSP '07*, (Stevenson, WA), Oct. 2007.

[69] J. D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline, "Detection of mutual inconsistency in distributed systems," *IEEE Transactions on Software Engineering*, vol. 9, no. 3, pp. 240–247, 1983.

[70] B. Schneier and J. Kelsey, “Automatic event-stream notarization using digital signatures,” in *Security Protocols Workshop*, (Cambridge, UK), pp. 155–169, Apr. 1996.

[71] A. R. Yumerefendi and J. S. Chase, “Strong accountability for network storage,” *ACM Transactions on Storage*, vol. 3, no. 3, 2007.

[72] P. Maniatis, M. Roussopoulos, T. J. Giuli, D. S. H. Rosenthal, and M. Baker, “The LOCKSS peer-to-peer digital preservation system,” *ACM Transactions on Computer Systems*, vol. 23, no. 1, pp. 2–50, 2005.

[73] New York Times, *HORSE RACING; 3 Sentenced in Breeders’ Cup Betting Plot*, Mar. 21 2003. p. S-3.

[74] J. Drape, “Horse racing; Ways to keep schemers from beating the system,” in *New York Times*, pp. D–8, Oct. 22 2003.

[75] Office of the Kansas Secretary of State, “Voting system security policy,” Mar. 2004. `http://www.kssos.org/other/voting_security_policy.html`.

[76] M. Bellare and B. S. Yee, “Forward integrity for secure audit logs,” tech. rep., University of California at San Diego, Nov. 1997.

[77] G. Itkis, “Cryptographic tamper evidence,” in *ACM Conference on Computer and Communications Security (CCS ’03)*, (Washington D.C.), pp. 355–364, Oct. 2003.

[78] R. Accorsi and A. Hohl, “Delegating secure logging in pervasive computing systems,” in *Security in Pervasive Computing*, (York, UK), pp. 58–72, Apr. 2006.

[79] W. Pugh, “Skip lists: A probabilistic alternative to balanced trees,” *Communications of the ACM*, vol. 33, pp. 668–676, June 1990.

[80] B. H. Bloom, “Space/time trade-offs in hash coding with allowable errors,” *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[81] E.-J. Goh, “Secure indexes.” Cryptology ePrint Archive, Report 2003/216, 2003. `http://eprint.iacr.org/2003/216/` See also `http://eujingoh.com/papers/secureindex/`.

[82] C. Lonvick, “The BSD Syslog protocol.” RFC 3164, Aug. 2001. `http://www.ietf.org/rfc/rfc3164.txt`.

[83] S. D. S. Monteiro and R. F. Erbacher, “Exemplifying attack identification and analysis in a novel forensically viable Syslog model,” in *Workshop on Systematic Approaches to Digital Forensic Engineering*, (Oakland, CA), pp. 57–68, May 2008.

[84] J. Kelsey, J. Callas, and A. Clemm, “Signed Syslog messages.” `http://tools.ietf.org/id/draft-ietf-syslog-sign-23.txt` (work in progress), Sept. 2007.

[85] P. Deutsch, “Gzip file format specification version 4.3.” RFC 1952, May 1996. `http://www.ietf.org/rfc/rfc1952.txt`.

[86] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, “Making data structures persistent,” in *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing (STOC)*, (Berkeley, CA), pp. 109–121, May 1986.

[87] A. Fiat and H. Kaplan, “Making data structures confluently persistent,” *Journal of Algorithms*, vol. 48, no. 1, pp. 16–58, 2003.

[88] D. Micciancio, “Oblivious data structures: Applications to cryptography,” in *Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC)*, (El

Paso, Texas), pp. 456–464, May 1997.

[89] M. Naor and V. Teague, “Anti-presistence: History independent data structures,” in *Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing (STOC)*, (Heraklion, Crete, Greece), pp. 492–501, July 2001.

[90] A. Anderson and T. Ottmann, “Faster uniquely represented dictionaries,” in *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science (SFCS)*, (San Juan, Puerto Rico), pp. 642–649, Oct. 1991.

[91] C. R. Aragon and R. G. Seidel, “Randomized search trees,” in *Proceedings of the 30th Annual Symposium on Foundations of Computer Science (SFCS)*, pp. 540–545, Oct. 1989.

[92] G. E. Blelloch and M. Reid-Miller, “Fast set operations using treaps,” in *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, (Puerto Vallarta, Mexico), pp. 16–26, June 1998.

[93] L. J. Guibas and R. Sedgewick, “A dichromatic framework for balanced trees,” in *Proceedings of the 19th Annual Symposium on Foundations of Computer Science (SFCS)*, pp. 8–21, Oct. 1978.

[94] G. S. Brodal, “Partially persistent data structures of bounded degree with constant update time,” *Nordic Journal of Computing*, vol. 3, no. 3, pp. 238–255, 1996.

[95] H. Kaplan, “Persistent data structures,” in *Handbook on Data Structures and Applications* (D. Mehta and S. Sahni, eds.), CRC Press, 2001.

[96] C. Okasaki, *Purely Functional Data Structures*. Cambridge University Press, 1999.

[97] P. Bagwell, "Fast functional lists, hash-lists, deques and variable length arrays," in *In Implementation of Functional Languages, 14th International Workshop*, (Madrid, Spain), p. 34, Sept. 2002.

[98] S. Micali, "Efficient certificate revocation," Tech. Rep. TM-542b, Massachusetts Institute of Technology, Cambridge, MA, 1996. `http://www.ncstrl.org:8900/ncstrl/servlet/search?formname=detail\&id=oai%3Ancstrlh%3Amitai%3AMIT-LCS%2F%2FMIT%2FLCS%2FTM-542b`.

[99] D. Naccache, D. M'Raihi, S. Vaudenay, and D. Raphaeli, "Can DSA be improved? Complexity trade-offs with the digital signature standard," in *EuroCrypt*, (Perugia, Italy), pp. 77 – 85, May 1994.

[100] J. Li, N. Li, and R. Xue, "Universal accumulators with efficient nonmembership proofs," in *Proceedings of the 5th International Conference on Applied Cryptography and Network Security (ACNS)*, (Zhuhai, China), pp. 253–269, June 2007.

[101] P. Wang, H. Wang, and J. Pieprzyk, "A new dynamic accumulator for batch updates," in *Information and Communications Security, 9th International Conference (ICICS 2007)*, (Zhengzhou, China), pp. 98–112, Dec. 2007.

[102] P. Wang, H. Wang, and J. Pieprzyk, "Improvement of a dynamic accumulator at ICICS 07 and its application in multi-user keyword-based retrieval on encrypted data," in *Asia-Pacific Services Computing Conference*, (Yilan, Taiwan), pp. 1381–1386, Dec. 2008.

[103] N. Bari and B. Pfitzmann, "Collision-free accumulators and fail-stop signature schemes without trees," in *EuroCrypt*, (Konstanz, Germany), pp. 480–494, May 1997.

[104] NIST Special Publication 800-57, *Recommendation for Key Management — Part 1: general*. National Institute for Standards and Technology, Mar. 2007.

[105] M. O. Rabin, "Probabilistic algorithm for testing primality," *Journal of Number Theory*, vol. 12, no. 1, pp. 128–138, 1980.

[106] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor, "Checking the correctness of memories," in *Proceedings of the 32nd annual symposium on Foundations of computer science (SFCS)*, (San Juan, Puerto Rico), pp. 90–99, Oct. 1991.

[107] C. Dwork, M. Naor, G. N. Rothblum, and V. Vaikuntanathan, "How efficient can memory checking be?," in *Proceedings of the Theory of Cryptography Conference (TCC)*, (San Francisco, CA), pp. 503–520, Mar. 2009.

[108] L. Nguyen, "Accumulators from bilinear pairings and applications," in *Cryptographers' Track at the RSA Conference (CT-RSA)*, (San Francisco, CA), pp. 275–292, Feb. 2005.

[109] J. Camenisch, M. Kohlweiss, and C. Soriente, "An accumulator based on bilinear maps and efficient revocation for anonymous credentials," in *12th International Conference on Practice and Theory in Public Key Cryptography (PKC2009)*, (Irvine, CA), pp. 481–500, Mar. 2009.