DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2017

Privacy Preserving Audit Proofs

ANTON LINDQVIST

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF COMPUTER SCIENCE AND COMMUNICATION

Privacy Preserving Audit Proofs

ANTON LINDQVIST

Master in Computer Science
Date: July 2, 2017
Supervisor: Roberto Guanciale
Examiner: Mads Dam
Principal: Hannes Salin, Omegapoint
Swedish title: Integritetsbevarande bevis av digitalt spårbara händelser
School of Computer Science and Communication

Abstract

The increased dependence on computers for critical tasks demands sufficient and transparent methods to audit its execution. This is commonly solved using logging where the log must not only be resilient against tampering and rewrites in hindsight but also be able to answer queries concerning (non)-membership of events in the log while preserving privacy. Since the log cannot assume to be trusted the answers must be verifiable using a proof of correctness.

This thesis describes a protocol capable of producing verifiable privacy preserving membership proofs using Merkle trees. For non-membership, a method used to authenticate Bloom filters using Merkle trees is proposed and analyzed. Since Bloom filters are a probabilistic data structures, a method of handling false positives is also proposed.

Sammanfattning

Den ökande avlastningen av kritisk funktionalitet till datorer ställer högre krav på loggning och möjlighet till övervakning. Loggen måste vara resistent mot manipulation och möjliggöra för andra parter att ställa frågor berörande en viss händelse i loggen utan att läcka känslig information. Eftersom loggen inte antas vara att lita på måste varje svar vara verifierbart med hjälp av ett bevis.

Denna rapport presenterar ett protokoll kapabelt till att producera verifierbara och integritetsbevarande svar på frågor om en viss händelse i loggen genom användning av Merkle-träd. Vid avsaknad av den förfrågade händelsen används ny metod för att autentisera ett Bloom filter med hjälp av Merkle-träd. Eftersom Bloom filtren är en probabilistisk konstruktion presenteras även en metod för att hantera falsk positiva svar.

Contents

Chapter 1

Introduction

Many critical tasks are by now delegated to computers due to its efficiency and reliability. This trend is not believed to decrease, but rather the opposite. The increased dependence on computers requires sufficient and transparent methods to audit and monitor its execution. The logs produced therefore act as evidence of the execution and the integrity of these logs must remain intact in order preserve history. Tampering and deletion of such logs must not go by unnoticed since it would disarm its reliance. Trusting computers to behave honestly is a weak assurance since trust begins where security ends.

Not only are the logs of importance to operation managers but also in dispute resolutions and as legal evidence. As an example, banking institutes are occasionally requested to provide evidence of existence for a certain transaction to the law enforcement. Being able to audit the log of transactions is indispensable in this scenario. The principal supervisor of this thesis have been asked for such a feature in private communication in his daily work with Swedish banks. Using the complete log as evidence is not an option due to its intractable size and more importantly, the actual evidence only concerns a subset of the log while the rest contains sensitive information an auditor is not entitled to access. Similar problems also exist in the medical domain since they acquire sensitive information tied to their customers and clients. Such information should never be leaked to unauthorized sources since it would infringe the integrity of their clients. The upcoming enrollment of the new General Data Protection Regulation (GDPR) will cause companies with customers resident in the EU-region with similar problems due to stricter requirements of protection of sensitive information tied to their customers.

Ideally, a query regarding a subset of the log must be answerable and the auditor must be able to verify the correctness of the answer while still preserving privacy. No additional knowledge concerning the log should be gained other than the mere answer to the query. The requirement of constructing a proof for each answered query implies a lack of trust towards the log since it is not expected to behave honestly in all possible scenarios due to potential vulnerabilities being exploited. Such proof also enables the auditor to verify that the history, as presented by the log, subjectively reflects the true order of the events as they occurred, minimizing the risk of tampering by someone with malicious intents to go by unnoticed.

Logging is often performed in a structured manner where each log entry is constituted by key-value pairs forming a list of such pairs. A query on such a log is only concerned about the presence or absence of a specific key-value pair which will be referred to as a (non)-membership proof. As presented in this thesis, several methods has been proposed as a solution to the construction of such proofs. An undesirable limitation of the previous work is often caused by constraints on the data to prove (non)-membership of, such as requiring a data structure with a capacity proportional to the cardinality of the input domain as opposed of letting the capacity be proportional to the number of values stored in the data structure. Since the main application of the work present in this thesis is made with logging in mind such limitation is sub-optimal. In addition, proposals on how a proof mechanism can be incorporated in a formalized protocol has received less attention. A majority of the related work is limited to particular property of such protocol, e.g. construction of privacy preserving membership proofs or privacy preserving comparison of data. An opportunity therefore exists to formalize a protocol capable of constructing (non)-membership proofs while still offer a higher degree of flexibility regarding the contents of the log entries. This is especially applicable in practice where the number of keys in a log entry may change over time or the values be of varying length. In addition, the protocol must be resilient against tampering, deletion and corruption of the log.

1.1 Research Question

The main objective of this thesis is to formalize a protocol capable of constructing (non)-membership proofs in an efficient manner while still preserving privacy. Such protocol must also enable tampering, deletion and corruption to be detectable by external auditors and thus certify history. The objectives can be summarized by the requirements as follows:

1. Efficient construction of (non)-membership proofs

   The construction of (non)-membership proofs must be efficient in terms of both time and space.

2. Preserve privacy

   An auditor requesting an audit proof should not gain any additional information by examining the proof other than the mere answer to the corresponding query which is only concerns a subset of the log. Since an auditor is only entitled to access a specific subset of the log, it is assumed that the auditor has knowledge of this particular subset. Additional information in this context refers to data not enclosed in the requested subset. Achieving complete preservation of privacy is an ambitious goal and this requirement is therefore relaxed to minimize leakage without impacting the efficiency of protocol in a drastic manner.

3. Certify history

   The protocol must enable tampering, deletion and corruption to be detectable by the auditors.

The protocol as proposed in this thesis will be evaluated according to these requirements.

1.2 Contributions

A method involving Merkle trees is proposed as a solution to the construction of membership proofs which preserves privacy. For non-membership, a method used to authenticate Bloom filters is proposed which enables such proofs to be constructed. By using the proposed caching strategy for the authenticated Bloom filters the time taken to construct such proofs is improved at a cost of trading a low amount of space. Since Bloom filters are a probabilistic data structure, a method of handling false positives is needed which is incorporated in the protocol.

1.3 Delimitations

The communication between the parties in the proposed protocol is assumed to be secure. How the communication is secured is of less importance for the protocol since it is considered an implementation detail. However, the protocol must not dictate any requirements that would hinder the process of securing the communication.

No suitable state-of-the-art implementation offering feature-parity with the proposed protocol was found during the background research. The implementation could therefore not be benchmarked against existing solutions. Except for the construction of non-membership proofs.

The proposed caching strategy for the authenticated Bloom filters is partially caused by the decision to not evaluate different cache eviction policies as part of this thesis. Such analysis would broaden the scope of thesis beyond feasible.

Chapter 2

Background

2.1 Hash Functions

2.1.1 Non-Secure Hash Functions

A non-secure hash function h is a unary function that maps data of arbitrary length to a fixed number of bits: $h : \{0,1\}^* \rightarrow \{0,1\}^k$. Its output is distributed uniformly.

Murmur hashing is one of the more commonly used hash function in the family of non-secure hash functions [1], invented by Appleby [2].

2.1.2 Secure Hash Functions

A secure hash function, also known as a cryptographic hash function, is considered a secure one-way function if the following properties are satisfied:

1. Pre-image resistant:
   A hash function h is pre-image resistant if it is hard to invert the function yielding x only given h and y where $y = h(x)$.

2. Second pre-image resistant:
   A hash function h is second pre-image resistant if it is hard to find x_2 given x_1 where $h(x_1) = h(x_2)$.

3. Collision resistant:
   A hash function h is collision resistant if it is hard to find any pair of x_1 and x_2 where $h(x_1) = h(x_2)$. This property differs from the second in the sense that both inputs x_1 and x_2 are unknown a priori.

The use of the word *hard* in this context refers to the unknown existence of a computationally feasible function capable of invalidating any of the three properties. Breaking pre-image resistance using a brute-force attack would require one to randomly choose x until $y = h(x)$. Since $h(x)$ maps to a space defined by $\{0,1\}^k$, 2^k different results are possible with equal probability 2^{-k}. On average, $2^k/2$ different inputs has to be examined until x where $y = h(x)$ is found. The same reasoning also applies to the matter of breaking the second pre-image property. Breaking collision resistance requires, on average, fewer attempts due to the birthday paradox: $\sqrt{\frac{\pi m}{2}}$, where $m = 2^k$ and $\sqrt{\frac{\pi 2^k}{2}} < \frac{2^k}{2}$ [3].

2.2 Pseudorandom Functions

A pseudorandom function is an efficient deterministic function capable of emulating generation of random values. In its essence, it is a mapping from the range of input values to the range of output values where each pair of input and output is distributed uniformly in a random manner. No efficient algorithm should be able to distinguish the output from a truly random function and a pseudorandom function. The generated values of such a function is determined by an initial seed which remains unknown to the caller. Thus, given a seed and a sequence of inputs it will always produce the same output [4].

2.3 Merkle Trees

A Merkle tree, also known as a hash tree, stores its data in the leaf nodes and all other non-leaf nodes (referred to as intermediate nodes) stores the hash of its children. The intermediate nodes therefore act as a fingerprint of the data present in its children. Thus, the root node is a fingerprint for the whole tree and its data [5]. Merkle trees was invented by Merkle [6] as a method used to authenticate large amounts of data.

Merkle trees are recursively constructed, starting in the leaves where the hash of the associated data is computed. All intermediate nodes concatenate the hash from its children and uses it as the input to the hash function, the hash will therefore solely rely on the values from its children. In Figure 2.1, l_i denotes the ith leaf node with the corresponding data d_i and i_j the jth intermediate node. Each node is labeled with its hash calculation where $||$ denotes the concatenation operator.

Presence of a particular leaf node can be proven by constructing an audit proof. The proof will guarantee that the data stored in the leaf node is a partial causation of the root node hash. An audit proof for leaf l_1 in Figure 2.1 consists of $\{l_0, l_1, i_2\}$. The corresponding hashes for those nodes allow the root node hash to be computed and compared against the advertised root node hash for verification. This assumes knowledge of the hash function used to compute the node hashes [7]. Information regarding whenever an intermediate node is identified as a node on the left or right is also of importance in order to re-construct the root node hash in correctly [8].

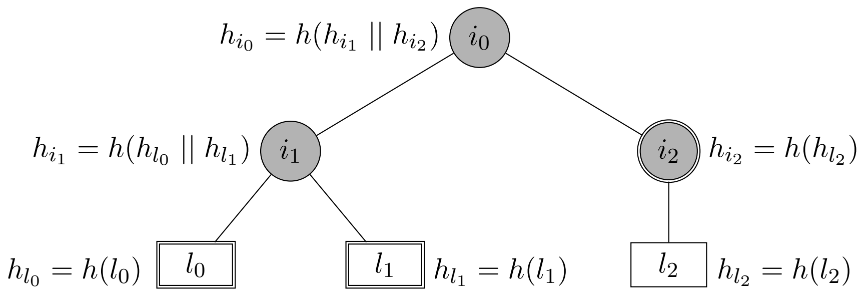

Figure 2.1: Merkle tree where white squares denote leaf nodes and gray circles intermediate nodes. The nodes with a double border, $\{l_0, l_1, i_2\}$, constitute the audit proof for l_1. All nodes are annotated with their corresponding hash calculation.

The size of an audit proof for a tree containing N leafs using a hash function h :

$\{0, 1\}^* \rightarrow \{0, 1\}^k$ requires $k\lceil\log N\rceil$ bits. That is, the hash of the $\log N$ number of sibling nodes along the path to the leaf node to prove existence of is only needed [9].

The audit proof has several applications, one being the ability to prove presence of a leaf without revealing the complete tree. Imagine a bank storing its transactions in a Merkle tree. Proving the existence of a certain transaction in the event of a dispute between the bank and a customer could then be performed without revealing transactions performed by others [7].

The other type of proof a Merkle tree can emit is regarding its consistency. Merkle trees can be joined together, forming a larger tree. Proving that a pre-existing tree constitutes a subset of the new tree is done using a consistency proof. In Figure 2.2, the tree $\mathcal{T}$ has been joined together with another tree, forming the final tree $\mathcal{T}^{'}$. In order to proof that $\mathcal{T}$ is a subset of $\mathcal{T}^{'}$ the hashes for the nodes $\{l_0, l_1, l_2, l_3, \mathcal{T}^{'}_4\}$ are needed in order to compute the root node hash which then can be compared against the advertised root node hash. Equality implies that tree to the left is a subset of the new tree.

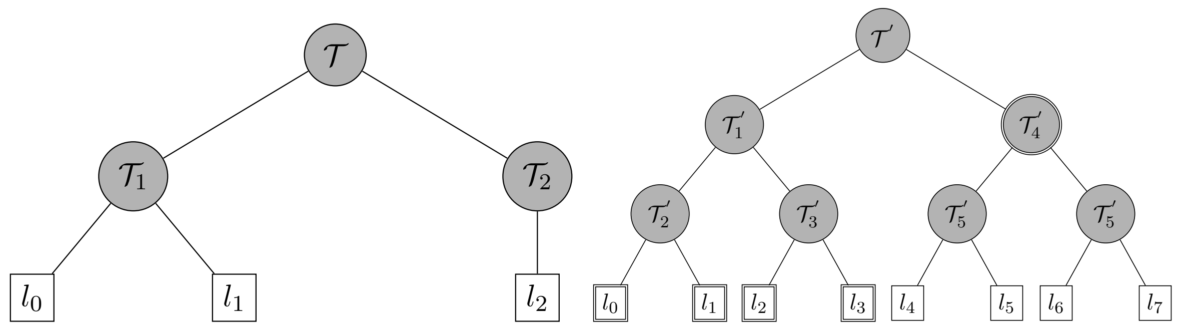

Figure 2.2: Merkle trees where nodes with a double border constitutes the consistency proof for the tree $\mathcal{T}$ being a sub-tree of the tree $\mathcal{T}^{'}$.

An application where Merkle trees are used is to enable integrity verification of system logs. In this type of application, proofs regarding the consistency of the log must be constructed in such way that it does not leak information about other leaf nodes. The choice of a secure hash function is of importance since the security of the tree is reduced to this function [10]. Such function is believed to not be invertible and provide resistance against collisions. However, system log entries may not contain sufficient entropy to ensure this property holds since their format is often predictable or follows a predefined scheme. An attacker who is aware of such schemes could therefore exhaustively generate all values and ask for an audit proof for each of them. In order to prevent such an attack a proposed solution is to concatenate the log entry with a random piece of data with sufficient entropy [7].

2.3.1 Sparse Merkle Trees

A sparse Merkle tree is a tree where the majority of the leaves are empty. First proposed by Laurie and Kasper [11] as an efficient method to prove whenever a Transport Layer Security (TLS) certificate is revoked or not. Constructing a tree including every possible certificate and its corresponding binary revocation status using an underlying hash function $h : \{0, 1\}^* \rightarrow \{0, 1\}^n$ would require a tree with 2^n leaves, where $n = 256$ in this particular scenario. The key observation is that such a tree contains many empty leaves and is therefore sparse. Every pair of empty leafs will have the same hash value $h(0)$ and its corresponding parent node $h(h(0) \,||\, h(0))$, the process is repeated according to this recursive construction. The empty leafs and their corresponding paths does not need to be explicitly stored but instead be simulated upon request. The amount of space needed for such tree is therefore reduced to the number of non-empty leafs as opposed of storing all leafs.

The work by Laurie and Kasper [11] was later formalized by Dahlberg et al. [10] who also improved the performance of the simulation by developing a caching strategy. The utilized cache is significantly smaller than the actual tree and uses a probabilistic approach where an intermediate node is cached with a probability. Their implementation is capable of generating audit proofs in less than 4ms using SHA-$\{256, 512\}$ as the underlying hash function.

2.3.2 Sorted Merkle Trees

A sorted Merkle tree retains all leafs in an ordered manner. This allows non-membership proofs to be constructed given an absent leaf by proving that the neighboring leafs do not contain the given leaf. Such proof relies on the underlying ordering of leafs in order to prove that the neighboring leafs constitute an interval which does not cover the given leaf [12]. The proof does however leak information about the neighboring leafs present in the tree, even if the data stored in each leaf is obfuscated [9].

2.4 Bloom Filters

A Bloom filter is a probabilistic data-structure developed by Bloom [13] that is used for membership queries on sets. Due to its probabilistic construction, false positives are possible but false negatives are not. A filter is represented as an array of m bits initially set to 0. Every filter has k distinct hash functions where $h_i : \{0, 1\}^* \to [1, m]$ for $i \in [1, k]$ that maps the given input to a bit in the array. Each member of a set $s \in S$ where $|S| = n$ is inserted into the filter by giving s as input to all k hash functions and setting the corresponding bits to 1 in the underlying array. Every bit can be set to 1 multiple times, but only the first is noted.

To determine if an element is present in the filter, all bits given by the k hash functions using the element as input has to be set to 1. If any of the bits are not set, the element is not in the set. Otherwise, the element is in the set with a high probability. This property can be leveraged to provide private indexing as proposed by Goh et al. [14]. Observing the underlying array of bits will not reveal any of its members since different members could map to the same bit.

The optimal number of hash functions given m and n is given by the formula:

$$k = \frac{m}{n} \ln 2 \tag{2.1}$$

If n and the desired false positive probability p is known, the required number of bits and hash functions is given by the formulas:

$$m = -\frac{n \ln p}{(\ln 2)^2} \tag{2.2}$$

$$k = -\frac{\ln p}{\ln 2} \tag{2.3}$$

The trade-offs and impact on both space and time tied to the choice of m, n and k and is further discussed in detail by Bloom [13].

Chapter 3

Related Work

In this chapter, the related work in the field of both privacy preserving audits and the applications of Merkle trees is presented. Every finding is summarised and its applicability in the context of this thesis is evaluated.

In the era of cloud-computing, customers trust external services to store their private data such as emails and photos. This trust is manifested in the service's ability to prevent the data from being exposed to unauthorized parties and being resilient to corruption and deletion. In order to move the burden of verifying the integrity of the stored data from the customers, many protocols involving a third party auditor has been proposed [14, 15, 16]. Common for all the proposed methods is the mechanism of letting the client split the data into blocks, encrypting them and generate a signature for the encrypted blocks. The encrypted blocks and signatures are then sent to the service. The integrity of the service relies on the correctness of these signatures [17]. Further efforts to make the encrypted blocks searchable by the service without the need to decrypt the data has been proposed by Cao et al. [18].

Lee et al. [19] describe a recent trend in computing: the rise of distributed networks. Such networks due to its distributed setup must also be audited in a distributed manner in order to detect intrusions since the amount of work no longer can be handled by a single node in the network. The authors provide a framework capable of performing distributed audits without requiring excessive sharing of private information and obfuscates the data being audited in order to minimize information leak. Despite being obfuscated, the ordering of the data is still intact which implies that the data can still be compared for equality. Assuming the data has some partial ordering, an auditor must be able to determine whether one value is greater than another value with respect to the ordering and without revealing any information about the actual values being compared. This is done by representing each value as a separate Bloom filter where each filter contains the value and all the values it dominates. Each filter is also equipped with a collision counter initialized to 0. In Figure 3.1 the set $\{a, b, c\}$ where $a < b < c$ and their corresponding Bloom filters B_a, B_b, B_c and collision counters c_a, c_b, c_c are illustrated. Each Bloom filter consists of 5 bits, using two hash functions h_1 and h_2. The number of set bits in each Bloom filter reflects the underlying ordering of the set without revealing the actual values, thus $|B_a| = 2 < |B_b| = 3 < |B_c| = 4 \iff a < b < c$. Since $h_1(c)$ and $h_2(c)$ maps to bits already set, its collision counter is increment and then hashed, causing a new bit to be set and the ordering to be preserved. However, the framework does not produce any proofs regarding the correctness of the comparisons.

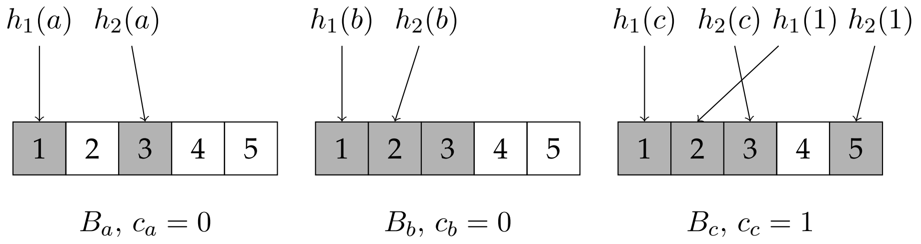

Figure 3.1: Bloom filters for each member of the set $\{a, b, c\}$ where gray boxes denote set bits.

Today's browsers are capable of detecting forged or faked SSL certificates. However, when a certificate issued by a certificate authority (CA) that has been compromised or gone rogue, browsers remain unaware. Recently, a Dutch CA was compromised and the attackers were able to access the system and issue new fake certificates. Since the CA was trusted, the browsers would also imply trust to all certificates issued by the same CA [20]. Laurie et al. [21] proposed a solution to this problem by storing all valid certificates in a Merkle tree. Its append-only characteristic is suitable in such a scenario where certificates cannot be modified in hindsight. Proof of existence given a certificate can also be computed in an efficient manner since the size of the proof will be proportional to the height of tree. The proposal also makes the task of auditing available to the public, hence the name Certificate Transparency.

Crosby and Wallach [22] explored the idea of being able to query a Merkle tree based on only a fraction of the data present in the leafs. In the scenario described, each leaf is treated as a set of attributes and only a subset of these attributes is of interest. Each intermediate node is annotated with data regarding the attributes present in its descending leaf nodes. Where the canonical example being: each leaf node is annotated with an important attribute, if applicable. Any intermediate node where there exists a path to an important leaf node is also annotated as important. On request, the tree can then be queried to only reveal all important leaf nodes without the need to examine all leaf nodes. In order to satisfy such a request the tree only needs to know about the associated attributes for each leaf node which are replicated along the intermediate nodes. The actual data stored in each leaf node could therefore be encrypted without sacrificing the ability to perform fine-grained audits. The down-side with this method is redundant data replication inside the intermediate nodes.

Other data structures than Merkle trees has been authenticated. Goodrich and Tamassia [8] used a hierarchical hashing technique with skip lists that produces set membership proofs of size $O(log\, n))$. The skip list which represents a set is maintained by an untrusted party which periodically receives signed updates from trusted parties. A user may then ask the untrusted party for a membership proof of an element. The correctness of the proof is verified using the signatures from the trusted source. The security is therefore reduced to the integrity and correctness of these signatures. One of the authors Tamassia [23], also did a recent evaluation of the field of authenticated data structures.

Micali et al. [24] generalized the idea of using a Merkle tree as the underlying data structure to produce zero-knowledge proof for (non)-membership of an element in a finite set. A zero-knowledge proof allows one party (the prover) to prove to another party (the verifier) that a given statement is true without giving away any information except the fact that the statement holds true [25]. Zero-knowledge is achieved by using a public commitment: the tree commits to the members in the set. The commitment can then be used to prove (non)-membership of a given element without the need to construct a traditional audit proof which would leak structural information about the tree. The downside with this method is the requirement on tree to be proportional to the cardinality of the input domain rather than the size of the set. For instance, the set of Swedish Social Security numbers where each element consists of 10 digits would require a tree proportional to the cardinality of the set: 10^{10}.

Chapter 4

Protocol

This chapter presents and formalizes the protocol by describing the methodology used, followed by an overview of the parties involved in the protocol and the communication between them. The proposed method on how to authenticate Bloom filters is then presented since it is a pre-requisite to the construction of non-membership proofs. The solution the to construction of (non)-membership proofs is then presented followed by an analysis of the protocol.

4.1 Methodology

In order to evaluate the protocol it is analysed and evaluated according to the method as follows:

This chapter is ended with a security analysis of the protocol with respect to the aspects of completeness, soundness and privacy preserving. The first aspect, completeness, covers the ability of the protocol to behave correct and honestly under the assumption that all involved parties act in a manner as defined by the protocol. Soundness concerns the opposite scenario in which a party does not act honestly or in a way that is not defined by the protocol. Counter-measures against such behavior is also presented in this part of the analysis. Privacy preserving concerns the ability of the protocol to not leak information an auditor is not entitled to access.

In the Methods 5 chapter, a description on how the proposed protocol was implemented is presented followed by metrics on its performance in the Results 6 chapter which are further discussed in the Conclusions 7 chapter. Since this analysis concern the performance of the implementation it allows the feasibility of the protocol in practice to be evaluated.

To summarize, the fulfillment of the protocol requirements as stated in the Research Question 1.1 is evaluated in the Conclusions 7 chapter.

4.2 Overview

The protocol involves three actors: applications, auditors and a logger.

Applications produce log entries over time, which are sent to the logger for persistence and later auditing. An application can observe the contents of every log entry it produces. An application will upon sending a log entry to the logger verify its integrity by requesting a proof for the transferred log entry. A log entry ℓ is further defined as a set of attribute tuples where each tuple (a_i, v_i) includes the name of the attribute a_i and the corresponding value v_i.

$$\ell = \{(a_1, v_1), (a_2, v_2), \ldots, (a_n, v_n)\} \tag{4.1}$$

$$|\ell| = n \tag{4.2}$$

All attribute names in each log entry is excepted to be unique:

$$(a, x) \in \ell \to (a, y) \notin \ell, x \neq y \tag{4.3}$$

Auditors are responsible for verifying that a certain event has occurred. An event is represented as a subset of attribute tuples present in a log entry. All auditors has some domain-knowledge of the attributes present in the log entries they are auditing.

The *Logger* persists log entries received from the applications. Given an attribute tuple, the logger can generate (non)-membership proofs. Further, let L denote the complete log stored by the logger.

All actors are assumed to be isolated from each other in the sense that one actor cannot directly access or modify data owned by another actor. Data may only be transmitted and received in the directions illustrated in Figure 4.1.

$$\begin{tikzcd}
Application_1 \arrow[dr, leftrightarrow] & & Auditor_1 \arrow[dl, leftrightarrow, dashed] \\
Application_2 \arrow[r, leftrightarrow] & Logger \arrow[r, leftrightarrow, dashed] & Auditor_2 \\
\cdots & & \cdots \\
Application_n \arrow[uur, leftrightarrow] & & Auditor_m \arrow[uul, leftrightarrow, dashed]
\end{tikzcd}
\qquad
\boxed{\begin{tikzcd}
P \arrow[r, leftrightarrow] & \ell \\
q \arrow[r, leftrightarrow, dashed] & P
\end{tikzcd}}$$

Figure 4.1: Protocol overview. The applications send log entries ℓ to the logger and requests audit proofs P used to verify the integrity of the transfered log entries. The auditors ask the logger if an event q has occurred and receives an audit proof P.

4.2.1 Trust

Every application is assumed to behave trustworthy and send the correct log entries to the logger as they occur. This property is ensured by an external actor which is beyond the scope for this thesis. Due to existence of vulnerabilities in the logger, an application could try to rewrite its own log entries in hindsight or completely try to delete them.

An auditor is considered untrusted if it tries to gain knowledge of attributes it is not entitled to possess knowledge of.

The logger could suffer from vulnerabilities exploited by the applications. Therefore, every audit must be strengthen by a proof to ensure that any vulnerability is detected.

4.3 Notation

- Let $h : \{0, 1\}^* \to \{0, 1\}^\lambda$ denote a secure hash function given an input of arbitrary length producing an output of λ bits. The choice of λ is a security parameter of the protocol. All parties are aware of the choice of both h and λ.
- Let $\beta_{m,k}$ denote a Bloom filter with a configuration of 2^m bits and k non-secure hash functions: $h_i : \{0, 1\}^* \to [1, 2^m]$ for $i \in [1, k]$. $\beta_{m,k}\{j\} \in \{0,1\}$ where $j \in [1, 2^m]$ denotes the state of the jth bit in the filter. All parties are aware of the choice of h_i for all $i \in [1, k]$.
- Let $\tau \overset{\$}{\leftarrow} \{0,\ 1\}^\Lambda$ be the output of a pseudorandom function which returns a uniformly chosen element in the set of binary strings of Λ bits. The choice of Λ is a security parameter of the protocol. Only the logger is aware of choice of Λ.
- Let an audit proof P be a list of hash values produced by h where the first element $P[1]$ is the hash of the requested leaf node and the last element $P[n]$ the hash of the root node, where the latter is used for verification.
- Let $log\, n$ denote the binary logarithm $log_2\, n$.
- Let $a \,||\, b$ denote the binary concatenation operator joining its operands together producing an output of length $|a| + |b|$.

4.4 Authenticated Bloom Filters

In order to verify the validity of a Bloom filter, it is authenticated using a Merkle tree where each bit in the filter is represented as a leaf in the tree. The mapping between the bits and leafs is a bijection where the ith bit in the filter maps to the ith leaf in the tree. Since the filter is represented as a tree the size of the filter must be a power of 2 in order to make the tree balanced.

Proving the validity of the state of any bit in the filter is performed by constructing an audit proof for the corresponding leaf. Proving the correctness of a membership query on the filter is done by constructing an audit proof for each bit given by the hash functions. The number of audit proofs needed is therefore equal to the number of hash functions in the filter. In Figure 4.2, the tree for $\beta_{2,2}$ with a configuration of 2^2 bits and 2 hash functions is illustrated. Given a member x with $h_1(x) = 2$ and $h_2(x) = 3$, its membership in the filter is proven by constructing an audit proof for each of the two corresponding leafs $\beta_{2,2}\{2\}$ and $\beta_{2,2}\{3\}$:

$$P_{\beta_{2,2}\{2\}} = \{\beta_{2,2}\{2\},\ \beta_{2,2}\{1\},\ i_3,\ i_1\} \tag{4.4}$$
$$P_{\beta_{2,2}\{3\}} = \{\beta_{2,2}\{3\},\ \beta_{2,2}\{4\},\ i_2,\ i_1\} \tag{4.5}$$

Recall that the last element, the root node hash, in both proofs is only used for verification.

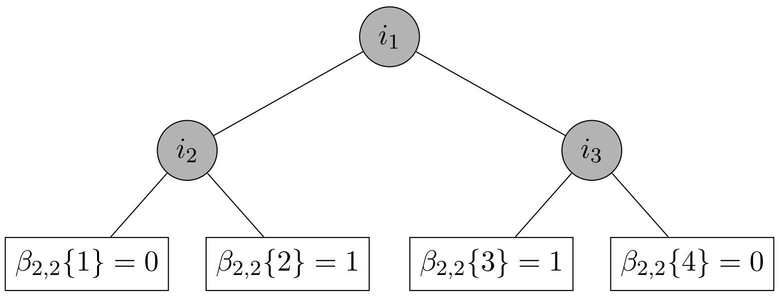

Figure 4.2: Authenticated Bloom filter $\beta_{2,2}$ with a configuration of 2^2 bits and 2 hash functions. Gray circles denote intermediate nodes and squares leaf nodes. Each leaf is annotated with its corresponding distinct bit in the filter.

The size of such a tree will be proportional to the size of the filter. Authenticating the filter would therefore require twice the amount of space. Such filter could instead be represented using a sparse Merkle tree in order to reduce space requirements at a cost of increased computation since the tree is simulated upon construction of an audit proof. Note that such tree is not necessarily sparse but rather of an intractable size.

4.4.1 Caching Strategy

The requirement to simulate the whole tree upon construction of every requested audit proof leads to redundant calculations on behalf of the logger since the hash of the intermediate nodes will be re-constructed on every invocation. Assuming the state of the filter has not changed since the last constructed audit proof. A caching strategy, as proposed by Dahlberg et al. [10], is utilized to cache the hash of intermediate nodes. Storing the hash of every intermediate node would require a cache with a capacity equal to the number of leafs in the tree which would defeat the purpose of the sparse tree representation. Therefore, the size of the cache is required to be strictly smaller than the number of leafs in the tree. Due to this constraint the cache has to be selective while caching the hash of intermediate nodes. Instead of caching an intermediate node with a fixed probability as proposed by Dahlberg et al. [10], nodes on a certain depth of the tree is only considered eligible for persistence in the cache. This non-probabilistic construction removes the need for a eviction policy as long as the cache has a capacity to store all nodes on the given depth. Meanwhile, it could lead to more computation in the worst case since fewer nodes will be eligible for caching. Thus, the effect of the chosen depth is of interest to analyze.

Further, let $\mathcal{C}_d$ denote a cache on depth d with a capacity of storing $|\mathcal{C}_d| = 2^d$ entries. The cache supports lookup $k \in \mathcal{C}_d$, read $\mathcal{C}_d\{k\}$ and write $\mathcal{C}_d\{k\} \leftarrow v$ operations given an arbitrary key k and value v. A key given to the cache represents an intermediate node expressed as a pair $(left, right)$ that covers the leafs in the tree located inclusively in between $left$ and $right$. In Figure 4.3, the keys for the intermediate nodes are illustrated. The current depth of a tree simulation for a filter $\beta_{m,k}$ can be expressed as $m - \lceil \log(right - left + 1) \rceil$. Thus, for a filter $\beta_{m,k}$ the key $(1, 2^m)$ would cover all leafs in the tree since its depth equals 0: $m - \lceil \log 2^m \rceil = 0$.

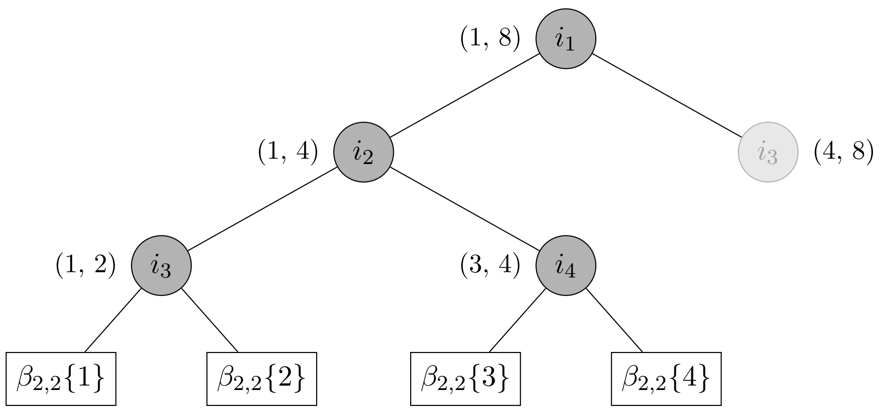

Figure 4.3: Simulated Merkle tree for an authenticated Bloom filter with $m = 3$. Each intermediate node is annotated with its corresponding key expressed as a pair which denotes the leafs the node covers beneath it. The right branch of tree is omitted.

Read Routine

Accepts a filter $\beta_{m,k}$, cache $\mathcal{C}_d$ and the two values $left$ and $right$ constituting the intermediate node. It rejects intermediate nodes on any depth other than d. Otherwise, the cached entry is returned.

Correctness

Every intermediate node on depth d expressed as a pair $(left, right)$ will cover exactly 2^{m-d} leafs beneath it. They can therefore be uniquely indexed on depth d: $\frac{left}{2^{m-d}}$, Since $left$ is bounded by the right-most pair $(2^m - 2^{m-d} + 1, 2^m)$ it follows that given every $1 \leq left \leq 2^m - 2^{m-d} + 1$ its corresponding index must be unique and collisions can not occur.

Running Time

Reading from the cache is considered as constant operation: $O(1)$.

```
CacheRead($\beta_{m,k}$, $\mathcal{C}_d$, $left$, $right$) =
    if $m - \lceil \log(right - left + 1) \rceil \neq d$ then
    |   return invalid depth
    end
    $i \leftarrow \frac{left}{2^{m-d}}$
    return $\mathcal{C}_d\{i\}$
```

Algorithm 1: Authenticated Bloom Filter cache read routine.

Write Routine

Accepts a filter $\beta_{m,k}$, cache $\mathcal{C}_d$, two values $left$ and $right$ constituting the intermediate node and its corresponding hash. It rejects intermediate nodes on any depth other than d. Otherwise, the hash is written to the cache.

Correctness

The same reasoning as for the read routine applies.

Running Time

Writing to the cache is considered as constant operation: $O(1)$.

```
CacheWrite($\beta_{m,k}$, $\mathcal{C}_d$, $left$, $right$, $hash$) =
    if $m - \lceil \log(right - left + 1) \rceil \neq d$ then
    |   return invalid depth
    end
    $i \leftarrow \frac{left}{2^{m-d}}$
    if $\mathcal{C}_d\{i\} \neq hash$ then
    |   $\mathcal{C}_d\{i\} \leftarrow hash$
    end
```

Algorithm 2: Authenticated Bloom Filter cache write routine.

4.4.2 Insertion

Accepts a filter $\beta_{m,k}$, cache $\mathcal{C}_d$ and the element x to insert. Insertion of an element x in an authenticated Bloom filter is done by passing it to all k hash functions and setting the corresponding bits to 1 in the filter. The cache needs to be informed about this change in order to invalidate its affected entries since some of them might cover the changed leaf.

Correctness

By removing the entries from the cache that covers the leaf given by $h_i(x)$ for $i \in [1, k]$ no stale entries will no longer be present causing the cache to behave correctly.

In Figure 4.2, assuming a $\mathcal{C}_1$ cache is used with the following entries present: $\mathcal{C}_1\{1\}$ and $\mathcal{C}_1\{2\}$. Passing x with $h_1(x) = 1$ and $h_2(x) = 2$ to Insert would only cause the $\mathcal{C}_1\{1\}$ entry to be removed from cache since it is the only entry that covers the first and second leaf.

Running Time

Since $h_i(x)$ for $i \in [1, k]$ is considered constant and k being small, insertion will be dominated by the cache invalidation. This operation is considered linear in the size of the cache: $O(|\mathcal{C}_d|)$.

```
Insert($\beta_{m,k}$, $\mathcal{C}_d$, $x$) =
    for $i \leftarrow 1$ to $k$ do
        $\beta_{m,k}\{h_i(x)\} \leftarrow 1$
        for $[left, right] \in \mathcal{C}_d$ where $left \leq h_i(x) \leq right$ do
            $i \leftarrow \frac{left}{2^{m-d}}$
            $\mathcal{C}\{i\} \leftarrow 0$
        end
    end
```

Algorithm 3: Authenticated Bloom Filter insertion routine.

4.4.3 Audit Proof Construction

Accepts a filter $\beta_{m,k}$, cache $\mathcal{C}_d$ and the element x to prove (non)-membership of. Constructing an audit proof for the element x is done by passing it to all k hash functions and constructing a proof for each corresponding bit which is represented as a leaf in the simulated tree. The AuditProof routine calls GenProof for each leaf which is responsible for simulating the tree.

Correctness

On the first invocation of GenProof $(left, right)$ will cover all leafs and on any subsequent recursive call the cover will be reduced in half. The algorithm will always favor the branch in which the requested leaf resides. This is of importance since the hash of the requested leaf node must always be the first element in the proof 4.3. By letting the requested leaf be the first reached leaf where the recursion terminates ensures this property holds. For any branch that does not cover the requested leaf, the algorithm is divided into two distinct scenarios: if $(left, right)$ is not present in the cache the sub-tree that constitutes this pair needs to be simulated resulting in continued recursion until the base-case is reached, that is when $left = right$. In the other scenario, $(left, right)$ is present in the cache causing the recursion to terminate.

Running Time

The worst-case occurs when the cache is empty causing every leaf to be reached in order to compute the hash of all intermediate nodes: $O(2^m)$.

In the optimal-case all intermediate nodes on depth d will be present in the cache. The only intermediate nodes that needs to be simulated are the 2^{d-1} number of intermediate nodes above the cache and the only leaf that needs to be reached is $leaf$ giving the lower bound: $\Omega(2^{d-1} + m)$.

In either scenario, reading and writing to the cache is considered a constant and negligible operation.

```
AuditProof($\beta_{m,k}$, $\mathcal{C}_d$, $x$) =
    $proofs \leftarrow \emptyset$
    for $i \leftarrow 1$ to $k$ do
        $proofs \leftarrow proofs \cup GenProof(\beta_{m,k}, \mathcal{C}_d, h_i(x), 1, m)$
    end
    return $proofs$

GenProof($\beta_{m,k}$, $\mathcal{C}_d$, $leaf$, $left$, $right$) =
    $mid \leftarrow left + \lfloor \frac{right-left}{2} \rfloor$
    if $left = right$ then
        return $h(\beta_{m,k}\{left\})$
    else if $leaf \in [left, mid]$ then
        $lhs \leftarrow GenProof(\beta_{m,k}, \mathcal{C}_d, leaf, left, mid)$
        $rhs \leftarrow GenProof(\beta_{m,k}, \mathcal{C}_d, leaf, mid+1, right)$
    else if $leaf \in [mid+1, right]$ then
        $rhs \leftarrow GenProof(\beta_{m,k}, \mathcal{C}_d, leaf, mid+1, right)$
        $lhs \leftarrow GenProof(\beta_{m,k}, \mathcal{C}_d, leaf, left, mid)$
    else if $[left, right] \in \mathcal{C}_d$ then
        return $CacheRead(\beta_{m,k}, \mathcal{C}_d, left, right)$
    else
        $lhs \leftarrow GenProof(\beta_{m,k}, \mathcal{C}_d, leaf, left, mid)$
        $rhs \leftarrow GenProof(\beta_{m,k}, \mathcal{C}_d, leaf, mid+1, right)$
    end
    $hash \leftarrow h(lhs \,||\, rhs)$
    $CacheWrite(\beta_{m,k}, \mathcal{C}_d, left, right, hash)$
    return $hash$
```

Algorithm 4: Authenticated Bloom Filter proof generation routines.

4.5 Logger Overview

The following data structures are maintained by logger in order to satisfy the requirements of the protocol.

A Merkle tree $\mathcal{T}$ where each log entry is stored in a separate leaf, thus log entry $\ell_i \in L$ is stored in leaf $l_i \in \mathcal{T}$. This allows construction of membership proofs for a given log entry.

Each leaf $l_i \in \mathcal{T}$ stores another nested Merkle tree $\mathcal{T}_i^{'}$. Each leaf in this nested tree represents an attribute tuple $(a, v) \in \ell_i$ where $h(a \,||\, v)$ is used as its hash in the nested Merkle tree. The root node hash for the nested Merkle tree $\mathcal{T}_i^{'}$ will be used as the hash for the corresponding leaf $l_i \in \mathcal{T}$.

When an application sends a log entry ℓ to the logger, the log entry is extended by inserting a random value τ_j where $j \in [1, n]$ after each existing attribute tuple:

$$\ell' = \{(a_1, v_1), \tau_1, (a_2, v_2), \tau_2, \ldots, (a_n, v_n), \tau_n\} \tag{4.6}$$

$$|\ell^{'}| = 2n \tag{4.7}$$

This will ensure that while constructing an audit proof for a nested tree $\mathcal{T}^{'}$ it will not leak the hash representation of another neighbouring attribute, as proposed by Buldas et al. [7]. Despite which attribute to prove membership of such proof will only consist of the hash of the requested attribute, the hash of a random value and the other intermediate nodes that is necessary to construct the proof.

For each attribute tuple $(a_i,\, v_i)$ where $i \in [1, n]$ and $(a_i,\, v_i) \in \ell$, an authenticated Bloom filter A_{a_i} is created if it does not already exist. The value v_i is then inserted into the filter. Different log entries where a_i is present therefore shares the same authenticated Bloom filter, thus A_{a_i} represents the set of seen values for attribute a_i. Since a separate filter is kept for each seen attribute the number of attributes does not need to be known a priori since filters can be created once a new attribute is received. The configuration of the underlying filter $\beta_{m_{a_i},k}$ will have an attribute dependent number of bits $2^{m_{a_i}}$ based on the estimated cardinality of the set of values for attribute a_i. The formulas 2.2 developed by Bloom [13] can be used with the caveat that the estimate needs to be rounded up to the nearest power of 2 in order to keep the tree balanced.

Insertion of a leaf to the tree $\mathcal{T}$ and insertion of values to authenticated Bloom filters are constant operations. The running time of the insertion performed by the logger is dominated by the construction of the nested tree $\mathcal{T}'$ which is linear in the number of leafs: $O(|\ell'|) = O(n)$.

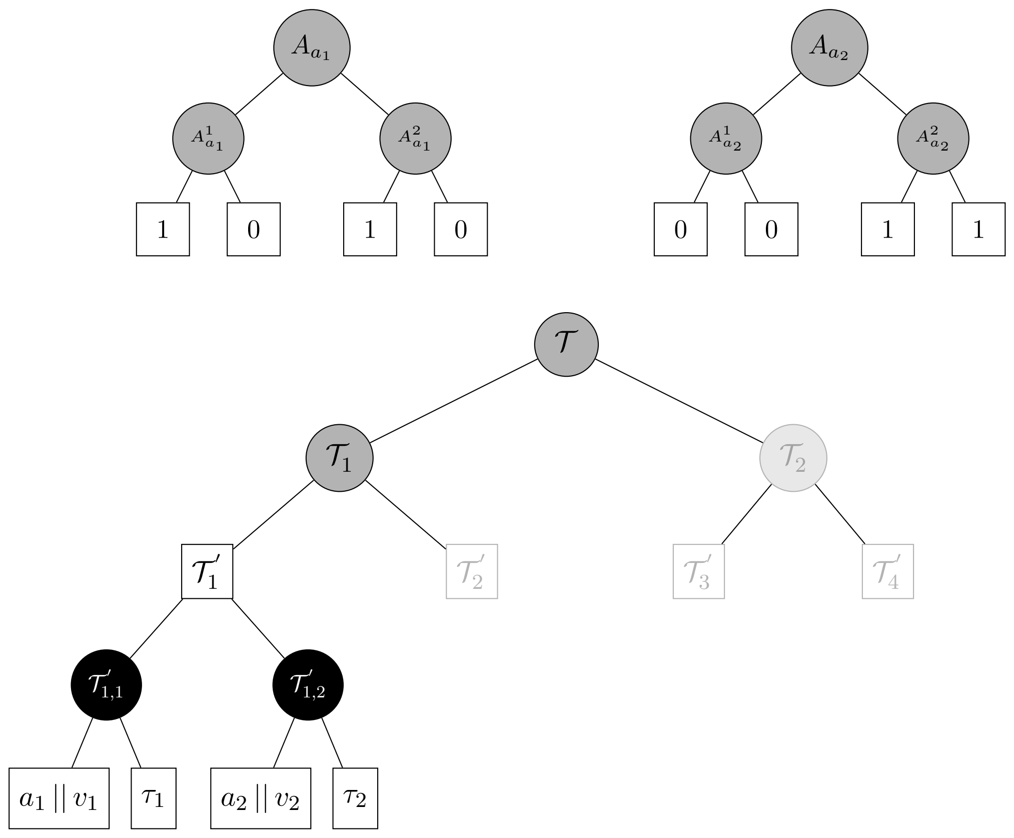

Figure 4.4: Merkle tree $\mathcal{T}$ for the corresponding log $L = \{\ell_1,\, \ell_2,\, \ell_3,\, \ell_4\}$. Each log entry $\ell_i \in L$ for $i \in [1,\, 4]$ stores its attribute tuples in a nested Merkle tree $\mathcal{T}_i'$. Circles denote intermediate nodes and squares leaf nodes. The tree $\mathcal{T}_1'$ for log entry $\ell_1 = \{(a_1,\, v_1),\, (a_2,\, v_2)\}$ and the corresponding authenticated Bloom filters A_{a_1} and A_{a_2} are illustrated. The nested trees for $\mathcal{T}_2'$, $\mathcal{T}_3'$ and $\mathcal{T}_4'$ are omitted.

4.6 Membership

The system must be capable of answering to (non)-membership queries expressed as follows: given an attribute x and a corresponding value y, forming the attribute tuple (x, y), does one or many log entries $\ell \in L$ exists where $(x, y) \in \ell$. The validity of the answer to such queries is proven by verifying the corresponding audit proof. The verification process remains the same regardless of where the proof originates from which could be either from an authenticated Bloom filter, nested Merkle tree $\mathcal{T}'$ or the Merkle tree $\mathcal{T}$.

4.6.1 Audit Proof Verification

Accepts a proof P. The first element $P[1]$ is the hash of the requested leaf node and the last element $P[n]$ the hash of the root node used for verification. The routine iteratively calculates the hash which reassembles the same calculation as performed by the tree where the proof originates from.

Correctness

Any proof that does not result in the same advertised root node hash $P[n]$ will be rejected. It follows from the fact that the hash function $h(x)$ is known by both the prover and verifier.

Running Time

Linear in the size of the proof: $O(|P|) = O(n)$.

```
VerifyProof($P$) =
    $prev \leftarrow P[1]$
    $root \leftarrow P[n]$
    for $i \leftarrow 2$ to $n - 1$ do
        $hash \leftarrow h(prev \,||\, P[i])$
        $prev \leftarrow hash$
    end
    if $root = hash$ then
        return $true$
    else
        return $false$
    end
```

Algorithm 5: Audit proof verifier routine.

4.6.2 Proving Membership

Proving membership of a given attribute tuple (x, y) which is present requires construction of two audit proofs:

1. Locate the leaf $l \in \mathcal{T}$ associated with log entry $\ell \in L$ where $(x, y) \in \ell$. In order to prove that $(x, y) \in \ell$ an audit proof for (x, y) in the corresponding nested tree $\mathcal{T}'$ rooted in leaf l is constructed. This will prove that the given attribute tuple is a

partial causation of the root hash for $\mathcal{T}'$ which implies $(x,\, y) \in \ell$. Such proof is of length $\lambda\lceil\log 2n\rceil$.

Note that locating the leaf $l \in \mathcal{T}$ is intentionally unspecified since it is an implementation detail rather than a concern of the protocol. A method with constant running time is recommended.

2. Construct an audit proof for $l \in \mathcal{T}$. This will prove that the leaf l associated with log entry ℓ where $(x,\, y) \in \ell$ is a partial causation of the root hash for $\mathcal{T}$ and hence present in L. Such proof is of length $\lambda\lceil\log|L|\rceil$.

In Figure 4.4, the membership proof for the attribute tuple (a_1, v_1) is constituted by the following three audit proofs, assuming $\beta_{2,1}$ is used as the underlying filter in A_{a_1} and $h_1(v_1) = 1$:

$$P_{A_{a_1}} = \{h(\beta_{2,1}\{1\}),\, h(\beta_{2,1}\{2\}),\, A^2_{a_1},\, A_{a_1}\} \tag{4.8}$$

$$P_{\mathcal{T}'_1} = \{h(a_1 \,||\, v_1),\, h(\tau_1),\, \mathcal{T}'_{1,2},\, \mathcal{T}'_1\} \tag{4.9}$$

$$P_{\mathcal{T}} = \{\mathcal{T}'_1,\, \mathcal{T}'_2,\, \mathcal{T}_2,\, \mathcal{T}\} \tag{4.10}$$

Correctness

Since every log entry is extended by inserting random values in-between each pair of attribute tuples, all attributes in the corresponding nested Merkle tree will have such a random value as its neighbouring leaf. This will guarantee that a proof rooted in a nested Merkle tree will never contain a hash representation of another attribute. Instead, only hash representations of the node to audit, random values and intermediate nodes will be present.

Running Time

The running time will be dominated by the audit proof construction for the authenticated Bloom filter. The worst case occurs when the cache is empty: $O(k2^{m_x})$. The lower bound is given when the cache is fully utilized: $\Omega(k(2^{d-1} + m_x))$.

4.6.3 Proving Non-Membership

Proving non-membership of a given attribute tuple $(x,\, y')$ requires construction of less proofs compared to proving membership since the absence of the requested attribute tuple can be proven at an earlier stage.

Lookup y' in corresponding authenticated Bloom filter A_x, with the underlying filter $\beta_{m_x,k}$ and cache $\mathcal{C}_d$, by passing y' to each of the k number of hash functions. Absence of y' implies that $\beta_{m_x,k}\{h_i(y')\} = 0$ for any $i \in [1,\, k]$ but this is not necessarily true due to Bloom filters probabilistic construction. The process is divided into two distinct scenarios:

1. If $\beta_{m_x,k}\{h_i(y')\} = 0$ for any $i \in [1,\, k]$ holds true it implies that all log entries $\ell \in L$ satisfies $(x,\, y') \notin \ell$. An audit proof for each corresponding leaf in A_x given a bit is constructed. Such proof is of length $k\lambda m_x$.

2. If $\beta_{m_x,k}\{h_i(y')\} = 1$ for all $i \in [1, k]$ holds true it implies that there exists a subset of log entries $L' \subseteq L$ where each log entry $\ell \in L'$ includes $(x, z) \in \ell$ thus $z \neq y'$. The bits given by the k number of hash functions given those values will include all bits given by passing y' to the same hash functions:

$$H_1 = \{h_1(z), h_2(z), \dots, h_k(z) \mid (x, z) \in \ell \wedge \ell \in L'\} \tag{4.11}$$
$$H_2 = \{h_1(y'), h_2(y'), \dots, h_k(y')\} \tag{4.12}$$
$$H_1 \cap H_2 = H_2 \tag{4.13}$$

Meaning, there exists other values for the attribute x which caused all the bits for the requested value y' to be set. In order to prove that the found log entries $\ell \in L'$ are all false positives the corresponding leaf $l \in \mathcal{T}$ associated with ℓ where $(x, z) \in \ell$ is located. By constructing an audit proof for the nested tree $\mathcal{T}'$ rooted in leaf l its correctness can be verified only to reveal that the audit proof always will be false since the first entry in the proof will consist of the hash representation of $x \,||\, z$ rather than the requested $x \,||\, y'$, thus $h(x \,||\, z) \neq h(x \,||y')$ since h is considered secure. Note that this only proves the existence of a collision in the authenticated Bloom filter.

In Figure 4.4, the non-membership proof for the attribute tuple (a_1, v_3) is constituted by the following two audit proofs. Assuming $\beta_{2,1}$ is used as the underlying filter in A_{a_1} and $h_1(v_1) = h(v_3) = 1$. Note that the first element in $P_{\mathcal{T}_1'}$ is expected to equal $h(a_1 \,||\, v_3)$ as opposed of $h(a_1 \,||\, v_1)$.

$$P_{A_{a_1}} = \{h(\beta_{2,1}\{1\}),\ h(\beta_{2,1}\{2\}),\ A^2_{a_1},\ A_{a_1}\} \tag{4.14}$$
$$P_{\mathcal{T}_1'} = \{h(a_1 \,||\, v_1),\ h(\tau_1),\ \mathcal{T}_{1,2}',\ \mathcal{T}_1'\} \tag{4.15}$$

Correctness

In the first scenario, the correctness is reduced to the correctness of audit proof construction for the authenticated Bloom filter 4.4.3. In the second scenario, the correctness is reduced to the hash function h resistance against collisions.

Running Time

The running time will be dominated by the audit proof construction for the authenticated Bloom filter. The worst case occurs when the cache is empty: $O(k2^{m_x})$. The lower bound is given when the cache is fully utilized: $\Omega(k(2^{d-1} + m_x))$.

4.7 Analysis

In this section, the security of the proposed protocol is analysed in terms of what kind of protection it offers and how well it prevents potential attacks.

4.7.1 Completeness

It shall be clear from the protocol definition that an honest logger always will be able to convince an honest application or auditor regarding the validity of the constructed membership proof for the requested attribute tuple.

For non-membership proofs, an honest logger can only convince an honest application or auditor regarding the fact that a collision is encountered.

4.7.2 Soundness

If a bit in any of the underlying data structures is corrupted either by an attacker or due to hardware failure an application or auditor requesting an audit proof accepts it with a negligible probability.

If the data stored in the leafs in either the Merkle tree $\mathcal{T}$ or the nested trees $\mathcal{T}^{'}$ is corrupted or tampered with all future proofs will be invalidated. Since the hash values of all nodes in the Merkle trees are calculated once upon insertion and not on request, modifying the data in a leaf will affect the hash values for all intermediate nodes descending to the particular leaf. The security is therefore reduced to the used hash function which is considered secure. Modifying the data stored in the leafs requires the ability re-calculate the hash values for the affected nodes in order to convince the auditor into accepting the proof. This would require direct access to the logger and any further analysis of this threat is not a concern of the protocol.

If a bit in any of the authenticated Bloom filters transitions from 0 to 1 that is not caused by an insertion, it will result in potentially more false-positives. The protocol is by design already capable of handling such scenarios.

In the opposite scenario, a bit transitions from 1 to 0. This is an invalid transition according to the definition of Bloom filters and thus authenticated Bloom filters. However, such transition could be made possible due to vulnerabilities in the logger. If such transition remains undetected it will cause the logger to construct invalid (non)-membership proofs since it no longer maintain a correct authenticated Bloom filter which represents the set of seen values for a specific attribute. In order to reduce the risk of such transition to go by undetected an extra security countermeasure is proposed during insertion of new log entries. Since the configuration (m and k in $\mathcal{B}_{m,k}$) of the underlying filter used by all authenticated Bloom filters is known by all applications and the logger, an application can prior sending the log entry to the logger determine which bits in each filter that should be set 1 based on the attribute tuples present in the log entry. By asking the logger for an audit proof for each attribute tuple prior sending the log entry an application can then issue the same audit proof requests again in order to make sure that all bits only transitioned from 0 to 1, assuming the bit was set to 0 prior sending the log entry. Any bits already set to 1 should remain in this state. Additional integrity could be achieved by letting a trusted party, such as the application, sign the root hash of the authenticated Bloom filter after each insertion. The signatures would form a chain of trust which could be verified.

4.7.3 Privacy-Preserving

The protocol leaks the following information:

1. The lower bound of the height of the Merkle tree $\mathcal{T}$ and any nested Merkle tree $\mathcal{T}^{'}$ is leaked through the proofs. An audit proof for a nested Merkle tree $\mathcal{T}^{'}$ reveals the lower bound of the number of attribute tuples present in the corresponding log entry the tree is representing since the height of the tree is proportional to the number

of leafs. Since such tree not necessarily is balanced only the lower bound can be deduced. This is also true for the tree $\mathcal{T}$, meaning that the lower bound of number of log entries in the complete log L can be estimated from any audit proof that originated from this tree.

2. Proving non-membership in the case of a false-positive reveals the hashed representation of another attribute tuple for the same attribute but with another value. An attacker with knowledge of the name of at least one attribute could perform a brute-force attack asking for membership proofs for the known attribute a using fictitious values. Once such a value results in a false-positive the hashed representation of the known attribute and another value (unknown to the attacker) is leaked. Recall, the hash is calculated as $h(a \,||\, v)$ and since the attacker already knows a the space needed to be exhaustively examined in order to invert the hash is reduced. If the attacker does not know the name of any attribute it would be forced to find one by constructing both fictitious attribute names and values for use in the following step of asking for membership proofs.

The choice of sufficiently large security parameters λ and Λ is of importance in order to prevent brute-force attacks against the protocol. Since λ dictates the number of bits returned by the hash function a lower value will have a direct impact on the space of possible values an attacker would have to exhaustively examine. This also applies to the choice of Λ where a low value would limit the number of possible random values making it more computability feasible to invert the hash.

Chapter 5

Methods

An implementation of the protocol has been developed and evaluated. In this chapter, the details regarding the implementation and the conducted experiments and evaluations are described.

5.1 Implementation

The implementation will be evaluated by simulating the canonical application of the protocol: a banking institute using the protocol to audit its transactions. All three actors are present in this simulation: the applications carry out transactions and send transaction receipts to the logger for later auditing. Such receipt is therefore equivalent to a log entry expressed in terms of the formal protocol definition. The log entry is sent in its JSON-representation according to the following key-value schema, each key-value pair forms an attribute tuple where the key is the attribute name and value the corresponding attribute value. The log entries are randomized but is expected to include following attributes:

```
{
  "ssn": "1996-10-01-2857",
  "account": 200794,
  "bank": "Svenske Bank",
  "currency": "SEK",
  "amount": 1024,
  "session": 162395,
  "signature": "ttrllkywddefgqfjgwdxaefriejruilpdecpobmsfryli",
  "timestamp": 1492601206
}
```

Auditors are responsible for verifying that a certain transaction has or has not been carried out. Their knowledge of the log entry schema is therefore limited since they are only concerned about one or many of the attributes present in each log entry.

The Merkle tree $\mathcal{T}$ will be pre-allocated to a size sufficient to hold all log entries. The hash values for the intermediate nodes in this tree will be calculated upon insertion. Meaning that no hash values needs to be calculated during construction of audit proofs originating from this tree. This ensures fair performance despite which log entry to prove presence of in the tree.

The Merkle tree and authenticated Bloom filters are implemented in C and the simulation in C++. The hash function used by the Merkle trees is the SHA-$\{256, 512\}$ implementation from OpenSSL, which also dictates the choice of $\lambda \in \{256, 512\}$. For the authenticated Bloom filters the Murmur reference implementation [2] is used since it does not require usage of a secure hash function. The random values are generated using the default `rand(3)` implementation on the host Operation System with $\Lambda = 32$. The GCC (version 4.8.3.3) compiler was used with optimizations (`-O2`) enabled.

All experiments and evaluations where performed on a Amazon EC2 virtual machine with the following specification:

Model	m4.2xlarge
CPU	2.3 GHz Intel Xeon(R)
Memory	32 GiB
OS	Amazon Linux AMI 2017.03 (Red Hat Enterprise Linux derivative)

5.2 Evaluation

The following tests where conducted in order to evaluate the performance of the implementation:

1. Time taken to insert a new log entry using both SHA-$\{256, 512\}$ as the hash function used by the Merkle trees.

2. Optimal cache depth for an authenticated Bloom filters of varying size. An audit proof originating from an authenticated Bloom filter consists of k separate proofs. One for each leaf given by the k the number of hash functions. In this evaluation the time needed to construct a proof for a single leaf is measured and the number of used hash functions is of less importance. The filter is populated with random members of varying length between 1-128 characters. In order to simulate the average case, the element to prove (non)-membership is randomized and the median from 1000 membership-proofs is recorded for each evaluated depth.

3. Time taken to construct an audit proof for an authenticated Bloom filter with a optimal cache configuration based on the results from the previous evaluation. This evaluation allows the trade-off in terms of space required by the cache and the reduction of proof construction time to be analyzed.

4. Estimation of the size of an authenticated Bloom filter. Using the formulas 2.2 proposed by Bloom [13], the size of the filter is estimated using $p = 0.01$ as the desired false positive probability and $n = 1000$. The entries inserted into filter are 1000 random entries from the standard dictionary (`/usr/share/dict/words`) present on many *NIX-platforms. The estimated m is rounded off to the nearest power of 2 in order to keep the authenticated Bloom filter balanced, giving the configuration $\beta_{14,7}$. Since this is an estimation surrounding values are considered in order to verify the reliability of the estimation in the context of authenticated Bloom filters. Any adjacent configuration giving a lower false positive rate than the pre-defined p is also of interest. The false positive rate is measured as the number of entries that during insertion did not cause any bits in the filter to be set. Implying a collision is detected.

Chapter 6

Results

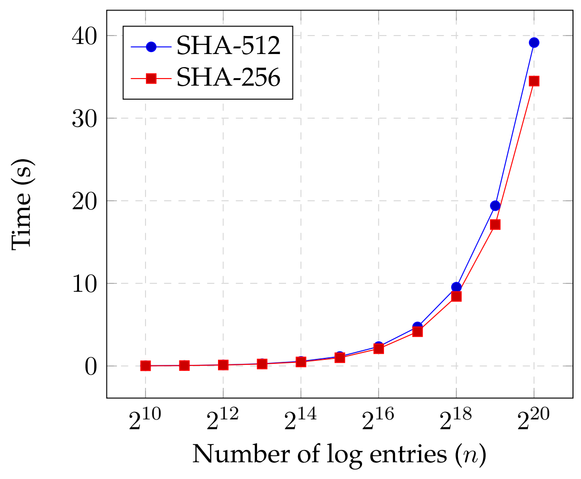

Figure 6.1: Insertion time.

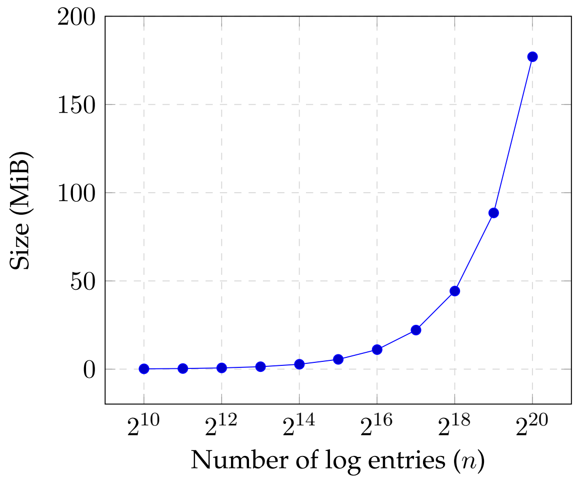

Figure 6.2: Size of the inserted log entries.

In Figure 6.1 the results of evaluation 1 is shown. Inserting twice as many log entries increases the time taken with the same factor. The corresponding size of the input is shown in Figure 6.2. Using SHA-512 is slower than its predecessor SHA-256 which becomes noticeable as the number of log entries increases.

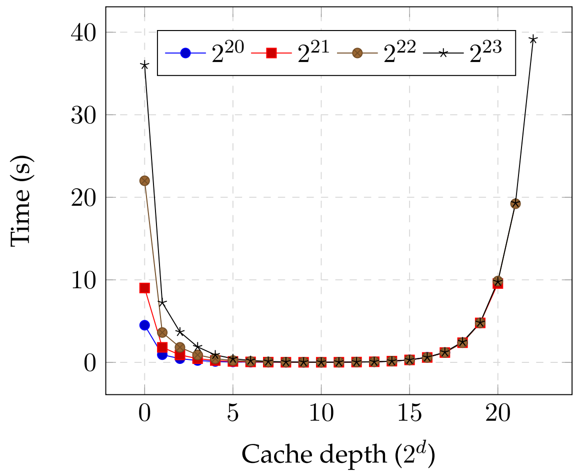

Figure 6.3: Authenticated Bloom filter audit proof construction time using a cache on increasing depth.

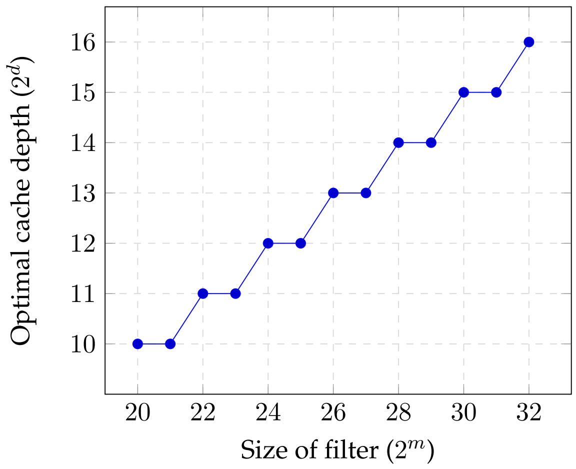

Figure 6.4: Optimal cache depth for Authenticated Bloom filters of increasing size okdwo kdwok dwok dokw.

In Figure 6.3 the results of evaluation 2 is shown: time taken to construct an audit proof for an authenticated Bloom filter using a cache on an increasing depth. In order to not clutter the graph only the filters where $m \in [20, 23]$ are included but the rest where $m > 23$ concludes the same behavior, as shown in Figure 6.4. The first data point denotes the time taken with caching completely disabled. Caching on a depth lower than m reduces the running time. The increase in time when d gets closer to m is caused by the overhead involved with updating a cache that is marginally utilized.

In Figure 6.4 the optimal cache depth as a function of the authenticated Bloom filter size is shown. Note how the optimal depth can be expressed as $2^{m/2} \leftrightarrow \sqrt{2^m}$.

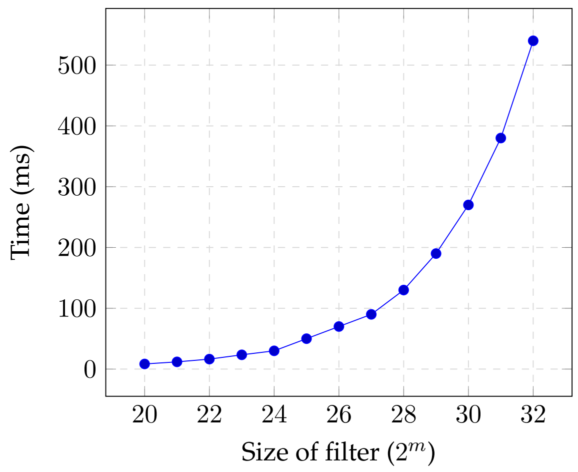

Figure 6.5: Authenticated Bloom filter audit proof construction time with optimal caching.

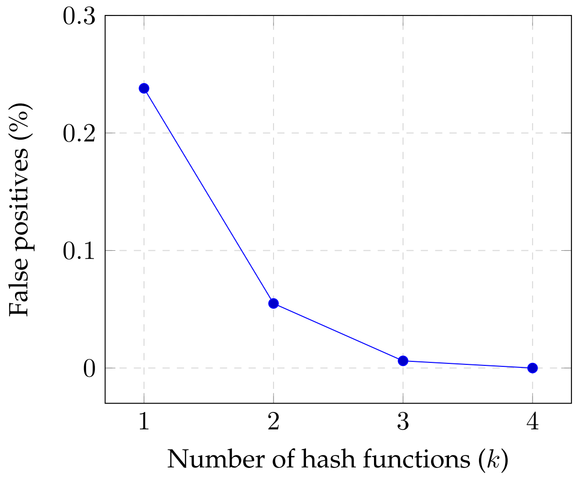

Figure 6.6: False positives as a function of the number of used hash functions (k).

In Figure 6.5 the results from evaluation 3 is shown. Each filter has an optimal cache configuration $\sqrt{2^m}$. The table in Figure 6.7 shows the running time with no caching (t_0), with optimal caching (t_{opt}) and their delta ($t_{opt} - t_0$) for $m \in [20, 23]$. All values are expressed in seconds.

m	t_0	t_{opt}	Δ
20	4.501	0.008	−4.493
21	9.011	0.012	−8.999
22	22.002	0.016	−21.986
23	36.038	0.023	−36.014

Figure 6.7: Running time improvement using optimal caching for the authenticated Bloom filters.

In Figure 6.6, evaluation 4 is shown. The underlying filter uses $m = 14$ while k varies. Configurations using $m < 14$ was evaluated but discarded since $m = 14$ was the first size causing a 0% false positive rate. Note that no false positives where reached at $k = 4$ which is lower than the estimated $k = 7$.

Chapter 7

Conclusions

7.1 Discussion

The insertion performance 6.1 of the implementation peeks at processing $30\,000$ log entries per second. Profiling shows that $1/4$ of the time is spent parsing log entries. Since the implementation only parses a single log entry at a time the performance could be improved by processing the incoming log entries in batches prior doing the insertion.

The empirical result of the optimal cache depth 6.4 shows that for a Bloom filter of size 2^m, a cache of size $\sqrt{2^m}$ yields the greatest performance for audit proof construction. While this being an interesting result, further testing and theoretical reasoning is required to conclude whether this is true for the average case in general.

Since the size of the authenticated Bloom filters is a power of 2 the results 6.6 indicate that the formula for m can be used if the result is rounded of upwards. Over-estimating m could prevent false positives which is desirable. The formula for k can be relaxed which in practice could be convenient under the assumption that finding suitable hash functions is harder than trading space.

7.2 Ethical Aspect

Since one of the initial requirements of the proposed protocol is to certify history and prevent tampering, it does neither allow nor encourage morally questionable actions such as rewriting the past. The logger will always present the log entries in their unaltered form and if a vulnerability in the logger where to be exploited several mechanisms are incorporated in the protocol to ensure malicious actions does not go by unnoticed.

The canonical application for the protocol was made with banking institutes in mind allowing the transactions carried out to be monitored by the auditors. In this application trust between the involved parties cannot be established and the importance of auditing is therefore severe. This particular application can further be generalized into the act of separating duties. The protocol is modelled based on the idea of having parties carrying out tasks that are logged to a centralized logger for auditing. This allow the carried out tasks to be audited by another party which is especially valuable if it is believed that the tasks are not carried out correctly under all circumstances. With this generalization in mind, the protocol has several potential applications: the log could reflect changes made to legal documents where several parties are involved allowing the revision history of such documents to be audited in order to prevent modifications that are not approved by all parties to go by unnoticed. It could also be used to track modifications of source code and deployments of production critical software. The protocol would in this application prevent non-authorized modifications and injection of malicious code to be detected by the auditors. In general, whenever a process involving several parties and complete trust between them cannot be established this protocol could help by making the process available for auditing. Since it is often desirable to prevent access to certain sensitive data between the parties the protocol has method of satisfying such conditions.

7.3 Conclusions

In order to prove membership, each log entry is inserted into a Merkle tree where each leaf is constituted by another nested Merkle tree. The nested Merkle tree stores each attribute tuple present in the corresponding log entry in a separate leaf. Great care has been taken while constructing the nested Merkle trees by ensuring that each attribute tuple is paired with a random value. This will ensure that no additional knowledge about other attribute tuples in the log entry is leaked through an audit proof. The choice of security parameters λ and Λ is of importance to prevent brute-force attacks in this scenario.

For non-membership, a method used to authenticate Bloom filters was proposed. The proposal allows the state of any bit in the filter to be proven by letting each bit represent a leaf in a simulated Merkle tree. Since a Bloom filter represents a set only an estimate of the upper bound of the cardinality of the set is required, as opposed of requiring the tree to be proportional to the cardinality of the input domain. This property is especially useful when working with sets consisting of non-numeric members of varying length. Empirical results 6.5 show that using optimal caching yields a significant improvement on the time taken to construct an audit proof. The proposed caching strategy does not require an eviction policy which allows for a simpler implementation. Due to the Bloom filters probabilistic construction a method of handling false positives is required which is incorporated in the proposed protocol.

The success of the protocol is reduced to its ability to fulfill the requirements established in the research question 1.1:

1. Efficient construction of (non)-membership proofs

   Since the hash values for all intermediate nodes are calculated upon insertion in the Merkle trees the time needed to construct membership proofs is proportional to the height of the tree which yields a scalable and efficient running time. The most costly operation is therefore dominated by the authenticated Bloom filters which in the average case using optimal caching significantly improves the running time. The running time is of the same order of magnitude compared to the sparse Merkle tree implementation by Dahlberg et al. [10] and is therefore considered efficient.

   In disregard of the space needed to store the Merkle trees the overhead in terms of space is dominated by the cache associated with the authenticated Bloom filters. Since this cache is significantly smaller than the underlying Bloom filter, a filter of size 2^m requires a corresponding cache of size $\sqrt{2^m}$, it is also considered efficient.

2. Preserve privacy

   Complete preservation of privacy was as expected not achieved. A lower bound of the number of leafs present in the Merkle tree that an audit proof originates from is leaked and proving non-membership in the case of a false-positive reveals the hashed representation of another attribute tuple. However, the protocol does not leak any data in its plain representation and the security is therefore reduced to the choice of a secure hash function and its corresponding security parameter λ.

3. Certify history

   The protocol incorporates a method of allowing tampering, deletion and corruption to be detected by an auditor.

7.4 Future Work

It is of interest to experiment with caching on several depths for the authenticated Bloom filters. Although the results indicate where the optimal depth resides, adding another cache on depth 1 with a capacity of storing 2 entries despite the size of the filter could impact the running time in the average case. Since the hash functions used by the filter distribute its input uniformly, the probability for each leaf to end up being the one to construct an audit proof for is also uniform: $\frac{1}{2^m}$, where m being the size of the filter. Implying that in the average case at each intermediate node the probability of choosing the left or right branch in the direction of the leaf is evenly divided: $\frac{1}{2}$. Thus, the probability of going down a specific path leading to a leaf is also uniform: $(\frac{1}{2})^{\log 2^m} = \frac{1}{2^m}$. Starting at the root node on depth 0, one of the two branches will not lead to the leaf and if a cache where to be present on depth 1 the unexamined branch would not be required to be simulated assuming it is already present in the cache. Since such a cache only requires 2 entries despite the size of filter it would be interesting to examine if it causes a noticeable improvement in running time.

The performance impact of adding TLS to the communication between the logger and the other parties as defined by the protocol is of interest to further analyze.

Bibliography

[1] Stefan Richter, Victor Alvarez, and Jens Dittrich. A seven-dimensional analysis of hashing methods and its implications on query processing. *Proceedings of the VLDB Endowment*, 9(3):96–107, 2015.

[2] Austin Appleby. Murmurhash3 64-bit finalizer. Technical report, Version 19/02/15. https://code.google.com/p/smhasher/wiki/MurmurHash3.

[3] Georg Becker. Merkle signature schemes, merkle trees and their cryptanalysis. Technical report, 2008.

[4] Oded Goldreich. *Foundations of cryptography: volume 1, basic tools*. Cambridge university press, 2001.

[5] Benjamin Dowling, Felix Günther, Udyani Herath, and Douglas Stebila. Secure logging schemes and certificate transparency. In *European Symposium on Research in Computer Security*, pages 140–158. Springer, 2016.

[6] Ralph Charles Merkle. Secrecy, authentication, and public key systems. Technical report, Citeseer, 1979.

[7] Ahto Buldas, Ahto Truu, Risto Laanoja, and Rainer Gerhards. Efficient record-level keyless signatures for audit logs. In *Nordic Conference on Secure IT Systems*, pages 149–164. Springer, 2014.

[8] Michael T Goodrich and Roberto Tamassia. Efficient authenticated dictionaries with skip lists and commutative hashing. Johns Hopkins Information Security Institute, 2000.

[9] Rafail Ostrovsky, Charles Rackoff, and Adam Smith. Efficient consistency proofs for generalized queries on a committed database. In *International Colloquium on Automata, Languages, and Programming*, pages 1041–1053. Springer, 2004.

[10] Rasmus Dahlberg, Tobias Pulls, and Roel Peeters. Efficient sparse merkle trees. In *Nordic Conference on Secure IT Systems*, pages 199–215. Springer, 2016.

[11] Ben Laurie and Emilia Kasper. Revocation transparency. *Google Research, September*, 2012.

[12] Scott A Crosby and Dan S Wallach. Authenticated dictionaries: Real-world costs and trade-offs. *ACM Transactions on Information and System Security (TISSEC)*, 14(2):17, 2011.

[13] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[14] Eu-Jin Goh et al. Secure indexes. *IACR Cryptology ePrint Archive*, 2003:216, 2003.

[15] Jin Li, Qian Wang, Cong Wang, Ning Cao, Kui Ren, and Wenjing Lou. Fuzzy keyword search over encrypted data in cloud computing. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–5. IEEE, 2010.

[16] Dawn Xiaoding Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *Security and Privacy, 2000. S&P 2000. Proceedings. 2000 IEEE Symposium on*, pages 44–55. IEEE, 2000.

[17] Parimala Raghavan, Sakthivel Subasree, and Sakthivel. Cluster based public auditing for shared data with efficient group user revocation in the cloud. *IIOAB JOURNAL*, 7(9):503–508, 2016.

[18] Ning Cao, Cong Wang, Ming Li, Kui Ren, and Wenjing Lou. Privacy-preserving multi-keyword ranked search over encrypted cloud data. *IEEE Transactions on parallel and distributed systems*, 25(1):222–233, 2014.

[19] Adam J Lee, Parisa Tabriz, and Nikita Borisov. A privacy-preserving interdomain audit framework. In *Proceedings of the 5th ACM workshop on Privacy in electronic society*, pages 99–108. ACM, 2006.

[20] Ben Laurie. Certificate transparency. *Queue*, 12(8):10, 2014.

[21] Ben Laurie, Adam Langley, and Emilia Kasper. Certificate transparency. Technical report, 2013.

[22] Scott A Crosby and Dan S Wallach. Efficient data structures for tamper-evident logging. In *USENIX Security Symposium*, pages 317–334, 2009.

[23] Roberto Tamassia. Authenticated data structures. In *European Symposium on Algorithms*, pages 2–5. Springer, 2003.

[24] Silvio Micali, Michael Rabin, and Joe Kilian. Zero-knowledge sets. In *Foundations of Computer Science, 2003. Proceedings. 44th Annual IEEE Symposium on*, pages 80–91. IEEE, 2003.

[25] Esha Ghosh, Olga Ohrimenko, and Roberto Tamassia. Efficient verifiable range and closest point queries in zero-knowledge. *Proceedings on Privacy Enhancing Technologies*, 2016(4):373–388, 2016.