Advanced Transparency System

Yuxuan Sun
Imperial College London
ys2924@ic.ac.uk

Yuncong Hu
Shanghai Jiao Tong University
huyuncong@sjtu.edu.cn

Yu Yu
Shanghai Jiao Tong University
yuyu@cs.sjtu.edu.cn

Abstract—In contemporary times, there are many situations when users need to verify that their information is correctly reserved by servers. At the same time, the servers need to maintain the transparency logs. Many algorithms are designed to solve this problem. For example, Certificate Transparency (CT) is designed to help reserving certificates issued by Certificate Authority (CA), CONIKS is designed to bring key transparency to end users, etc. However, they either suffer high append time or imbalanced Inclusion-proof cost and Consistency-proof cost. To find an optimal solution, we constructed two different but similar authenticated data structures to deal with two different lookup protocols. We propose ATS (Advanced Transparency System) to use only linear storage cost to reduce append time and balance the time cost of server's and users'. When solving the value-lookup problem, this system allows servers can append users' information with only constant-level time and realize radical-level Inclusion proof and Consistency proof. When solving the key transparency problem, this system needs logarithmic-level time complexity to perform the append operation and acceptable Inclusion proof and Consistency proof.

Index Terms—key management, transparency log, authenticated data structure, merkle tree, chunking

I. Introduction

Nowadays, people show more and more interest in safely preserving information. [1], [2], [3], [4], [5], [7], In practical terms, whether it is the storing of website certificates [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [19] the end-user key verification service [23], [22], [6], [18], [19], [24], [25], [26], [27] or other situations [28] it is necessary to maintain the user's public key in a reasonable way to ensure that its content is safe and secure. Clients will conduct some operations to add information to a public key/certificate-preserving system, as well as checking if their information is in the server. The data management tool in these situations is called transparency logs. To make sure the information provided by users are safely preserved, the transparency logs should satisfy the characteristics below:

Transparency logs should remain consistent. Neither clients nor servers have the access to delete or modify the previous information in the log. Some malicious servers can change the information in the transparency log. So, there must be someone to monitor all the data to make sure the server's behavior is appropriate. However, most clients don't have the time to do the monitoring and it doesn't need so many monitorings. In this way, introducing some auditors into this system can help. A client can also be a auditor volunteerly. This protocol can convince that the server processes information legally.

User infomation shouldn't be maintained separately. [20] For example, in Certificate Transparency [8], [9], [10], [11], if having no precautions taken, malicious servers may build several Merkle Trees with different user information preserved in different trees. In this way, even if each Merkle Tree remains consistent, clients will still fail to find all users' information since they don't have access to other Merkle Trees. To deal with this problem, users can gossip with each other to make sure that each user's digest remains the same.

There are two kind of requirements for lookup proof. Some requires key-value lookup proof, others only need to lookup values in the system. The first one requires higher functionality than the second, in return, clients can bear larger time complexity or space complexity.

To achieve these characteristics, many systems and data structures are designed. Each systems is superior in some certain operations. For instance, CT can support consistency proof and value-based lookup proof in logarithmic-sized time complexity. However, it can't support efficient non-membership proof and key-value proof since it uses chronological tree as its data structure. By contrast, CONIKS supports key-value proof with membership and non-membership proofs by using lexicographic tree. But auditors need to download linear-sized data in order to make sure that the log is consistent. So, it becomes inefficient when the number of clients becomes larger. Also, since it does not save historical versions, clients have to do the monitoring for every epoch to make sure that their data haven't been changed. As a result, this requires a lot of storage space on the user's computer.

After knowing the advantages and disadvantages of these two methods, a viable option for balancing the speed of both users and servers is to combine the two trees (chronological tree and lexicographic tree) together using some methods. There are some algorithms getting inspiration from it. For example, in Enhanced Certificate Transparency(ECT) , it asks its auditors to make sure that the operations made on the two trees stay the same. The time required for auditor to monitor values is related to the number of appends between epochs. Since servers can't assume that this number is low, the time complexity of auditors could be huge and sometimes unacceptable. Another algorithm is $Merkle^2$ [19], with chronological tree

in their outer layer and lexicographic tree in their inner layer to record all the previous version. The main obstacle for this algorithm is its storage space. It needs O(nlogn) space complexity for the storaging, which is the bottleneck of it. Comparing the two systems, ECT has the advantage that it does not require the clients to monitor, and because it queries directly on the lexicographic tree, its lookup proof efficiency is higher than that of $Merkle^2$'s. Also, since the two trees are separated, the space complexity is only $O(N)$. However, for $Merkle^2$, it is less demanding on the auditor's computing power because its monitoring cost for auditors is not based on the number of values appended between each epoch and has a lower time complexity than ECT.

In addition, there is another path that also eliminates the need for clients to monitor themselves and lowers the monitoring cost of auditors. Append-Only Authenticated Dictionaries(AAD) successfully did this. This algorithm combines the properties of bilinear maps and cyclic groups to construct bilinear accumulators and performs (non-)membership proofs and disjointness proofs and subset proofs through the properties of polynomials and cyclic groups. However, the cost of performing the FFT when computing polynomial multiplication is very high, so its append cost is also very high. And due to its high append cost, it is also very challenging to implement into practical applications.

Currently, for most algorithms, append cost always takes at least $O(\log n)$ time, but this is very bad if there are far more appends than queries in real application scenarios (e.g., in the securities trading domains). So, we began to wonder if there could be a system that made the append cost much smaller and kept the proofs in an acceptable range for these certain areas? To address this problem, we propose ATS System, which significantly boosts the speed when append times are far more than proofs. To meet different conditions of security level, we construct four systems —- ATS_{Base}, ATS^1, ATS^2, ATS^{2*}, which balance the benefits and defects of each kind of operations.

In TABLE I, E stands for the number of epochs and P stands for the number of appends between epochs. The table compares the space and time complexity for all the

TABLE I
Time Complexity for each system

Systems	Storage	Append	Auditor	Owner	Lookup
CT	n	$\log n$	$\log n$	$\log n$	-
CONIKS	$n \log n/$ $E * n$	$\log^2 n$	-	$E \log n$	$\log n$
ECT	n	$\log n$	$P \log n$	-	$\log n$
$Merkle^2$	$n \log n$	$\log^2 n$	$\log n$	$\log^2 n$	$\log^2 n$
AAD	λn	$\lambda \log^3 n$	$\log n$	-	$\log^2 n$
ATS_{Base}	n	$Constant$	$\sqrt{n}$	$\sqrt{n}$	–
ATS^1	n	$Constant$	M	M	–
ATS^2	n	$\log n$	$\sqrt{n}$	$\sqrt{n}$	$\sqrt{n} * \log n$
ATS^{2*}	n	$\log n$	$\sqrt{n} * \log n$	$\sqrt{n}$	$\sqrt{n} * \log n$

A. Summary of our techniques

Motivations. There have been a lot of transparency systems constructed nowadays. However, nearly all of them are built on Merkle Trees. However, this brings them a natural flaws that their append cost will be at least log-level since they use the tree-like data structure. From it, we can see that their appending time are all larger or equal to $log(n)$. From the table, However, a chunking structure can easily surpass these in appending operation. So, we decided to construct our system based on chunking algorithm and expect to find some breakthrough in the efficiency of appending.

The Data Structure. We propose four different systems with two different data structure constructions. In ATS_{Base} and ATS^1, we use chunking structure only in order to maximumly lower the time complexity for appending operation while meeting the requirement of value verifying. To meet the requirement of key-value pair inserting and querying, we also utilize a data structure of prefix tree inside chunking structure. In this way, the time complexity of appending time is still only $O(\log n)$ while the auditor's proof will be larger than ECT for the situation where appending numbers are much larger than proof numbers.

II. System overview

This system is designed to maintain the values or key-value pairs provided by transparency logs. For ATS_{Base} and ATS^1, the system maintains values only. For ATS^2 and ATS^{2*}, this system supports the operation of querying the value of a certain key (user ID).

In this section, we will go over the Authenticated Data Structure of ATS_{Base}, ATS^1, ATS^2 and ATS^{2*} and their operations respectively.

A. Overview of System Members

Similar to $Merkle^2$, the ATS systems consist of three members, Server, Auditors, and Clients. We divide the time in different epochs, and next we will describe the responsibilities of each member in the epochs.

Server. Server's job is to maintain transparency logs to make it easier for clients to query others for key and value. However, server is untrusted, so it needs to provide Clients and Auditors with interfaces for making proofs to ensure that it doesn't change the clients' data. Also, the server needs to ensure that each ID has only one value in the database (clients can have multiple IDs). It requires a digital signature for each operation behavior.

Clients. Clients will store his value into the server and need to check that his data is stored correctly in the server (not being maliciously deleted, changed or added). This part can be done automatically by the clients software instead of being done manually by clients.

Auditors. Auditors need to check in each epoch if the transparency logs are add-only and gossip with others to

get the relevant values in the system. They need to ensure that the server does not manipulate the data in a malicious way, so that the server is consistent across users. They are required to sign for the correctness of the transparency logs after each validation to ensure the trustworthiness of the server. If the server maliciously fakes the logs, they will issue a statement to that effect, signing off on the fakery. Clients can also monitor the behavior of the server as a member of auditors.

B. ATS_{base} and ATS^1's Operations

In this section, we will describe the operations in ATS_{Base} and ATS^1. To make this system safe and reliable, they provides the following operations (Note that all the proofs with "Next" belongs to the ATS^1 system to check the data in the second chunk and ensure their accuracy and integrity):

- Append(val): This operation supports the clients to insert their values into the database. Once inserted into the system, they can't change or delete the value of the same ID. The specific appending details will be discussed in the Section III.A.
- Extension_Proof(_Next)(⟨ver1, ver2⟩): This operation is designed to ensure that the data in the previous version is a subset of the subsequent version for the first chunk structure. We will discuss the details in the Section III.B.
- Membership_Proof(_Next)(ID): Because the server may change the value of a specific ID user, the user needs to prove that a user exists in the server. So, this operation verifies that a user's value exists. The correctness of the value is determined by a series of hashes returned by the server. However, note that the ID in the Membership_Proof_Next operation is not [0,n) and the exact procedure is given in the Section III.C.
- *Concordancy_Proof(ID): This operation is only available in ATS^1. The purpose of this operation is to verify the consistency of the structure of the two blocks, i.e., whether the server deletes information when it is added to the other block. Because the server is untrustworthy, it is possible for the server to add a modified message to the second block while maintaining two blocks at the same time. Therefore, auditors should make sure that the clients' information in the second block should be consistent with the first block. The procedure of it will be provided in Section III.D.

C. $ATS^{2(*)}$'s Operations

In this section, we will describe the operations in ATS^2. To make this system safe and reliable, ATS^2 provides the following operations:

- Append(⟨ID, val⟩): This operation supports the clients to insert their ID-Value pairs into the database. They may change or delete the value of the same ID in "ATS"^{2} system. The specific appending details will be discussed in the Section IV.D.
- Extension_Proof(⟨ver1, ver2⟩): This operation is designed to ensure that the data in the previous version is a subset of the subsequent version for the first chunk structure. We will discuss the details in the Section IV.E.
- Monitoring_Proof: This operation guarantees that the all the key-value pairs have not been modified by the server. The specific process of it will be shown in Section IV.F.
- Lookup_Protocol(ID): This operation is used to lookup the value for each ID. Similar to $Merkle^2$, the server will return all the values added by the same ID. Also, it will provide a dividing point for the values later or no later than the current epoch so that they can check the membership of the value. The exact procedure is given in the Section IV.H.

III. ATS_{base} and ATS^1's Authenticated Data Structure

A. Data Structure Overview

In this data structure, we record only the values and do not consider maintaining key-value pairs, as shown in Figure 1. We number all users sequentially from 0 and their values are denoted by "Val"$_i$. To maximize insertion efficiency, we maintain all values using only the *chunking algorithm*. Assuming that the maximum number of users is N, the size of each chunk will be $\lfloor\sqrt{N}\rfloor$. However, this means that the data structure needs to be reconstructed for every $2n+1$ new users added. By doing this, the average complexity of the server will be $O(N)$ for each added user, which is unacceptable.

In this case, there are two solutions. One is to specify the chunk size as a constant that is approximately equal to the root of the maximum number of users to be accommodated. The other is to dynamically construct the chunk structure. In the first solution, it's easier to realize and more certain since the size of chunks is a constant. Also, it doesn't need to care about the consistency proofs of the problem I'll discuss next. The first one is ATS_{Base}. However, it's not the optimal one when the number of users is large because it needs to go through much more chunks if the chunk size is small. So, we will introduce the second method next.

We reconstruct the chunk structure every time when the total number of users becomes 2^m, so that the average cost of constructing a chunk is $\frac{2^{m+1}}{2^m} = 2$, and the equalized complexity required to reconstruct the chunks becomes $O(1)$.

But this leads to another problem. Because every time the number of users reaches 2^m, the server needs to reconstruct the chunk structure, which can cause the server to lag. To solve that, we construct another chunk structure together with the first one. In the first data structure, the size of the chunk equals to the largest integer power of 2 less than or equal to N. In the second data

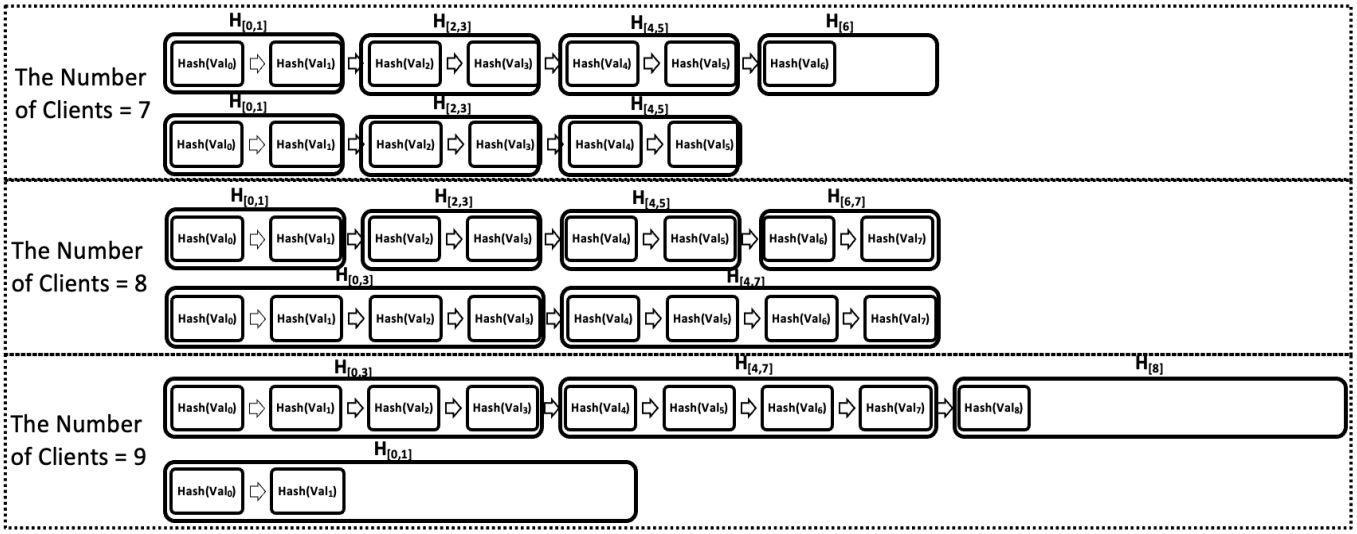

Fig. 1. This diagram depicts the process of maintaining the data structures in ATS^1. Take N to be 7,8,9 and demonstrate each. We define $H_{[a,b]}$ as $H(Val_a, Val_{a+1}, ..., Val_b)$. In this figure, ATS_{base} can be regarded as the upper side of each structure.

structure, the size of the chunk equals to the smallest integer power of 2 greater than N. In practice, we maintain both structures simultaneously.

For each user added, we add the value of that user to the first data structure, and two values immediately following the one already added to it to the second data structure. Whenever the number of users exceeds an integer power of 2, we deprecate the first data structure and treat the second one as the first one. At the same time, we create a new chunk structure, adding the values of the first and second users to it. In this way, we maintain both chunk structures at the same time for each addition, guaranteeing a time complexity of O(1) for each step, instead of an equalized O(1) complexity. However, in this way we need to construct $log_2 n$ chunk structures, each storing n data, which will result in a space complexity of $O(nlogn)$ if we use a two-demensional array with fixed size. In order to minimize the loss of space in this section, we can implement the chunks through a rolling array. This makes the space complexity $O(n)$.

B. Extension Proofs

The extension proofs ensure that this system is only additive. The implementation is as follows:

This system provides an interface to query whether the former version of two versions is a subset of the latter version. That is, the user specifies two versions ⟨ver1, ver2⟩, and the server needs to give the proof whether all the data in ver2 contains all the data in ver1.

Assume that there are n users now, so the chunk size is $\lfloor\sqrt{n}\rfloor$. And assume that the last user ID that has been added before ver1 is ID_1, and the last user ID that has been added before ver2 is ID_2. Notate all chunk hashes prior to ID_1 as $C_0, C_1, ..., C_m$ and all the hashes of element in that chunk as $E_0, E_1, ..., E_k$. Also, notate ID_1 in the chunk as id_1, ID_2 in the chunk as id_2, $Hash(...Hash(Hash(A_0, A_1), ...), A_m)$ as $H(A_1, A_2, ..., A_m)$. So, we determine whether the system keeps append-only by calculate this:

$$H(C_0, C_1, ..., C_m, H(E_0, E_1, ..., E_{id_1}))$$

Then, we can compare this value with the total value of Hash to determine whether the data in ver1 state is correct or not. Then, we can categorise the other works into 2 scenarios:

1) When ID_1 and ID_2 are in the same chunk: The server will first send a label which stands for doing the first kind of operation to clients. Then, it needs to provide all the chunk hashes of the first $\lfloor\sqrt{ID_1}\rfloor$ chunks (excluding the chunk where ID_1 is located) and all the hashes of the elements in the chunk where ID1 and ID2 is located. The client software will provide will calculate

$$H(C_0, C_1, ..., C_m, H(E_0, E_1, ..., E_{id_2}))$$

and see if it equals to the total Hash value which is gossiped by clients and auditors. In this way, we can judge whether the dataset of ver2 contains all the elements in ver1.

2) When ID_1 and ID_2 are not in the same chunk: The server will first send a label which stands for doing the second kind of operation to clients. Then, it needs to provide all the chunk hashes of the first $\lfloor\sqrt{ID_1}\rfloor$ chunks (excluding the chunk where ID_1 is located), all the hashes of the elements in the chunk where ID1 or ID2 is located, and the hash values of all the chunks between ID1,ID2 (excluding the chunk where ID1,ID2 is located). Notate ID2 and all elements hashes in that chunk whose position is before it as $G_0, G_1, ..., G_{id_2}$ and all the total hashes of

chunks between ID_1 and $ID2$ as $F_0, F_1, ..., F_t$. In this way, we can calculate

$$H(C_0, C_1, ..., C_m, H(E_0, E_1, ..., E_k),$$
$$F_0, F_1, ..., F_t, H(G_0, G_1, ..., G_{id_2}))$$

and see if it equals to the total Hash value which is gossiped by clients and auditors. If both values are the same, it means that the data of ver2 is an extension of that of ver1. Otherwise, it isn't and the security of the server needs further investigation.

For Fig. 1, if we want to check whether the data in user volume 7 is a subset of the data in focus of user volume 8, the server will give us in the same chunk, $H_{[0,1]}$, $H_{[2,3]}$, $H_{[4,5]}$, $H_{[6]}$, $H_{[7]}$, and at this time, we can verify that the computed $H_{[0,6]}$ is the value of the previous gossip; and at the same time, we can verify that $H_{[0,7]}$ is the value of the gossip as well. value. Because this data structure is arranged in chronological order, we can finish its extension proof.

For ATS_1, since we maintain two chunk structures simultaneously, we need to check both of them.

C. Membership Proofs

Membership proofs ensure that user information is verifiable in the system. That means that, other users can use certain information to confirm that a particular user's information can be verified to exist in the system.

When doing Membership Proofs, the server issues the user a hash of all the elements of the chunk in which the entered ID is located, as well as the chunk hashes of all other chunks. We notate the value of the user numbered ID Val_{ID}, the block before the block in which the ID is located as $C_0, C_1, ..., C_m$, the block after the block in which the ID is located as $C_{m+1}, C_{m+2}, ..., C_k$, and the elements in the block except Val_{ID} are $E_0, E_1, ...E_{ID-1}, E_{ID+1}, ..., E_t$.

In order to determine whether a user exists in the database, we need to calculate:

$$H(H(C_0, C_1, ..., C_m), H(E_0, E_1, ..., Val_{ID}, ..., E_k),$$

$$C_{m+1}, C_{m+2}, ..., C_k)$$

Further, we compare it with the total hash value at this point to see if it is equal, if it is equal, then it proves that this user information exists in the current database, otherwise it does not exist.

For example, in Figure 1, we want to check if the user with ID 6 and value Val_6 exists when the number of users is 9. The server will provide $H_{[0,3]}, H_{[4]}, H_{[5]}, H_{[7]} and H_{[9]}$. After getting all the value, we calculate:

$$H(H_{[0,3]}, H(H_{[4]}, H_{[5]}, H(Val_6), H_{[7]}), H_{[9]})$$

If this value is the same as the value coming out of gossip, the value is considered to exist, otherwise it does not.

D. Concordancy Proofs

This section is only for ATS^1. Because this data structure is dynamically constructed with two chunks to avoid the surge in arithmetic demand at 2^n brought about by reconfiguration, the user needs to supervise the server's behaviour of maliciously modify data in the second chunk. Which is, clients and auditors need to make sure that the data in previous chunk structure and the latter one remains consistent.

For Users, each time the server rebuilds, the user will receive a notice whether or not to accept do the membership proof. If they accept, then the server will deliver all the user data for the chunk in which the client is located and all the chunk hashes for the other chunks in this chunk structure. This way, for each refactoring, he can perform a membership proof to ensure that his values have not been maliciously tampered with due to alternating chunks. For a database of 1,000,000 users, a user only needs to perform the operation for 20 times maximumly.

For Auditors, they can randomly check a chunk in the current version in each of their operations. For each operation, the server needs to provide all the hashes elements of in the current-version data in one chunk and the chunk hashes of the other chunks in this version. At the same time, it needs to provide all the elements of all the chunks that contain those elements and the chunk hashes of the other chunks in the previous version. By calculating the values in the previous chunk and comparing it with the gossiped value, we can know whether the server provides the correct values for the previous version. Then, we can compare the values in the previous version and the current version so that we can make sure that the values given by the server stay the same. Then, we need to calculate the values in the current version to make sure that the value provided in the server is indeed the Hashes in the current chunk. It can be proved that when the number of users reaches 1,000,000, about 7,000 such operations need to be performed.

IV. $ATS^{2(*)}$'s Authenticated Data Structure

As shown in Figure 2, this data structure consists of a chunk structure and a merkle tree.

A. Chunk Structure

In Figure 2, the data structure on the lower side is that of the chunking aspect chosen for ATS^2 system. As it shows, we choose the structure of the ATS_{Base} system for the chunk data structure to ensure stronger security of the data structure and less burden on auditors and users. In this way, the clients and auditors don't need to do the concordance proof, which costs a lot of time for them. The Chunk Structure is arranged in chronological order, so it is relatively easy to check the operations related to time. However, it is difficult to save key-value pairs and

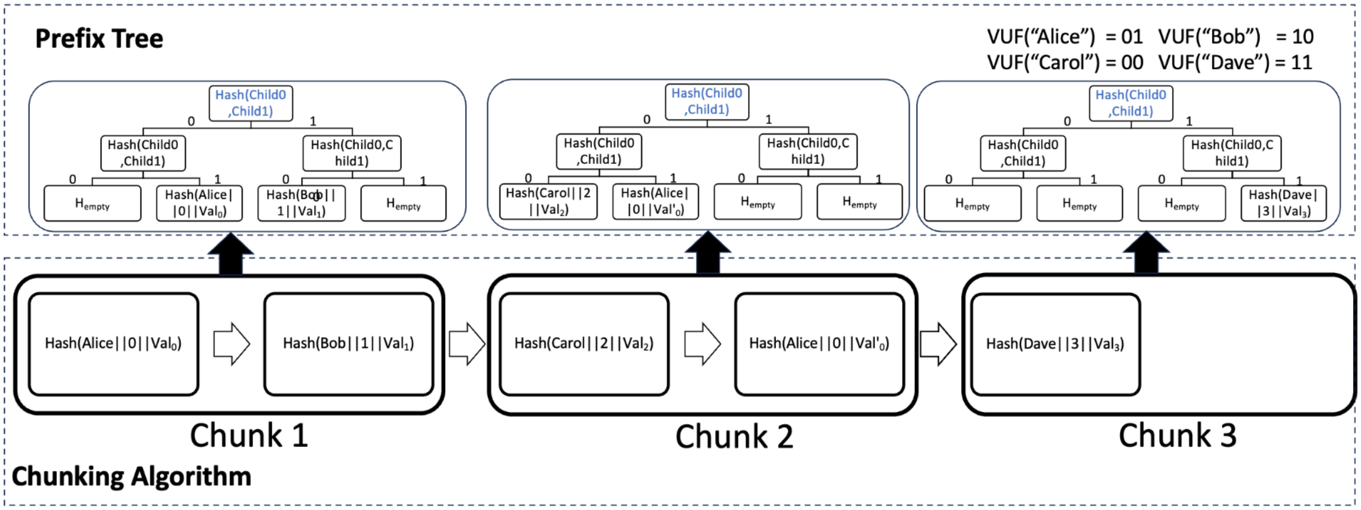

Fig. 2. This diagram depicts the process of maintaining the data structures in ATS^2. Assuming a chunk size of 2, a total number of operations of 5 is simulated. VUF stands for verifiable unpredictable function. In the whole data structure, Compressed Prefix Tree is utilized in the inner layer and Chunk Data Structure is utilized in the outer layer. In the figure, E stands for the number of epochs and P stands for the number of append operations in epochs

cannot perform non-membership proofs, which makes its use highly limited.

B. Merkle Tree

In Figure 2, the data structure on the upper side is that of the chunking aspect chosen for ATS^2 system. In the implementation, we use a prefix tree. The prefix tree stores the user's key-value pairs in dictionary form. The construction of the prefix tree is described below:

- The leaf nodes of the prefix tree are hashes of each user's information.
- The parent nodes are calculations of hashes of the children nodes.

To fix the maximum length of the key, we encrypt the user's key using the verifiable unpredictable function(VUF). The encryption length is 2logn (N is the fixed maximum number of users). The explanation of this will be discussed in the Appendix.

The advantage of a prefix tree is that it is easy to query information about key-value pairs since each of its nodes is categorised by the hash value of the user's key. However, due to its unique structure, it is not easy to query time-related information, such as, it is not easy to perform extension proofs.

C. Combination of Chunk Structure and Merkle Tree

As in Fig. 2, the overall structure of ATS2 is shown. This data structure has two layers, the chunking algorithm is the outer layer and the Merkle Tree is in the inner layer. Intuitively, the benefit is that it maintains a certain degree of temporal ordering while allowing for querying key-value pairs. For each block, the overall information maintained is the hashes of all the elements in that chunk and the root value of the merkle tree. This information is maintained if and only if a block's elements are full.

D. Append Operation

In append operation, both chunk structure and Merkle Tree will be operated in order to maximize the advantages of them which have discussed above.

- For chunk structure, a key-value pair is added with O(1) complexity in the chronological way for each operation. The Hash value should be calculated with the key, the number of append operation and the value of the key which are shown in the figure 2.
- For Merkle Tree, each time a new key is added, the current-chunk Merkle tree's index stays at an empty node which has the longest common prefix in the Merkle tree. After reaching the empty node, the value of that leaf node will be updated with same values in chunk structure's append operation and the value of each parent node is computed retrospectively by following the rules of arithmetic for each node of the prefix tree upwards. However, it's hard to know whether clients' information in the prefix tree is consistent with the one in chunk structure if the prefix tree is not fixed. We also notice that the only prefix tree that's not fixed is the one of the last chunk. So, to make the history persistent, we use the data structure of persistent prefix tree for the last chunk. Since each appending operation changes only $logn$ nodes in the tree, it requires a time complexity of $O(logn)$.

In conclusion, the total time complexity for adding a key-value pair will be $O(logn)$.

Take Carol's append operation for example. When Carol appends his key-value pair into the database, the chunk

structure will add hash of the key "Carol", together with his number "2" and his value Val_2 in it. Then, the corresponding prefix tree will add this hash value to the left node of the root since "0" is a prefix of an empty node for the second prefix tree. Then, the process of appending is finished.

E. Extension Proofs

A data structure that maintains a transparent log needs to ensure that it does not change previous data. Extension proofs are used to check this requirement. The extension proofs for ATS^2 are very similar to those for $ATS1_{Base}$, and the differences are covered next.

Similarly, the extension proofs of ATS^2 query whether the information stored in the current block structure is the same as that of the previous block. However, the auditors should also make sure that the values in prefix trees don't change. In this way, each item will be bound with the current root value of the current prefix tree.

In each chunk, we notate the value of each user $E_0, E_1, ...E_m$ and each root value of the prefix tree $Rt_0, Rt_1, ..., Rt_m$. Then, for each chunk, we maintain the following value:

$$H(E_0||Rt_0, E_1||Rt_1, ..., E_m||Rt_m)$$

The server needs to provide all the values used in ATS_{Base}'s as well as all the root values in the needed chunks. Obtaining these values, we do the same operation as the ATS_{Base}'s to verify if data in the current version is an extension of a previous version's.

This is the extension proof for ATS^2 and its time complexity is $O(\sqrt{n})$ since the data provided by the server is in radial level. However, it can't defend the attack which changes the values in a version the current chunk and changes it back when the number of current chunk hasn't changed. In this way, the values in the current chunk may not be correct. Even if it's not a serious problem for medical or share transaction scene since they don't have to monitor the current data and the data are needed to verify much fewer times, we need to provide a way for auditors to verify the last chunk's correctness.

So, besides the similar extension proofs of the ATS_{Base}'s, we also need auditing to ensure that the current prefix tree is also append-only. In this way, we need to use Persistent Prefix Tree to record the historical versions. However, since the previous versions are being verified and can't be easily modified, we only need to maintain the Persistent Prefix Tree of the last chunk. In this way, there will be at most $\sqrt{n}$ nodes that need to be operated extension proofs for each version. Then, we have two options:

- Auditors audit all the extension proofs for current chunk, which costs $O(\sqrt{n} * \log n)$ in total.
- Each auditor randomly verifies one key-value pair in the current chunk, which needs $O(\log n)$ for each operation. The expectation for the times verifying all the pairs is $\sum_{i=1}^{m} \frac{m}{i}$, which m stands for the number of users in the current chunk.

So, the total time complexity of extension proofs will be nearly $O(\sqrt{n} * \log n)$ for each epoch in ATS^{2*}.

F. Monitoring Proofs

We have to ensure that each key-value pair is not modified.

For each key-value pair, it must be included in a chunk. For monitoring proof, we perform membership proof on the prefix tree related to that chunk. At the same time, we need to check its chunk structure to make sure that the server provides the correct values.

- First, the server provides the authenticated path for the key-value pair that wants to be verified, together with the chunk hashes and elements in the chunks.
- Then, the clients or auditors can verify whether the value is correctly

G. Signature Chains

In extension proofs and monitoring proofs, we prevent attackers from deleting or modifying key-value pairs. However, attackers still can add incorrect key-value pairs without being noticed. So, like $Merkle^2$, we implement signature chains to prevent this situation.

For multiple values corresponding to a single id, for the latter value, we store the location of the previous value and sign it with the private key corresponding to the previous value, so that anyone can use the previous value to verify the signature. This ensures that each value on the signature chains is authentic.

Also, there's another way of signing for users to lookup the latest value to the key easier. We may change the signature chains to use the first value corresponding to the key (instead of the previous value) for signing, so that the first value and the last value can be easily used.

H. Lookup Protocol

This protocol need the system to find all values corresponding to the given key.

The server looks up the key in the prefix tree corresponding to each of the chunking structure and give all its occurrences. The user verifies that the signature chains are correct.

For the prefix trees where the key exists, a membership proof is performed; for the prefix trees where the id does not exist, a non-membership proof is performed. In this way, whether the signature chains are correct can be easily verified.

Since membership proof and non-membership proof both require the root hash of the prefix tree, not only the authentication path on the prefix tree should be provided, but also the hash value of the total Hash value of the first to the second last element and value of last element of each chunking algorithm should be given to recalculate the root

hash of each chunks. By calculating the gossiped value using the given hashes, we can verify whether the root hashes of the prefix tree given by the server are correct.

Let the number of values corresponding to the key be l, which stands for the length of the signature chains is l, and each chunk of total chunking structure has to perform a (non-)membership proof. The total time complexity is $O(l + \sqrt{n} * \log n)$.

Lookup For the Latest Key-Value Pair: The above method looks up every value corresponding to the id, and sometimes only the last value.

By changing the signature chains to use the first value corresponding to the id (instead of the previous value), the lookup only requires the first and last values and membership proof. But we also need to prove that this is indeed the first and last value, so we need non-membership proof of the prefix tree to the left of the first value and to the right of the last value.

In total, there are at most 2 membership proofs and $O(\sqrt{n})$ non-membership proofs, with a total time complexity of $O(\sqrt{n} * \log n)$.

V. Applications of ATS

In this section, we will discuss two applications of Advanced Transparency System:

First, in medical field, it can be widely used. In the medical field, patient information is only needed when the patient is queried for a specific purpose. However, this situation is very rare, so it is more efficient to use this data structure.

Second, it can also be applied in the financial field. In the financial field, it is only necessary to check when the supervisor believes that there is an abnormality. However, the demand for append operations is very large. Therefore, this system is also very good.

Acknowledgment

Special thanks to Siyuan Cheng for all the help he gave me in this endeavor. He and I have worked together on this topic with different methods. Chatting with him has been a great inspiration to me.

References

[1] O. Gasser, et al., "In log we trust: Revealing poor security practices with certificate transparency logs and internet measurements," in Passive and Active Measurement: 19th International Conference, PAM 2018, Berlin, Germany, Mar. 26–27, 2018, Springer International Publishing, 2018.

[2] R. Peeters, T. Pulls, "Insynd: Improved privacy-preserving transparency logging," in Computer Security–ESORICS 2016: 21st European Symposium on Research in Computer Security, Heraklion, Greece, Sep. 26-30, 2016, Proceedings, Part II, vol. 21, Springer International Publishing, 2016, pp. 121-139.

[3] R. Samavi and M. P. Consens, "Publishing privacy logs to facilitate transparency and accountability," Journal of Web Semantics, vol. 50, pp. 1-20, 2018.

[4] C. Yue, T. T. A. Dinh, Z. Xie, et al., "GlassDB: An Efficient Verifiable Ledger Database System Through Transparency," Proceedings of the VLDB Endowment, vol. 16, no. 6, pp. 1359-1371, 2023.

[5] Q. Scheitle, O. Gasser, T. Nolte, et al., "The rise of certificate transparency and its implications on the internet ecosystem," in Proceedings of the Internet Measurement Conference 2018, 2018, pp. 343-349.

[6] Laurie, B., and Kasper, E. "Revocation Transparency," Google Research, September 2012, vol. 33.

[7] N. J. Al Fardan and K. G. Paterson, "Lucky thirteen: Breaking the TLS and DTLS record protocols," in 2013 IEEE Symposium on Security and Privacy, IEEE, 2013, pp. 526-540.

[8] R. Housley, W. Polk, W. Ford, and D. Solo. "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile," RFC 5280, May 2008. Available online: https://www.rfc-editor.org/rfc/rfc5280.

[9] Laurie, B. "Certificate Transparency," Communications of the ACM, vol. 57, no. 10, pp. 40–46, 2014.

[10] Chuat, L., Szalachowski, P., Perrig, A., Laurie, B., and Messeri, E. "Efficient Gossip Protocols for Verifying the Consistency of Certificate Logs," in 2015 IEEE Conference on Communications and Network Security (CNS), pp. 415–423, 2015.

[11] J. Gustafsson, G. Overier, M. Arlitt, et al., "A first look at the CT landscape: Certificate Transparency logs in practice," in Passive and Active Measurement: 18th International Conference, PAM 2017, Sydney, NSW, Australia, Mar. 30-31, 2017, Springer International Publishing, 2017, pp. 87-99.

[12] B. Dowling, F. Günther, U. Herath, et al., "Secure logging schemes and certificate transparency," in Computer Security–ESORICS 2016: 21st European Symposium on Research in Computer Security, Heraklion, Greece, Sep. 26-30, 2016, Proceedings, Part II, vol. 21, Springer International Publishing, 2016, pp. 140-158.

[13] R. Dahlberg, T. Pulls, "Verifiable light-weight monitoring for certificate transparency logs," in Secure IT Systems: 23rd Nordic Conference, NordSec 2018, Oslo, Norway, Nov. 28-30, 2018, Proceedings, vol. 23, Springer International Publishing, 2018, pp. 171-183.

[14] E. Fasllija, H. F. Enişer, B. Prünster, "Phish-Hook: Detecting phishing certificates using certificate transparency logs," in Security and Privacy in Communication Networks: 15th EAI International Conference, SecureComm 2019, Orlando, FL, USA, Oct. 23–25, 2019, Proceedings, Part II, vol. 15, Springer International Publishing, 2019, pp. 320-334.

[15] S. Pletinckx, T. D. Nguyen, T. Fiebig, et al., "Certifiably vulnerable: Using certificate transparency logs for target reconnaissance," in 8th IEEE European Symposium on Security and Privacy Workshops, IEEE, 2023.

[16] S. Eskandarian, E. Messeri, J. Bonneau, et al., "Certificate transparency with privacy," arXiv preprint arXiv:1703.02209, 2017.

[17] A. Langley, E. Kasper, B. Laurie, "Certificate transparency," Internet Engineering Task Force, 2013.

[18] Ryan, M. D. "Enhanced Certificate Transparency and End-to-End Encrypted Mail," Cryptology ePrint Archive, 2013.

[19] Hu, Y., Hooshmand, K., Kalidhindi, H., Yang, S. J., and Popa, R. A. "Merklê 2: A low-latency transparency log system," in 2021 IEEE Symposium on Security and Privacy (SP), pp. 285–303, 2021.

[20] Li, J., Krohn, M. N., Mazieres, D., and Shasha, D. E. "Secure Untrusted Data Repository (SUNDR)," in Osdi, vol. 4, pp. 9–9, 2004.

[21] Yu, J., Cheval, V., and Ryan, M. "DTKI: A new formalized PKI with verifiable trusted parties," The Computer Journal, vol. 59, no. 11, pp. 1695–1713, 2016.

[22] A. K. Lenstra, J. P. Hughes, M. Augier, et al., "Public keys," in Annual Cryptology Conference, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 626-642.

[23] Melara, A., Blankstein, A., Narayanan, A., Gunnar, L., and Freedman, M. J. "CONIKS: Bringing Key Transparency to End Users," in Proceedings of the USENIX Security Symposium, August 2015. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/melara.

[24] Bonneau, J. "EthIKS: Using Ethereum to Audit a CONIKS Key Transparency Log," in International Conference on Financial Cryptography and Data Security, pp. 95–105, 2016.

[25] H. Malvai, L. Kokoris-Kogias, A. Sonnino, et al., "Parakeet: Practical key transparency for end-to-end encrypted messaging," Cryptology ePrint Archive, 2023.
[26] B. Chen, Y. Dodis, E. Ghosh, et al., "Rotatable zero knowledge sets: Post compromise secure auditable dictionaries with application to key transparency," in International Conference on the Theory and Application of Cryptology and Information Security, Cham: Springer Nature Switzerland, 2022, pp. 547-580.
[27] Tomescu A, Bhupatiraju V, Papadopoulos D, et al. Transparency logs via append-only authenticated dictionaries[C]//Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. 2019: 1299-1316.
[28] Andersen, M. P., Kumar, S., AbdelBaky, M., Fierro, G., Kolb, J., Kim, H.-S., Culler, D. E., and Popa, R. A. "WAVE: A Decentralized Authorization Framework with Transitive Delegation," in 28th USENIX Security Symposium (USENIX Security 19), Santa Clara, CA, pp. 1375–1392, 2019.

Appendix A

Proof for Expected Verifying Times

We can simplify the problem to this:

If there are N numbers, each time we check one randomly, what is the expectation for the times we need to check all the numbers at least once?

Let's consider selecting the first number. Since there are n different numbers, the probability of choosing any one number in the first try is $\frac{1}{N}$. Hence, on average, it's expected to select a new number with the expectation of $N * \frac{1}{N}$, which is 1 in the first trial.

For the second new number, since the first number is chosen, there are $N1$ numbers left. Therefore, the expected number of trials to choose the second new number is $\frac{N}{N-1}$. This is because there's a $\frac{N-1}{N}$ chance of selecting a new number.

After that, for i^{th} new number, the expected number of trials for each new number chosen will be $\frac{N}{N-i+1}$.

So, in conclusion, the expected verifying times will be $\sum_{i=1}^{N} \frac{N}{i}$

Appendix B

All Data Structures

Chunk Structure

In chunk structure, we maintain the information using chunks. In each chunk, we maintain a total information for all data in the chunk. In this way, it becomes more efficient since we only need to look for information in $\sqrt{n}$ chunks.

Merkle Tree

Merkle Tree is a tree-like data structure to efficiently verify the integrity and accuracy of data by combining the hash values of data blocks into has values of larger data blocks.

To authenticate the integrity and accuracy of data, Merkle Tree can provide the authenticated path the clients and auditors, which contains the all the node hashes which are on the co-path of the given index.

Prefix Tree

A Prefix Tree is composed of nodes and edges. Each node (except for the root) typically represents a character of the string. The root node does not contain a character or represents an empty string. Each node contains links or references to other nodes. The connection between them is a character. For any node in the tree, the string composed by all the links on the parent path is a prefix of some keys. Next are a couple of variants of this data structure:

Persistent Prefix Tree: After acknowledging the construct method of prefix tree, consider the application scenario of maintaining history versions. One way of achieving this is recording all the history versions by using n trees. (n is the number of the history versions) However, it's not optimal since it will need $O(n \log n)$ space complexity to maintain the data. Thus, we need a new data structure.

We find that only $O(\log n)$ nodes are changed in the tree when adding a new value in the tree. So, we can build a new root and link all the co-nodes which doesn't change in the modifying operation. So, we only need to add $O(\log n)$ nodes to the tree for each operation. In this way, the space complexity will be $O(n \log n)$ if there are n added values. In ATS^{2*}, since we only need to maintain the persistent prefix tree for last chunk, the total complexity will be $O(\sqrt{n} * \log n)$, which is smaller than $O(n)$. So, the space complexity for ATS^{2*} is $O(n)$.

Compressed Prefix Tree: In the prefix tree, many nodes are not demanding since they don't provide new information. For example, after hashing using VUF, a key will be hashed into a fixed-length string. Assume that it's the first key to be inserted, we find that it is not necessary to insert all the characters in the string, since too many nodes doesn't provide more information than one node. So, we employ a strategy that we create one new nodes to the tree if and only if one of its prefix is different from all the prefix inserted to the tree. Then, we create a new node, which stands for all the characters except the common prefix.

So, for a new inserting, we shall only create one more node, which guarantees that the total space complexity will be $O(n)$ instead of $O(n \log n)$