Supporting Multi-Leaf, Atomic Updates on Log-Based Merkle Trees

Nikhil Athreya
University of California, Berkeley
nathreya@berkeley.edu

Nicholas Riasanovsky
University of California, Berkeley
njriasanovsky@berkeley.edu

Abstract

The use of Public-Key Infrastructure (PKI) [16], specifically Certificate Transparency (CT) [8] to manage and distribute user certificates has brought with it significant scalability challenges due to its usage of Merkle trees [11]. Specifically, the hierarchical nature of Merkle trees resists attempts at parallelization of multi-node updates [4] in a transactional manner. As users accumulate more devices that share public keys, it becomes increasingly necessary to support updating all of a user's devices (nodes in a Merkle tree) to a new public key; this can happen if, for example, a user's device becomes compromised.

This paper introduces a system that supports these types of updates based of Google's existing Merkle tree implementation, Trillian [9] and provides a performance comparison between Trillian and the modified system. Benchmarks show that the modifications made have similar performance numbers when run on read-heavy workloads, but provides speed ups compared to a base Trillian implementation on write-heavy workloads where transactions involve multiple nodes.

1 Introduction

The rise of Public-Key Infrastructure (PKI) [16], a set of techniques to deal with the management, distribution, and revocation of digital certificates for public-key encryption, has brought with it a set of unique challenges. Among these are the common scaling issues of the management of its many moving parts and its ability to meet the needs of a growing number of users storing their public keys in the infrastructure. Yet an equally important challenge is its ability to update these keys as they change over time and distribute these keys to an ever-growing number of users. We consider the ability to update public keys of particular importance because users need the ability to react quickly to the changing state of the security of their devices.

One approach to PKI by Google is Certificate Transparency (CT) [8], which seeks to add transparency to the certificate creation and distribution process. The way that CT attempts to solve this problem is by providing a log of certificates. This log has three properties:

1. It is append-only, which means that certificates cannot be inserted at arbitrary points in the log and entities (possibly malicious), cannot delete or change log entries in an adhoc way.

2. It is cryptographically secure. Users of CT can be sure that their public keys weren't tampered with.

3. Closely related is CT's ability to be publicly auditable. A major reason that users can be sure that their public keys weren't tampered with is because anyone can request a proof that their certificate has been legitimately appended to the log. Since this audit is cryptographically secure, it is guaranteed with a very high probability, that no malicious entity would have the ability to tamper with the log.

The way that CT provides these strong guarantees is through the use of a Merkle tree for its append-only log. Since a Merkle tree is a tree composed of a chaining of cryptographically secure hashes over its nodes, with very high probability, it will be impossible for a malicious entity to be able to forge log entries.

Thus, a key component in the performance of CT is its backing Merkle tree implementation. However, since a Merkle tree is organized as a hierarchical chain of hashes, it is very difficult to achieve update parallelism on Merkle trees [4] when compared to other tree-based data structures. There has already been significant work on how to perform single-node updates to Merkle trees [9, 11] efficiently. However, with the application of Merkle trees to PKI, specifically CT, this is quickly becoming an insufficient solution to a growing problem of scale.

A common use-case for PKI and CT is the management of public keys for user devices [7]. As the number of users grows, so to does the size of the Merkle trees required to store their public-key information. However, the problem now is that it is very common for users to have multiple devices between which they share a single public key. Thus, when a user's device becomes compromised, it would be desirable to change the public keys of all of a user's nodes to a new public key. This poses a significant performance problem, since it is very difficult to perform scalably parallel updates on Merkle trees as noted earlier. An additional challenge is posed by the fact that the update from one public key to another public key must be performed atomically. The entire tree must be in a consistent state at any point in time, and a partial update would mean that until the entire transaction completed, some devices of a user might still utilize an older, insecure public key.

This paper introduces a modified Merkle tree based off of Google's Trillian [9] Merkle tree implementation, that has support for transactional, multi-node updates for log-based Merkle trees. The addition of these changes should not introduce prohibitive latency overheads and should retain comparable throughput.

Work on atomic, multi-leaf update operations for Merkle trees would have far-reaching consequences, since Merkle trees are utilized in a wide variety of applications, from cryptocurrencies like Bitcoin [12], Ethereum [17], etc., to PKIs like CT [8], CONIKS [10], etc., to filesystems like ZFS [13], IPFS, etc. All of these would benefit from efficient, transactional, multi-node updates to Merkle trees, since enabling this class of operations would mean opening the door for significant optimizations with regards to the kinds of transactions that could batched together, as well as the consistency guarantees these systems could provide. Additionally, for the purpose of CT, support for this class of operations would mean faster response times to the changing state of the security of user devices, which is an integral component in any PKI.

The remainder of this paper is structured as follows: Section 2 discusses some of the more detailed mechanics with regards to Merkle trees and membership proofs and provides background information about Google's Trillian Merkle tree implementation. Section 3 describes the changes that were made to Trillian's implementation to support transactional, multi-node updates. Section 4 describes the methodology used to provide a performance comparison between Trillian and the modified implementation. Section 5 looks at the work of others in relation to multi-node Merkle tree updates and Merkle trees in general. Section 6 describes some future directions that research based off of this paper could take. Section 7 concludes.

2 Background

2.1 Merkle Trees

A Merkle tree, first introduced by Merkle in [11] is a data structure designed to produce a probablistic proof of inclusion. A Merkle tree is constructed by computing hashes of the stored data and then aggregating those hashes until there is a single hash value for the root. An example of a binary Merkle tree is shown in Figure 1. Proof of membership is possible by returning the path from the leaf to the root, which given a one way hash function demonstrates membership with high probability. Merkle trees are a popular choice of verifiable data structure because they support efficient insertions. To add a new element to a Merkle tree requires at most computing $\log n$ hashes, where n is the size of the tree. As a result

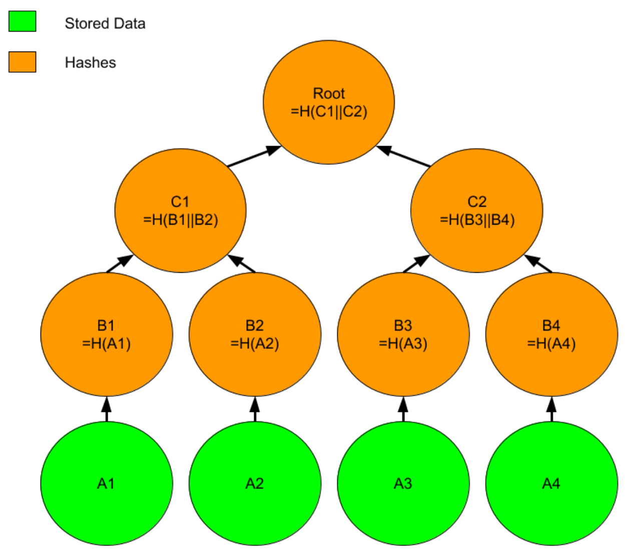


Figure 1: An example of a binary Merkle tree computed over four elements. The green nodes represent the data and orange nodes are the output of a hash function applied to the concatenation of their children.

many Public Key Infrastructure solutions, such as Coniks [10] have Merkle trees in their core design.

2.2 Trillian

For our project we built off of Google's Merkle tree implementation, Trillian [9]. Trillian provides support for both a log based Merkle tree and a map based Merkle tree used to implement key value stores. Since public key infrastructures often rely on a log that can be audited [10], we opted to work directly with the log-based implementation. The log-based system, shown in Figure 2, consists of a client process which facilitates requests, a server process which handles client requests such as returning a proof of membership or queuing a leaf to be added to the tree, and a signer process which is responsible for actually including leaves into the tree and creating a proof of work. When a user wants to add elements to the tree, it places a request with the server which then adds the elements to the storage layer. Then periodically the signer polls the storage layers for leaves to add, selecting up to some batch size number of leaves in the order they were added to the storage layer, and adds them to the tree. While there are many different options for the storage layer we opted to analyze Trillian using MySQL [1].

Trillian implements a Merkle tree by introducing many subtrees for any given depth. For the subtrees at depth d, a particular leaf stored at index i is a member of the tree denoted by its uppermost d bits. A subtree is considered perfect if all possible indices in a subtree are present in the storage layer. If a subtree is perfect then the root

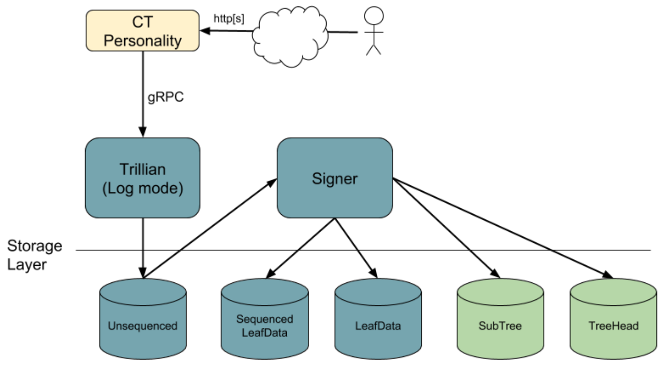


Figure 2: A diagram of the Trillian architecture originally produced in [9]. Trillian consists uses two different processes: the server (shown as Trillian in the diagram) and the signer to update the tree. The server handles user requests and the signer actually incorporates the data into the tree. These two processes do not formally interact and instead communicate through an underlying storage layer.

of the subtree can be stored long-term because no path to computing it will ever change. Trillian optimizes for this case by storing only the roots of the largest perfect subtree to which each leaf is a member, as shown in Figure 3. When a root needs to be computed again, these stored roots are hashed together to compute the root of overall tree, minimizing the necessary storage while also removing any strictly unnecessary hashes.

When adding leaves to the tree, Trillian can either place leaves in strictly increasing locations or at predetermined locations. For our project we chose to only consider increasing locations because it should be more computationally efficient and thus provides a harsher metric to compare against. Adding leaves in a strictly increasing order minimizes the number of imperfect subtrees. This has the effect of both reducing the storage necessary for Trillian (and thus cost of interacting with the storage layer) and reducing the total number of hashes that need to be computed to recompute the root.

3 System Design

To implement atomic user transactions in Trillian we opted to make changes to both the server and the signer to facilitate proper functionality. At the server we introduced a virtual leaf and mapped virtual leaves in the storage layer. We then introduced versions to allow for updating virtual leaves. Collectively these allowed us to construct our transactions. Then we introduced a transaction cache in the signer and modified all the tree updates to route through this cache. This allowed us to achieve atomic updates of the tree. We present an overview of our modifications to Trillian in Figure 4.

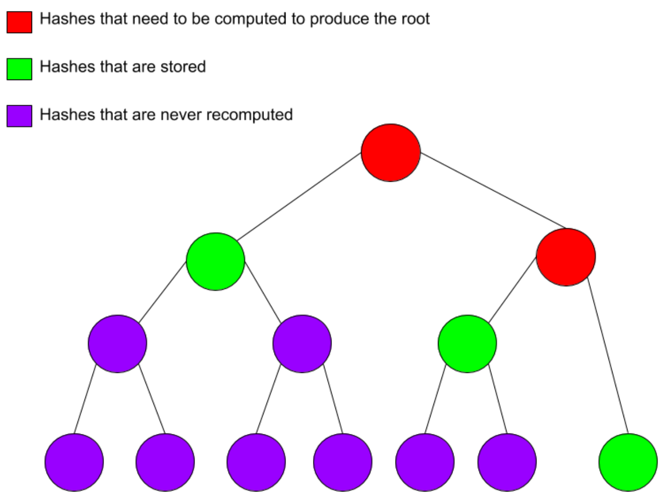

🟥 Hashes that need to be computed to produce the root
🟩 Hashes that are stored
🟪 Hashes that are never recomputed

Figure 3: A diagram displaying the hashes necessary for a binary Merkle tree with seven leaves. The nodes in green are the hashes explicitly stored by Trillian. The nodes in red are hashes that need to be recomputed anytime the root is updated. Finally the nodes in purple are hashes that will never need to be recomputed because they have a stored parent.

3.1 Virtual Leaves

Since we are building based off of a log-based Merkle tree, we needed to associate a series of a leaves with a single virtual identity. To do so we opted to construct key value pairs, with a key being a public key and the values as tuples of (user identifier, device identifier). We believe this is a good model because a user may have many devices but will likely register many of them with a common public key. Furthermore users may share devices, creating the need for identifying a particular user. When a user elects to change their public key, they most likely have lost the associated private key or their device has potentially been compromised. For our implementation we assume all public key changes are a result of lost keys and omitted considering deleting or transferring devices. To facilitate key replacement, when an update to a particular key is requested, a new leaf is added to the tree for each value mapped to by a public key value. Then the mapping is deleted and the virtual leaves are redefined by a mapping from the new public key. As a result, although there may be many different leaves for the same (user identifier, device identifier), the only actual accessible leaves will be those described by valid public key values.

3.2 Versioning

To ensure that requests from the tree occur properly it was also necessary to modify the read API to implement our virtual leaves. Initially we believed it would be possible to facilitate reads entirely using our virtual leaves. However, because the signer runs as a separate process,

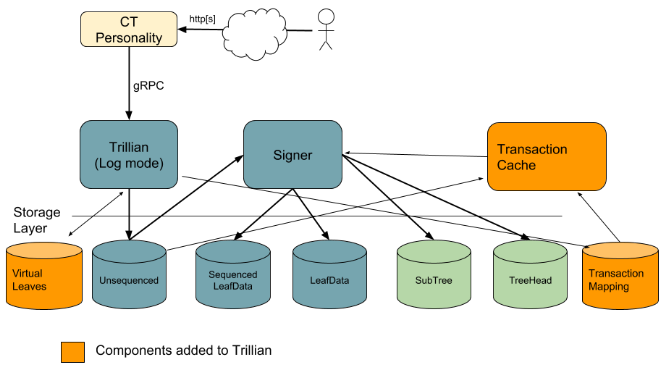


Figure 4: Diagram showing our changes to the Trillian architecture in orange. We made two storage level changes to support virtual leaves and transaction mapping and one change to the signer for transaction caching.

it is possible that the updates to the virtual leaves may not be incorporated into the tree yet. Consequently when performing a request for a leaf, our modified server will fetch all actual leaves that were ever incorporated into the tree for a virtual leaf. Then the server returns the leaf with the latest timestamp. One potential implication of this is that leaf requests now require reads from storage that are linear in the number of key changes. We believe this could be later optimized but may require larger architectural changes.

3.3 Transaction Mapping

In addition to having the server generate individual transactions we also needed to implement a system to manage existing transactions. This is because the signer is running as a separate process and thus cannot directly accept a transaction. Additionally it should not be necessary for the server to generate the transaction and clients should be able to link leaves together in a transaction (although we do not demonstrate this case). To handle this situation we introduce a transaction map and assign a transaction identifier to every leaf we add to the storage layer. This transaction map holds the number of leaves that need to be added to the tree for each transaction identifier not yet included in the tree. All processing by the signer requires an interaction with this mapping to verify that any transaction it attempts to update is complete. Then the signer is able to implement these transactions atomically by relying on the inclusion of transactions at the storage layer in an atomic manner.

3.4 Transaction Cache

To support updating atomic transactions we implemented an in-memory, software transaction cache. When attempting to append leaves to the tree, the signers will still dequeue leaves from the storage layer in order. However, rather than immediately adding the fetched leaves to the tree, the leaves are first added to data structures containing other leaves for a given transaction. This will be paired with a request from the storage layer for the size of all pending transactions to determine when a transaction has been completely fetched. If the transaction is full it will be added to the tree, possibly in batches with at most a specified "batch size" number of other complete transactions. If there are more than the specified batch size of complete transactions, currently cached full transactions will be added without interacting with the storage layer on future signing intervals. Otherwise leaves are fetched based upon the number of leaves missing from the most recent "batch size" transactions. Crucially all leaves that are actually inserted into the tree are still inserted with increasing indices. We chose to do this to maintain Trillian's efficient hashing scheme that we discussed in 2.2.

4 Evaluation

To conduct our evaluation we pitted our modifications to Trillian against the original Trillian implementation. To do so we first created a transaction generator to produce either read or write requests. Then since we tested our implementation on a single t2.xlarge Amazon EC2 instance with 4 virtual CPUs and 16 GB of memory, we produced a network simulator to model the additional delay associated with our increased interactions with the storage layer.

4.1 Transaction Generation

In order to conduct our experiments we first had to create a transaction workload. To do so we developed a transaction generator designed to run concurrently in clients. It produces a set of devices for 1000 different users by sampling from a Poisson distribution. We believe a Poisson distribution is an appropriate model for the number of registered devices because there are a discrete number of devices and there is no relationship between disjoint users. Furthermore the Poisson distribution's large tail is a good model for uncommon users with either very few or very many devices, which we want to be considered in our workloads. We choose to assign each user the same public key for all devices. Although we support users holding multiple public keys for different or even the same devices, we considered attempts to split user devices across public keys as less telling because the behavior is less concrete.

Once the users are generated, we create an initial base Merkle tree. To do so, we create a request to generate a new virtual leaf for each of the specified user and device pairs. We run this in a single client as a series of sequential write requests. For all experiments conducted we waited for the signer to completely integrate the base tree before handling any further requests.

To generate our transactions on which we based measurements, we specified a number of clients and partition the number of users into discrete domains on which they could operate. For example with 5 clients each would have a series of 200 distinct users for which it could generate transactions. These were selected as a matter of convenience, as we intended to run this clients concurrently. Since our virtual leaves require specifying a leaf to update with the latest public key, two concurrent clients could have conflicting views of a user's current public key in the presence of an update.

To actually generate transactions we opted to have each client generate 1000 transactions, each of which was either a read or a write selected by some specified probability and sampled a virtual leaf uniformly from its partition of the tree. We also opted to include the option to have bursty writes. When enabled, the probability of generating a write on the next transaction is higher if the previous transaction was a write. We believe this is appropriate because we foresee a possibility where due to security and performance concerns, updates to the tree could be conducted by administrators responsible for handling user requests. If this tree could still be queried by general users, then the majority of transactions will be reads, but when administrators update the tree there will be many consecutive write transactions.

4.2 Network Simulation

Merkle trees used in PKI and CT are generally spread across multiple machines due to the volume public keys that they manage. However, we did not have access to such a large number of public keys. With a transaction generator, we had the ability to synthesize public keys for users with a corresponding average device count per user. We now needed the ability to simulate a Merkle tree being spread across multiple machines. To achieve this, a network delay generator was also implemented to simulate the Merkle tree being spread across multiple machines.

Much literature regarding simulating the delays caused by the network due to its complexity and bursty behavior exists. Since the focus of this project was not simulating the network, we instead opted to use a simple Poisson model for network delays, which is a well-established traffic model [5].

A common usage setting for the Merkle trees used in PKI is that these trees live datacenters with operators issuing update requests and general users issuing read requests for leaf data and consistency proofs. To this end, we looked into the typical delays experienced by distributed MySQL read and write transactions within datacenters [3] and made these delays Poisson as well. Since delays will vary depending on the physical machine topology within the datacenter, we chose to base our network delays off of a highly interconnected machine room topology.

Within the Trillian code, we inserted datacenter network delays on incoming write transactions to simulate operators issuing a public key update and did not delay any read requests from users. The reason for this choice is because we wanted to stress both systems, and having no delays from users across the WAN is equivalent to users having a fast network connection to the datacenter, so there could be a higher throughput of read requests. On read and write to the underlying MySQL storage layer, read and write transaction datacenter delays were inserted.

We subjected both Trillian and our modified implementation to this network delay generation in all of the following benchmarks. We did not run any experiments in parallel since only a single machine was used for trials, so different experiments running concurrently could have led to an unfair distribution of CPU usage to one experiment.

4.3 Transaction Size

In our first experiment we decided to assess how well our modifications to Trillian performed at various average transaction sizes. We opted to look at two different transaction sizes, 3 and 9. We believe that 3 is a reasonable baseline transaction size because one study found that the average number of devices per user in the United Kingdom is currently 3.5 [2]. While we think 3 devices per user is a reasonable amount now, we expect the number of devices per user to increase into the future. This is partially because rapid development in the Internet of Things is connecting more devices to the internet and potentially creating more devices that a user must manage. However we believe that studies like [2] suggest that in general, users tend to accumulate more devices over time. Furthermore, when users replace devices, if they do not dispose of these devices then it is possible they could still be associated with a user's public key and therefore must be managed. Ultimately there is nothing particularly significant about an average transaction size of 9 devices other than it being a larger number to demonstrate how well our modifications may scale into the future.

To conduct the actual experiment we constructed transactions for five different use cases: low write frequency, low write frequency and bursty, medium write frequency, medium write frequency and bursty, and high write frequency. For the low write frequency we opted to make the probability of generating a write transaction 5%, for the medium write frequency the probability of generating a write transaction is 30%, and for the high write frequency we made the probability of a write transaction 80%. There is nothing in particular that led to the selection of these numbers as opposed to similar ones. However, we opted to study these three different settings to verify our implementation produced acceptable performance under a variety of workloads. Additionally for bursty situations we opted to make the write probability on a burst 80% simply to be consistent with there being a high probability of a write.

Additionally, in conducting our experiment we opted to run 5 concurrent clients all on a single machine. We elected to consider only 5 clients as opposed to more because we wanted to prevent competition for processor time from interfering with our results. Trillian also allows the operator to set the polling rate of the signer. We opted to set the polling rate to 50ms for all experiments we ran after some manual configuration showed this en-

Trillian Type	Write Probability	Bursty	Transaction Size (leaves)	Average Time to Fill Tree (s)
Original	0.05	No	3	19
Modified	0.05	No	3	17
Original	0.05	Yes	3	17
Modified	0.05	Yes	3	16
Original	0.30	No	3	15
Modified	0.30	No	3	22
Original	0.30	Yes	3	55
Modified	0.30	Yes	3	22
Original	0.80	Not Applicable	3	55
Modified	0.80	Not Applicable	3	30
Original	0.05	No	9	20
Modified	0.05	No	9	19
Original	0.05	Yes	9	21
Modified	0.05	Yes	9	18
Original	0.30	No	9	33
Modified	0.30	No	9	27
Original	0.30	Yes	9	34
Modified	0.30	Yes	9	26
Original	0.80	Not Applicable	9	60
Modified	0.80	Not Applicable	9	42

Table 1: Table of average times to completely update the Merkle Tree for each experiment.

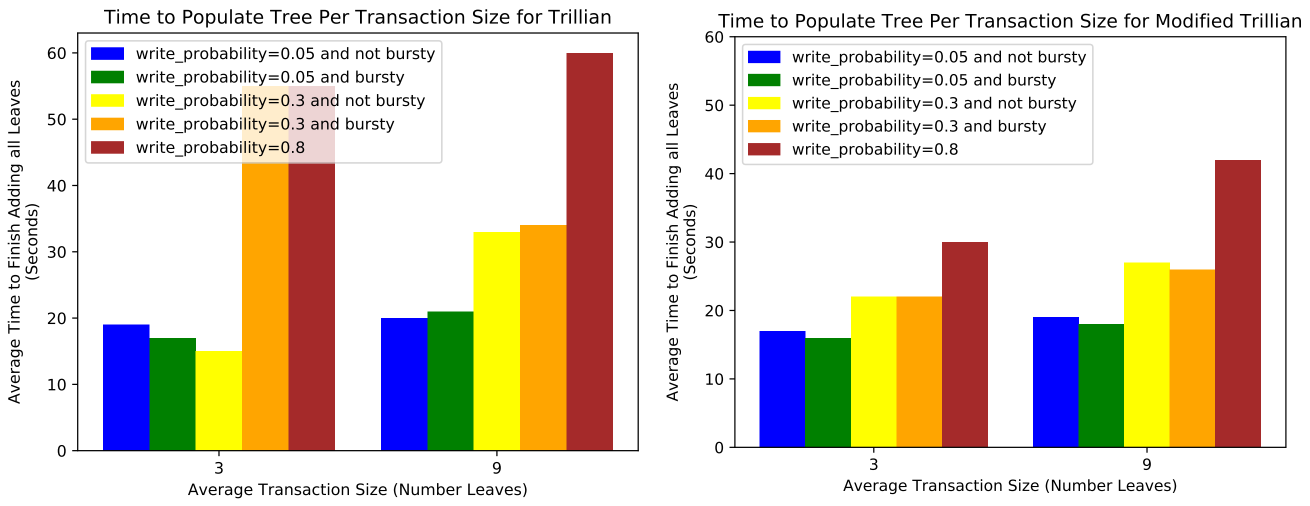


Figure 5: Graphs showing the average time it takes to finish populating the Merkle tree for Trillian and our modifications to Trillian. In nearly every experiment our modifications to Trillian resulted in finishing populating the tree faster than the original Trillian. This difference was most prominent when the write probability was 0.8 and Trillian took 42% longer to finish appending to the tree.

abled timely updates without consuming an unrealistic amount of CPU.

To assess the performance of our modifications we compared our modifications to the original Trillian with respect to both the average time to finish updating the tree and the average throughput while updating the tree. We elected to consider the time to completely update the tree because this end-to-end time considers all delays associated with a particular workload; not only would this include throughput and latency, but also other unforeseen delays that were encountered due to the network for example. The results of our experiment can be seen in Table 1 and are graphed in Figure 5 for easier comparison. Our results showed that at a low write rate the time to update the tree was nearly identical. However when there were a larger number of writes, either because the transactions were because bursty or it was in the high write setting, our implementation is much faster. For example, with an average transaction size of 9 leaves and a high write workload, Trillian takes 42% longer to finish updating the tree.

Additionally, we felt explicitly including the throughput was also insightful because the probabilistic components of our transaction generator created trees of varying sizes across trials. This is helpful because it introduces a normalization component in our analysis. Our throughput results are displayed in Table 2 and are graphed in Figure 6. These results are consistent with what was observed by analyzing the time to update the tree. Our modifications demonstrate a throughput that increases as the number of writes increase. Trillian seems to also show this except for the medium not bursty experiment with a transaction size of 3. This is consistent with our expectations because a larger number of writes produces more signing attempts that will be able to completely fill the max workload.

Our modifications produce a higher throughput in nearly every experiment. For a smaller write frequency we believe this is most likely a reflection of the inaccuracy in our attempts to create Trillian batches equal to the size of our transaction updates. While Trillian has its batch size set to the expected mean, with a small number of writes deviations from the mean will be more significant. If the transaction sizes are smaller than the expected batch size then Trillian will not be able to process additional leaves because there will not be extra leaves in storage. In contrast, if the transaction sizes exceed their expectation, our modifications will be able to fully process them, while a particular signing instance of Trillian will only be able to include up to the expected mean.

However, for large write frequencies we believe our high throughput values are a reflection of the improvements in our architecture. If the write rate is high, then leaves will be added to the tree every time the signer polls the storage layer. To add leaves to the tree it is also necessary to compute all the hashes to create a new root, and thus in heavy write situations the time to update the tree may exceed our polling interval. This scenario is the best demonstration of the differences in architecture because throughput is no longer being capped by the decision to poll. In this instance we believe that the ability to cache additional leaves in memory provides the advantage of decreased interactions with the storage layer and increases our overall throughput.

4.4 Client Scaling

For our next experiment we opted to explore how our throughput changed with the number of concurrently running clients. We believe this is a useful metric because it both indicates how our modifications perform with heavier workloads and because using a larger number of clients presents a greater possibility of transactions overlapping at the storage layer. Transactions are capable of overlapping because leaf fetching occurs based upon the timestamp of leaf creation and not the time at which the leaf enters the storage layer. Since the server is a multi-threaded application this allows for transactions to have overlapping times and to be fetched concurrently.

We opted to explore throughput with 5, 10, 20, 40, and 80 concurrent clients all running on the same machine. For each trial we generated all transactions using the medium and bursty settings from 4.3 with an average transaction size of 3. A graph of the average throughput experienced by both Trillian and our modified version of Trillian are shown in Figure 7.

We found that our modifications to Trillian produced a consistently higher throughput than original Trillian for every number of concurrent clients. For both implementations throughput decreases as the number of concurrent clients grows. We attribute this to an increased demand on the server to process both read and write transactions. More concurrent clients produce more read transactions which causes the server to have to fulfill reads before it can add leaves to be appended to the tree. While the slopes of both graphs are very similar, the difference in throughput does grow with the number of clients. We believe this is likely a reflection of the benefits of our transaction cache. Since a greater number of clients will produce more disjoint transactions, our transaction cache is more likely to contain nodes from previous transactions and thus reduces the number of visits to the storage layer.

4.5 Latency

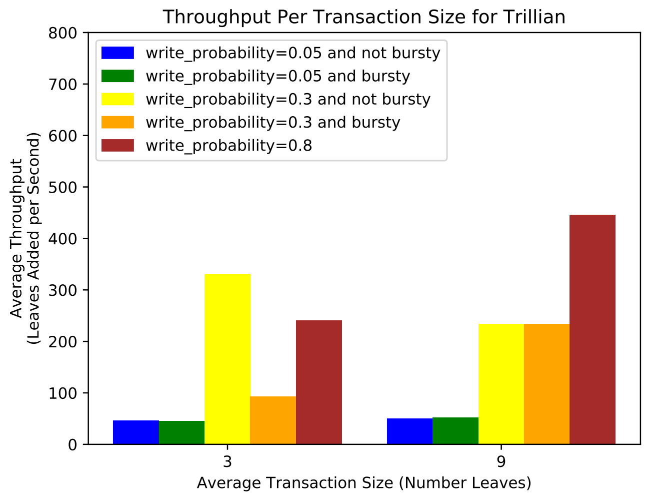
As a final experiment we decided to measure the average latency to integrate a particular leaf into the tree. We believe this is an important metric because this may directly reflect how long a user has an incorrect public key which can be accessed. Any amount of time in which a user's public key is incorrectly listed leaves their associates vulnerable to wrongfully trusting a malicious party. To measure the latency we opted to only consider the latency to be added to the tree once the server adds a leaf to the storage layer. We opted not to consider any overall latency from clients because we believe any interaction from a client would have its latency directly effected by the available network speed, which is beyond the control of our implementation.

We opted to measure the latency as we varied the number of concurrent clients to be 5, 10, 20, 40, and 80. For each trial we generated all transactions using the medium and bursty settings from 4.3 with an average transaction size of 3. A graph of the average latency per leaf experienced by both Trillian and our modified version of Trillian are shown in Figure 8.

Our results indicate that our modifications to Trillian seem to result in nearly double the latency for smaller numbers of clients, while for a larger number of clients the disparity decreases to 1.3 times the latency. We attribute the increased latency for smaller numbers of clients to the need to perform transactions completely atomically as well as increased interaction with the storage layer. Since we computed the average latency for each individual leaf, it is inevitable that when running a transaction the leaves that are queued earlier will have to wait longer for the remaining leaves in the transaction to be processed. While our transaction caching appears effective for over-

Trillian Type	Write Probability	Bursty	Transaction Size (leaves)	Average Throughput (leaves/s)
Original	0.05	No	3	46.21
Modified	0.05	No	3	93.70
Original	0.05	Yes	3	45.23
Modified	0.05	Yes	3	93.14
Original	0.30	No	3	331.13
Modified	0.30	No	3	323.52
Original	0.30	Yes	3	92.67
Modified	0.30	Yes	3	315.65
Original	0.80	Not Applicable	3	240.69
Modified	0.80	Not Applicable	3	481.75
Original	0.05	No	9	49.88
Modified	0.05	No	9	101.21
Original	0.05	Yes	9	51.88
Modified	0.05	Yes	9	94.89
Original	0.30	No	9	233.36
Modified	0.30	No	9	418.52
Original	0.30	Yes	9	233.36
Modified	0.30	Yes	9	393.77
Original	0.80	Not Applicable	9	446.07
Modified	0.80	Not Applicable	9	731.14

Table 2: Table of average throughput achieved while updating the tree for each experiment.



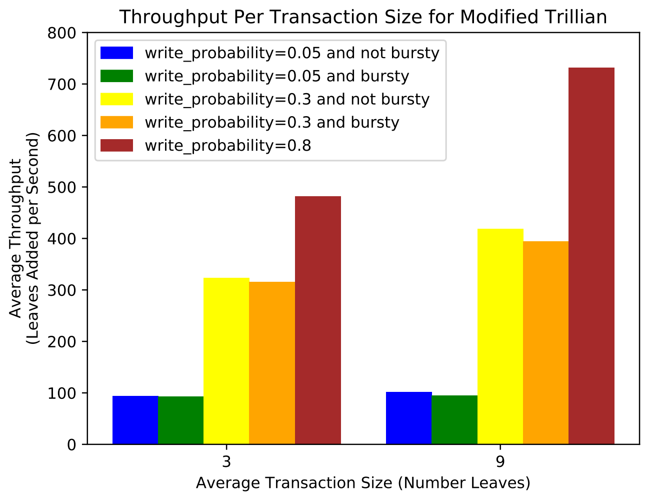


Figure 6: Graphs showing the average throughput achieved while populating the Merkle Tree for Trillian and our modifications to Trillian. In nearly every experiment our modifications to Trillian resulted in greater throughput. However for both implementations throughput increased as the number of leaves to add to the tree increased.

all throughput, any leaf that was ever added to the cache is always waiting longer than if it was added to the tree directly. Additionally, the transaction map introduced a second interaction with the storage layer, likely producing additional delays on individual leaves. For larger numbers of clients we mostly attribute the lack of disparity between the two versions as a ramification of the competition for processor time that developed from running 40 or 80 concurrent clients on the same machine as the signer.

Ultimately while the latency is increasing for individual nodes, we believe the latency that must be endured is acceptable. All of our latency is on the order 7 seconds, even when we believe the machine to be overloaded. Since we expect a user to face far greater latency issues when working over the network or even attempting to swap out a key with their provider, we do not believe our latency numbers are a cause for concern.

5 Related Work

Improving the efficiency of multi-node, transactional updates would have a wide-reaching consequences. As noted in Section 1, many applications utilize Merkle trees like cryptocurrencies [12, 17], filesystems [13], and PKIs

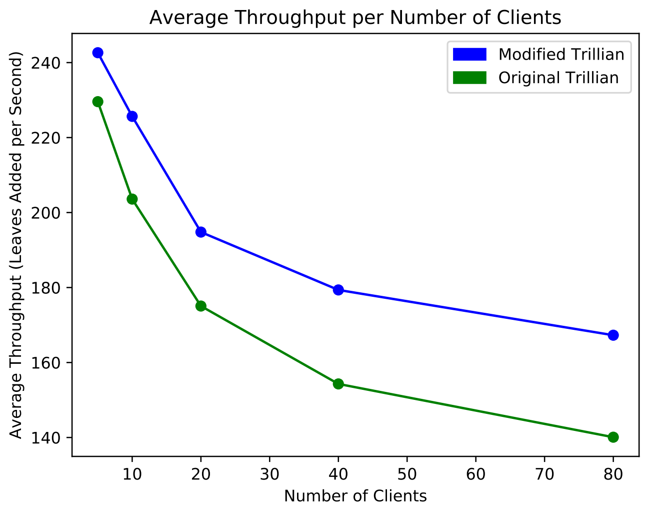


Figure 7: This graph shows how the average throughput varied as the number of clients increased with a fixed transaction size. We compared the modified Trillian implementation and the original Trillian implementation with corresponding fixed batch and transaction sizes while maintaining a 0.3 probability of encountering a write transaction with bursty behavior. The modified Trillian implementation has a higher throughput for a wide range for the number of clients.

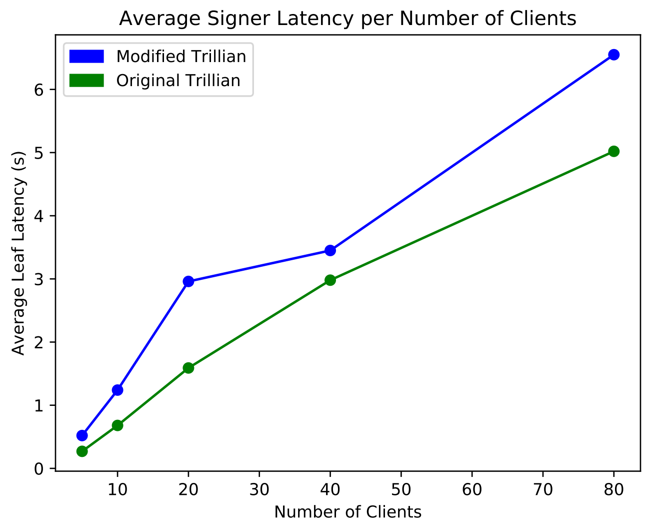


Figure 8: This graph shows how the average latency varied as the number of clients increased. As with the previous figure, transaction and batch sizes were fixed for the modified and original Trillian implementations respectively and 0.3 probability of encountering a write with bursty behavior was also preserved. The modified Trillian implementation has a higher, but overall similar latency to the original implementation.

[8, 10]. With respect to the implementation of efficient Merkle trees, other implementations like Angela [15] decided to focus on the distributed consistency aspect of the problem. This is very important, as the data these trees hold are often too large for a single machine to contain.

Concerto [4] is another project that looks at the application of Merkle trees to a highly concurrent key-value store. Concerto implements an algorithm for the efficient parallelization of membership verification in Merkle trees, which was important to verify the integrity of their store. Scalable verification of Merkle trees is also important in the context of databases, as [6] shows. It is demonstrated that Merkle trees impose significant concurrency constraints and an alternate protocol based on signature aggregation is proposed as a counter to the hierarchical implementation.

6 Future Work

6.1 Sparse Merkle Trees

In this project we focused on implementing transactions on ordered log-based Merkle Trees. For future works we seek to explore efficient transactions on different forms of Merkle trees. One possible target is the sparse Merkle tree, described by Dahlberg et. al in [14]. sparse Merkle trees insert nodes to a particular hash index and thus implementations will not be able to benefit from the reduced hashing presented by Trillian. As a result we suspect efficient solutions will require greater overhead in coordinating transactions to minimize the amount of necessary hashing. However, sparse Merkle trees have the benefit of a much more compact representation, so with large trees, the need for the usage of sparseness grows.

6.2 Network Simulation

While the network traffic delay model was based off of a Poisson model, a Poisson distribution does not accurately reflect all traffic. Other models, like a packet-train and self-similar model [5] could be more appropriate for datacenters. Additionally, while delays were inserted around calls to MySQL read and write transactions, in a production-setting distributed system, delays would be interspersed within the transaction itself. Additionally, MySQL could have cached parts of the transactions we generated during the course of benchmarks, so the delays caused MySQL might have been lower than what could be seen in a datacenter. Investigating these delays could yield a more accurate network model which would strengthen the veracity of any simulations.

As [3] notes, the models that were used to measure the delays for datacenter networks and distributed databases were based solely off of several fixed machine room layours. A highly interconnected datacenter, while one possible model, is not the only possible model for a datacenter. For a truly accurate representation of real datacenter, the network delay used would have to be adapted to the rough topology and scale of the datacenter itself.

7 Conclusion

In this paper we introduced modifications to Google's Trillian Merkle tree implementation to support atomic updates to user public key changes. Our modifications focused on layering virtualized leaves atop the existing log-based Merkle tree and introduced a Transaction Cache for more efficient insertion into the tree. Our evaluation demonstrated for all transaction sizes and number of concurrent clients tested we obtained a throughput comparable or better to what Trillian was previously producing. Although our implementation did introduce as much as twice the existing latency for leaves to be included in the tree, we believe this is still acceptable performance especially since introducing transactions almost guarantees increased latency without considerable changes and optimization to the existing architecture. Our hope is that this project demonstrated not just an effective solution to including atomic transactions in log based Merkle trees, but also can guide future work when attempting to include transactions in even more concurrent and complicated Merkle trees.

8 Acknowledgements

We thank Professor John D. Kubiatowicz for his guidance throughout this project and Professor Raluca Ada Popa for motivating this project and providing readings pertaining to Merkle trees. We thank the RISELab for providing us access to an Amazon EC2 instance.

References

[1] Mysql. `https://www.mysql.com/`.

[2] Our average number of connected devices is at an all time high. "`https://www.verdict.co.uk/average-number-connected-devices-time-high/`".

[3] Reem Abdullah Alshahrani. Delay modeling in data center networks: A taxonomy and performance analysis. Master's thesis, August 2013.

[4] Raghav Kaushik Donald Kossmann Pingfan Meng Vineet Pandey Ravi Ramamurthy Arvind Arasu, Ken Eguro. Concerto: A high-concurrency key-value with integrity. SIGMOD Conference 2017, 2017.

[5] Michela Becchi. From poisson process to self-similarity: a survey of network traffic models. "`https://www.cse.wustl.edu/~jain/cse567-06/ftp/traffic_models1/`".

[6] Kyriakos Mourtadis HweeHwa Pang, Jilian Zhang. http://www.vldb.org/pvldb/2/vldb09-625.pdf. VLDB, August 2009.

[7] Google LLC. Google trust services. `https://pki.goog/`.

[8] Google LLC. How certificate transparency works. `https://www.certificate-transparency.org/how-ct-works`.

[9] Google LLC. Trillian. `https://github.com/google/trillian`.

[10] Joseph Bonneau Edward W. Felten Michael J. Freedman Marcela S. Melara, Aaron Blankstein. Coniks: Bringing key transparency to end users. 2014.

[11] Ralph C. Merkle. A digital signature based on a conventional encryption function. 1988.

[12] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system.

[13] Oracle. Zfs end-to-end data integrity. "`https://blogs.oracle.com/bonwick/zfs-end-to-end-data-integrity`", 2005.

[14] Roel Peeters Rasmus Dahlberg, Tobias Pulls. Efficient sparse merkle trees: Caching strategies and secure (non-)membership proofs. 2016.

[15] Alex Russell-Saw. Angela. "`https://github.com/arussellsaw/angela`".

[16] RSA Data Security. Understanding public key infrastructure (pki). Technical report.

[17] Dr. Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger.