

Fern : An updatable authenticated dictionary suitable for distributed caching

E. Freudenthal, D. Herrera, S. Gutstein, R. Spring, and L. Longpré

University of Texas at El Paso,
{efreudenthal,daherrera,smgutstein,rcspring,longpre}@utep.edu

Abstract. Fern is an updatable cryptographically authenticated dictionary developed to propagate identification and authorization information within distributed systems. Fern incrementally distributes components of its dictionary as required to satisfy client requests and thus is suitable for deployments where clients are likely to require only a small fraction of a dictionary's contents and connectivity may be limited.

When dictionary components must be obtained remotely, the latency of lookup and validation operations is dominated by communication time. This latency can be reduced with locality-sensitive caching of dictionary components. Fern dictionary's components are suitable for caching and distribution via autonomic scalable locality-aware Content Distribution Networks (CDNs) and therefore can provide these properties without requiring the provisioning of a dedicated distribution infrastructure. Competitive approaches require either the sequential transfer of two-to-three times more vertices or the replacement of a greater number of already distributed vertices when updates occur.

Key words: binary trie, authenticated dictionary, distributed systems, content distribution network, Merkle tree.

1 Introduction

The maintenance of consistency between Certificate Authorities (CAs) and access controllers has been a persistent problem in distributed systems. A variety of approaches have been implemented including online validation, limitation of certificate lifetimes, and dissemination of certificate revocation lists. These approaches have complementary advantages, and hybrid implementations are common. For example, by issuing certificates with limited lifetimes, a CA can limit the number of unexpired certificates that access controllers must reject. This *certificate revocation list* (CRL) can be disseminated directly to all access controllers, or to a set of trusted proxies who provide online validation services. The dissemination of complete CRLs can impose onerous communication, storage, and computational costs to access controllers that only reference a small subset of CA's certificates. Similarly, the provisioning of trusted proxies increases the communication and computational cost, and possibly the latency of authorization decisions. Furthermore the determination of whether an online software

system has security properties suitable for secure online transactions is notoriously difficult.

Distributed updatable online authenticated dictionaries based on skiplists[1, 2] and self-balancing trees[3] have recently been proposed as a source of authorization information in distributed systems. These structures disseminate name-to-value mappings that can provide evidence of identity (i.e., that some public key identity K_A is an Id associated with a person named “Alice”) or that a set of certificates has been revoked. Composed cryptographic hashes are embedded within these dictionaries, permitting their distribution via untrusted proxies to access controllers who can efficiently validate mappings or determine their absence. The integrity of any search path within these data structure can be verified with a single public-key root certificate. Furthermore, these partial copies of the dictionary’s structure can be cached and therefore utilized to answer future queries that share common paths. These properties can be exploited by proxies and security-sensitive clients that incrementally obtain and validate portions of their search structure as required to answer queries.

The latency of a search within a distributed authenticated dictionary is dominated by the latency of obtaining vertices in a search path. So, it is useful to minimize these *necessarily serialized* operations. Path length in a search tree corresponds to leaf depth. Expected leaf depth is $U \log_b N$ where b is the tree’s branching factor. U (typically below 2) accounts for the structure’s expected lack of balance.

Modifications to mappings stored within an authenticated dictionary typically cause changes to the dictionary’s search structures. Since components of an authenticated dictionary may be cached, it is desirable if these updates are limited to the ancestors of the changed components and thus require only a few updates. Thus, algorithms that rely on structural modification to achieve good performance (e.g. self-balancing trees) reduce this cacheability. Updates to a randomized skiplist do not result in structural changes unrelated to the updated vertex’s original search path, but the expected number of objects in a skiplist’s search path is $3 \log_2 N$ for a skiplist containing N items.

Fern utilizes a randomized search structure with empirical path lengths of $1.1 \log_2 n$. Given the high latency of inter-host communication in distributed systems, *the constants do matter*, and the number of vertices that must be transferred to lookup a value stored within Fern is comparable to the most aggressive algorithms based upon self-balancing trees and far lower than skip-lists. Like skiplists, but unlike balanced trees, Fern does not require restructuring and thus is also well suited for distributed caching.

2 Fern

As illustrated on the left side of Figure 1, Fern’s internal structure is a binary Merkle-trie with path compression. Like the authenticated authorization framework suggested by Tamassia et. al. in [3], vertices from Fern’s Merkle-trie are suitable for distribution by peer-to-peer (P2P) Content Distribution Networks

(CDNs) to clients and access controllers who construct, maintain, and validate search paths that serve as evidence of authorizations upon which they depend. Following the notation of Martel et. al.[4], we refer to validated search paths as *verification objects* (VO).

ID:value mappings are stored in leaf vertices whose search key is equal to its ID's SHA-1 hash (when used to disseminate set membership, the *value* field can be empty.) Following the model of a Merkle-tree[5], internal nodes also contain SHA-1 hashes of each immediate descendant (hashes are not indicated in Figure 1). Incorporation of hashes within all internal vertices permits access controllers to validate the integrity of individual search paths without obtaining the entire trie. The current root and its hash is stored in a *root certificate* signed with the originator's public key.

A Fern VO for a particular ID contains a root certificate and the search path containing its mapping. If the ID is not stored within Fern, it contains a search path that demonstrates that the referenced ID is not in the trie.

Fern publishes root certificates and vertices using the CoralCDN scalable, self-organizing, and locality-aware content distribution network. This helps to minimize network congestion that arises from distributing vertices and permits a large number of access controllers to be served by a single server with modest computational and network resources.

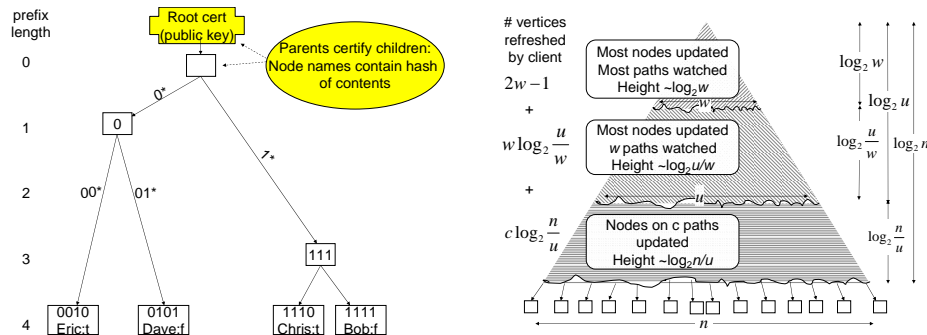


Fig. 1. On left: Fern trie for 4-bit keys containing leaves (0010, 0101, 1110, and 1111). On right: Number of nodes that need to be updated when the number of watched data entries is smaller than the number of updated data entries (see Section 3)

In contrast to Fern, which uses 160-bit SHA-1 hashes, the trie depicted in Figure 1 only uses four-bit search keys. ID:value mappings are stored in leaf vertices with search-keys corresponding to the ID's hash digest. In this figure, the name “Bob” (mapped to f) is depicted as having a hash equal to 1111.

Routing decisions within this trie are made one bit at a time, in decreasing order of significance. Each internal vertex is labeled with the search prefix corresponding to the common prefix shared by all its descendants. Vertices within

tries that have no branching are collapsed as is illustrated by the absence of vertices corresponding to the prefixes 1, 10, 11, and 110.

A client with interest in an ID's mapping stored within Fern obtains the set of trie nodes that comprise the path from the root to the desired leaf. Monte Carlo experiments indicate that leaves within Fern's binary trie are generally at a depth of $1.1 \log_2 n$ where n is the number of mappings stored within the trie, equal to the most aggressive self-balancing algorithms.

Each internal vertex of a Fern authorization trie contains node identifiers (NIDs) of its children. NIDs are used as a locator by clients requesting particular nodes. Like SFS-RO's[6] self-certifying pathnames, Fern's NIDs include cryptographic hashes of the referenced node's contents and are also used to verify the integrity of a nodes obtained from insecure channels.

A Fern CA periodically publishes a signed, time-limited *root certificate*. Using an asymmetric cipher Fern clients and access controllers know the CA's public-key identity and use it to determine the integrity of root certificates.

A VO for a referenced identifier, I , thus contains all Fern-nodes between the Fern-root and I . A full certification path can be included within the payload of a single message transmitted from one Fern client to another. Alternatively, since each internal node of a Fern-trie contains the NIDs of immediate children serving as locators, a Fern client can obtain a certification path for I by obtaining intermediate nodes as it traverses the path from the root toward the leaf node containing I . A search-path that comprises a complete VO can be transmitted directly between clients, or instead be constructed by a client that searches for the node corresponding to the ID's hash.

3 Analysis: Number of Refresh Queries

Fern search-tries are well balanced due to their use of a good hash function (e.g. SHA-1) to evenly distribute a set of elements over an ordered range which is large compared to the number of elements being distributed.

The hash value of a Merkle-tries's leaf and all of its ancestors changes whenever the leaf's contents are updated. In this section, we consider the number of vertices that must be obtained by a client who is monitoring mappings stored within a set of $w = |W|$ *watched* leaves within a dictionary storing a total of n mappings in the event that $u = |U|$ leaves are *updated*. Let $c = |U \cap W|$ be the number of vertices in W whose mappings have changed. Assume that $w \leq u$ (this should be commonly the case.) Monte Carlo experiments support these results, especially as w , u and n become large. A technical report version of this paper [7] provides a more formal analysis and also examines the case where $w > u$.

As illustrated on the left side of Figure 1, our analysis divides the trie into three regions. The upper region, which extends to depth $\lfloor \log_2 w \rfloor$ has approximately w vertices at its (approximate) lower edge. We assume that the hash function uniformly distributes keys, that most of these (approximately) w vertices are ancestors to the w leaves mapping members of W . Since $u \geq w$, it is

likely that most of these (approximately) w vertices members are also ancestors of the members of $|U|$. Thus our client is likely to require updates for all (approximately) $2w - 1$ vertices in the upper region.

The middle region of this trie extends to depth $\lceil \log_2 u \rceil$. Like our analysis of the upper region, it is likely that most vertices above depth $\log_2 u$ contain nodes that are ancestors of the u updated leaves and most vertices in this region are likely to have been updated. However, only w paths of length $\lceil \log_2(u/w) \rceil$ through this middle region are likely to be ancestors of vertices being *watched*, and only $w \lceil \log_2(u/w) \rceil$ of these vertices are likely to be needed by the client.

The lower region extends down from (approximately) level $\lceil \log_2 u \rceil$ and has approximately $\lceil \log(n/u) \rceil$ levels. However, the approximately $c \lceil \log_2(n/u) \rceil$ vertices along the c paths to leaves that are both watched and updated will be obtained by clients. Thus we expect that approximately $(2w - 1) + w \lceil \log_2(u/w) \rceil + c \lceil \log_2(n/u) \rceil$ nodes that will require updating. In practice, the shift from one section to the next may not occur exactly at the same level on every path. The analysis for $w > u$ is symmetric, so the same result holds with a switch between w and u .

We have begun evaluation of Fern upon the Planetlab[8] global distributed testbed. The plot on the left side of Figure 2 conducted upon hundreds of hosts distributed globally provides empirical evidence supporting our analytical model of refresh queries. In this experiment, $k = 2058$, $w = 60$, and $u = 200$. Each host's set of watched mappings was chosen with a distribution corresponding to the routing table of a Kademia[9] DHT. Plots indicate the average number of Fern vertices obtained for clients distributed globally upon Planetlab. As predicted by the analytical model, refresh queries tend to terminate around depths $\log_2 u = 7$ and $\log n = 11$. In these experiments, the average number of vertices obtained by clients from CoralCDN is 154. Our analytical approximation prediction is 146.

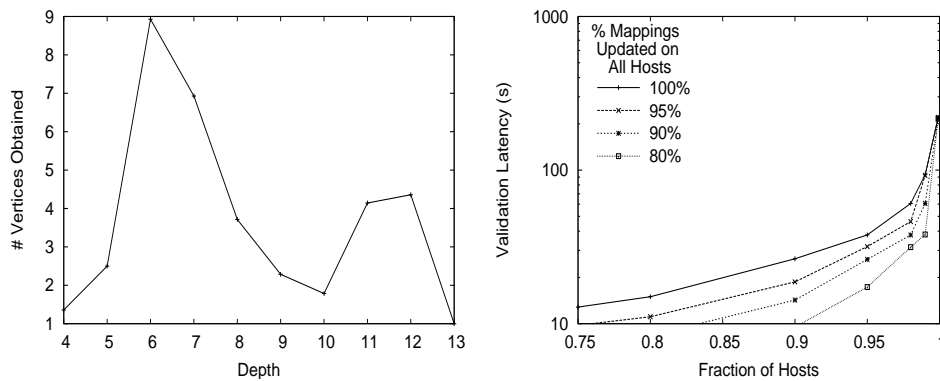


Fig. 2. Measured results from Executions on Planetlab: Left side: Average Depth of Refresh Queries. Right side: Cumulative distribution of refresh query times

The plot on the right side of Figure 2 depicts the cumulative distribution of update latencies for watched mappings in an experiment with 324 hosts distributed in 37 countries. Fern vertices were disseminated using CoralCDN. In this experiment $n = 2048$ and each host watched $w = 37$ mappings chosen with mapping corresponding to its view of a Kademia routing table. Roots were published at five minute intervals, and thirty-two mappings were changed during each interval. Each plot depicts the cumulative distribution of times required for a fraction of the hosts to obtain a fraction of the updated mappings. Note that 0.9 (90%) of the hosts obtained 90% of their 37 watched mappings in ~ 10 s, and 0.95 of the hosts obtained all of their 37 watched mappings in ~ 30 s.

4 Conclusion

We have described a data structure for updatable authenticated dictionary with low maintenance and with better latency than current techniques.

Acknowledgments. This work was supported in part by the U.S. Army Research Laboratory, Survivability/Lethality Analysis Directorate, Information and Electronic Protection Division, Information Warfare Branch.

References

1. Goodrich, M, Shin, M., Tamassia, R. and Winsboro, W.: Authenticated dictionaries for fresh attribute credentials. In Nixon, P., Terzis, S., eds.: *iTrust*. Volume 2692 of *Lecture Notes in Computer Science.*, Springer (2003) 332–347
2. Anagnostopoulos, A., Goodrich, M., et al: Persistent authenticated dictionaries and their applications. In: *Lecture Notes in Computer Science, Proc. Information Security Conference (ISC)*. Volume 2200., Springer-Verlag (2001) 373–393
3. Tamassia, R., Triandopoulos, N.: Efficient content authentication over distributed hash tables. Technical report, Brown University (2005)
4. Martel, C., Nuckolls, G. et al: A general model for authenticated data structures. Technical Report CSE-2001, Stubblebine Labs (2001)
5. Merkle, R.C.: A certified digital signature. In Brassard, G., ed.: *Advances in Cryptology – CRYPTO ’89*. Volume 435. Springer-Verlag (1990) 218–238
6. Fu, K. , Kaashoek, M. F., and Mazieres, D.: Fast and secure distributed read-only file system. *Computer Systems* **20**(1) (2002) 1–24
7. Freudenthal, E. Herrera, D. et al: Fern: An updatable authenticated dictionary suitable for distributed caching. Technical Report 06-45, Computer Science Department, University of Texas at El Paso (2006)
8. Planetlab: (An open platform for developing, deploying, and accessing planetary-scale services) <http://planet-lab.org>.
9. Maymounkov, P., Mazieres, D.: Kademia: A peer-to-peer information system based on the xor metric. In: *Proc. IPTPS02*. (2002)