A Peek Into Merkle^2

Rasmus Dahlberg

rasmus.dahlberg@kau.se

1 Introduction

Computer security is largely about managing system complexities. A system complexity that is not transparent is difficult to manage in a trustworthy way. Therefore, a secure and trustworthy system will likely be transparent. Examples of transparency can range from having open designs to disclosing certain data.

The practise of disclosing certain data to public scrutiny is often referred to as *transparency logging*. The basic idea is to make the data available in a *public log*. If the system in question requires that data is publicly logged before use, anyone can detect malicious data by inspecting the log. Today's web browsers require that website certificates are transparency logged so that the legitimate domain owners can detect if anyone tries to impersonate their domains. More generally, a transparent log is helpful to *discover data*. Other use-cases than Certificate Transparency include discovery of routing announcements, executable binaries, source code, YouTube videos, tax declarations, and documents of ownership.

1.1 Constructing Transparent Logs From Merkle Trees

The construction of a transparent log is complicated by the observation that the operator should not be a third-party that is trusted blindly. A transparent log should instead be *cryptographically verifiable* to ensure that no trust is misplaced.

What makes a transparent log verifiable is an underlying *authenticated data structure*. The cryptographic properties, or *proofs*, that the authenticated data structure supports shape what the log can be used for. Examples of proof types:

Membership: proves that some data is in the log.
Non-membership: proves that some data is not in the log.
Append-only: proves that nothing in the log has been removed or modified.

There are a wide range of primitives that can be used to construct authenticated data structures. Simple ones are usually based on Merkle trees. A Merkle tree is a tree data structure where all nodes compute a hash value. Each interior node concatenates the values of its children, hashing them to form its own value. This process repeats from the bottom and up until a single *root hash* remains. A proof is accepted by a verifier if it facilitates reconstruction of a known root hash. Verifiers *gossip* known root hashes to ensure that they see the same log.

The catch is that today's transparent logs (based on Merkle trees) cannot produce efficient membership, non-membership, *and* append-only proofs. This is a significant limitations that complicates *monitoring* of the log's data. A *Merkle prefix tree*, which supports efficient (non-)membership[1] proofs, requires the data

[1] *(Non-)membership* is a common shorthand for *membership and non-membership*.

owner to inspect every root hash to discover all their data. A *chronological Merkle tree*, which supports efficient membership and append-only proofs, requires the data owner to download the entire log. Monitoring thus involves linear work.

Key Transparency projects usually use a data structure that is similar to the Merkle prefix tree. Certificate Transparency instead uses a chronological Merkle tree. The approach of Merkle^2 is to intertwine the two tree types in a new way.

1.2 Importance

Today's transparent logs suffer from the monitoring problem. It does not scale that every data owner needs to inspect an entire chronological tree or every root hash in a prefix tree. This introduces unfortunate trade-offs. For example, trusted third-party monitoring services are used in Certificate Transparency. Slow update frequencies are used in Key Transparency projects. A possible solution is to start using an authenticated data structure that supports efficient (non-)membership *and* append-only proofs. Such solutions exist on paper, but the involved cryptographic primitives are too expensive in practise. The premise of Merkle^2 is interesting because the involved building blocks are not as costly.

1.3 Limitations

Merkle^2 requires a *signature chain* construct to make monitoring efficient. It is assumed that the data owner can keep a secret signing key private. It is also assumed that the data owner manages to claim the identifiers that they want.[2]

There is no discussion on what happens if a data owner's secret signing key is leaked or compromised. There is also a lack of discussion regarding privacy. Privacy-friendly gossip protocols in Certificate Transparency tend to rely on infrequent update intervals. Merkle^2 has a frequent update interval *by design*. While frequent updates enable low-latency applications, it complicates gossip.

1.4 Contributions

The authors proposed a new way to intertwine two types of Merkle trees as a forest. The resulting authenticated data structure is named *Merkle square*. Auditors can verify efficiently that the forest they observed in the past remains a subset of the forest they observe at the present. The log can also update the forest efficiently using an amortization trick. The straw man design on how to monitor the forest is not more efficient than a normal prefix tree. What makes it more efficient is the introduction of signature chains. Signature chains allow the data owner to make assumptions about *which new prefix trees need to be monitored in the forest*. As long as the secret signing key is kept private, a large majority of prefix trees can be skipped. End-users verify efficient Merkle^2 look-up proofs by recreating a *forest digest* and checking that the signature chain is valid. The proposed system has a proof of concept implementation with benchmarks.

[2] An identifier is claimed by proposing the first identifier-value pair.