**Signature Scheme Based on Hash Trees**

**Introduction**

The following paper aims to explain a method in which users can achieve integrity and authentication for their messages. This scheme includes multiple parties, each party verifying each other to ensure the secure and correct function of the algorithm. Compared to other means of generating digital signatures, such as RSA, this approach requires multiple systems for signature validation and keeping one another accountable for any mishaps. This also means that the method becomes completely unreliable and insecure when two out of the three parties involved become compromised or are impersonated by ill-intentioned people.

**Related Work**

A close up of a map

Description automatically generated In order to understand the way the systems work it is crucial to understand the inner workings of hash functions and aggregate data structures, such as Merkle Trees (or more colloquial, Hash Trees). A well-built hash function is a one-way function which maps a specific input to a seemingly random output, changing one bit of the input leading to a completely different result. A Hash Tree is a data structure built from nodes that have the property that their associated values are obtained from hashing the associated values of all their children. Leaf nodes contain preliminary values, having no children to gain hashes from. Another concept used is that of a hash chain – a list of hashes necessary for obtaining the root value of a Hash Tree. An example of such a chain can be found below.

To ensure the security of the mechanism, the usage of a secure hash function is mandatory. If this is not the case, a number of users can find collisions, rendering impersonation attacks possible if a high enough number of malicious end-users band together.

Another requirement is that secure pseudorandom generators are used for private key generation. As we will see, this exacerbates the impersonation problem if the correct means of generating random keys are not used.

A screenshot of a cell phone

Description automatically generated**How It Works**

There are three parties involved: the signer, the server and the repository. The signer is the user of the scheme, the entity that wishes to provide integrity and authentication to its messages. The server provides the interface through which different users access the system, adding a layer of authentication. Finally, the repository has the last word in regard to which signatures are valid or not, but also providing means of starting an audit of the signatures saved.

Each interaction between a part of the system with another is followed by presenting a hash chain from the tree along the root value, allowing the receiver to calculate the root value itself. The next step is comparing the previous computed value with the one received, and finally, with the public one available either on the repository-side (the previous summaries) or the server-side (the signers’ public keys). These interaction introduce the validation from all the parties, negating all forgeries done by a single malicious user, server, or repository.

The signer needs a number of private keys, which are randomly generated and serve as the leaf nodes in the hash tree the signer possesses. These private keys are used sequentially for signing messages. After being used, a private key is published for audit reasons. The public key is calculated by generating a hash tree with the private keys as leaf nodes and building it from there. After it is generated, the public key is used to authenticate all the transactions the respective user starts.

The visual representation of the hash tree can be seen to the left.

The server holds a rather similar hash tree in use. The leaf nodes are obtained by hashing a key counter and the last message a user signed. For a new user, a nonce can be used instead of the last message’s hash.

These 2 values are changed whenever a user successfully signs a new message. The value at the top of the tree is calculated after each update of a leaf node and is used for validation. This value is called a “summary” and all the valid values are appended to a read-only list inside the repository. There are two ways a summary can be updated. One is after each modification of a leaf node. The other one is after each user has signed a message. The first one was used in this implementation, although it introduces performance issues if the system is heavily loaded. The second one introduces parallel programming and multi-threading issues which are not the scope of this paper, thus the slower variant was used.

The last component of the scheme is the repository. It is the last layer of validation and it is the only layer that is not using a hash tree to hold its data. It can also include the verifier entity for extra security. Both the server and the signer submit queries to it in order to obtain the last summary published against which to compare the results calculated through hash chains.

**Result**

Through experiment it was found that a similar scheme can sign and validate 8192 signatures in roughly 16 seconds on a single i5 2.3 GHz processor. Due to the distributed nature of these means, higher performance could be achieved linearly with the number of processors. This method is safe as long as only one of the parties is affected. As long as there are no vulnerabilities in the PRGs and hashing functions used, even a majority of users being compromised doesn’t leave the system open for exploitation due to computation difficulties. Having a large number of private keys per user and a large number of users per server also seems to alleviate such impersonation attacks. As long as the server and repository are not compromised at the same time, there is no way an attacker can break the scheme due to the extra-party check in place. Some problems could arise when a user published all his private keys, requiring new ones. It is mandatory that no key is used more than once so extra precautions are taken so that once a key is used it will never appear in a signer’s hash tree again.

The load can, theoretically, be split per thread. The users can be assigned to a specific task, using some hash function or other heuristic. Other approach is to have a single repository and multiple servers.

Although the concept presented in this paper appears hard to crack in principle, due to its complexity, some bugs, and thus vulnerabilities, can appear in implementations. When multi-threaded environments are taken into account, the method quickly becomes prone to human error. Its scalable nature and fault tolerance are two big pluses of this approach. Its obvious downside is the difficulty that implementing it and releasing it for public use imply.

**Conclusion**

In conclusion, this BLT-based scheme improves its original design, taking the concept of one-time time-bound keys of its predecessor and deriving an improved one-time key distributed system that is both secure and scalable. The downside is that it is highly susceptible to vulnerabilities in its source code if badly implemented and the security of the primitives used.