Location Certificate Transparency for third-party-verifiable location proofs 

Pedro Carvalho[0000−0002−7761−9157], Samih Eisa[0000−0003−0972−4171], Leonardo S. Rocha[0000−0002−2608−1844], and Miguel L. Pardal[0000−0003−2872−7300]

INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal
{pedro.matias.carvalho,samih.eisa@inesc-id.pt,miguel.pardal}@tecnico.ulisboa.pt
LAGIC, Universidade Estadual do Ceará, Brasil
leonardo.sampaio@uece.br

Abstract. The vast expansion of mobile and ubiquitous computing has led to the daily and widespread use of Location-Based Services. These services detect locations and are trusted by applications. However, the services may fail or be subject to fraud. This is a problem for critical applications that rely on accurate location data. One approach to deal with location spoofing is to require the use of Location Certificates (LCerts). LCerts allow any party to independently verify the location claim and digital signatures. However, by themselves, they do not provide trusted storage and verification at a later date. In this work we propose a Location Certificate Transparency framework to provide accountability to all the entities involved in location proofs: provers, witnesses and verifiers. We developed a tamper-proof ledger based on Merkle-Trees and provide APIs for the storage, retrieval and verification of LCerts by monitors and auditors, to further protect end-users from malicious parties.

Keywords: Location Certificate · Certificate Authority · Digital Certificate · Location · Certificate Transparency · Security.

1 Introduction

Information and communication technologies are growing with the expansion of mobile and ubiquitous computing. The technology is affordable, and with the widespread availability of GPS and Wi-Fi receivers, the use of Location-Based Systems (LBS), where the location is captured by the device and then trusted by an application, is now very common. However, in these systems, the location can be forged and since some apps critically need the real device's location we need to ensure that the location is a certified attribute, as proposed by Saroiu and Wolman [8], and implemented in systems such as APPLAUS [9], CREPUSCOLO [1]. SureThing [4] goes beyond a single system and provides a framework for Location Certificates (LCerts). In SureThing, provers issue a Location Claim (LClaim), that can then be endorsed by multiple witnesses. The verifier may accept or reject a LClaim. If it accepts the LClaim, it will generate a Location Certificate (LCert). It is worth mentioning that these documents – Location

Claim, the endorsements from the witnesses and the Location Certificate – are all digitally signed documents. Each entity has a private key, and a certificate for the public key. A certificate is a document that is used to prove the ownership and assert a set of attributes related with the owner identity, including the public key. A certificate also contains the digital signature of an entity that verified the certificate's contents, which is called a Certificate Authority (CA). If we guarantee that the signature of the certificate is valid and the software examining the certificate trusts the CA that issued the certificate, then we can use this public key to verify signatures made with the corresponding private key and, consequently, we can securely communicate with the owner of the certificate.

There are different attestations taking place: the CA attests the identity of the signer, whereas the Verifier attests the location claim to produce an LCert. However, the LCert, by itself, does not provide trusted storage and verification at a later date. In this work, we propose to bring what is called Certificate Transparency in the World Wide Web environment to the SureThing framework in the form of *Location Certificate Transparency*. Thus, the verifiers, when receiving a Location Claim issued by a prover, generate the LCert and submit it to a tamper-proof, append-only verifiable public log so that it can be re-checked later, when needed.

The public log has other types of clients, *monitors* and *auditors*. Monitors contact the log servers periodically to watch for suspicious LCerts. A suspicious LCert can be a certificate that says that a user is in a place at a specific time, and then another certificate says that the same user is in another place, which is physically impossible to happen in the time interval between certificates. An example can be a certificate saying that a user is in Portugal and, one minute later, another certificate says the same user is in Sweden. We can then conclude that at least one of the certificates is suspicious, if not both. Auditors can verify that a log is consistent, has added the new entries and was not corrupted retroactively by someone inserting, deleting or modifying one or more LCerts. These two clients can communicate between each other, also known as *gossip*, in order to detect inconsistent views of the logs.

The main objective of this work is to help to detect the tampering of location certificates and also suspicious LCerts as it provides a transparent view of all the location certificates that are being issued and their contents. We developed the ledger and used it for storing location certificates with integrity guarantees so that we can use them for future revalidation. Another objective is to enable the classification of witnesses as trustworthy or not, which will be used to further validate a presented LCert.

The remainder of the document is structured as follows. The most relevant background work is discussed in section 2. Section 3 describes the architecture of the proposed solution and section 4 its implementation. Section 5 shows the system evaluation results. Finally, in section 6 we conclude the paper.

2 Background

Certificate Transparency is something that is already deployed in the World Wide Web environment for the use of HTTP over TLS (HTTPS) in this section, we present the system used for logging TLS certificates where we describe the most important concepts. Also, in order to assure the correctness of Certificate Transparency, we also present some Gossip protocols.

Laurie et al. [5] described a system for publicly logging Transport Layer Security (TLS) certificates as they are issued. In this system, anyone can audit Certificate Authority (CA) activity to see if suspect certificates are being issued or even review the certificate logs themselves. The goal is to have logging as one of the criteria for a valid certificate which means that clients will eventually refuse to accept certificates that are not logged. Therefore, CAs are forced to log every certificate that they issue. When a certificate is submitted to the log, the log must return a Signed Certificate Timestamp (SCT), which corresponds to the log's promise of incorporating the certificate in the Merkle Tree within a Maximum Merge Delay (MMD). The append-only property of these logs is achieved using Merkle Hash Trees that are described by Merkle [6] and by Mykletun et al. [7] and illustrated in figure 1. Each version of the log is a superset of the previous version. This data structure is helpful as provides evidence to trust the logs: if a log tries to show different views to different people, this can be detected by comparing tree roots and consistency proofs.

Types of Log Clients In this protocol, there are four different types of clients: the *submitters*, the TLS *clients*, the *monitors* and the *auditors* .
Submitters submit certificates to the log and can use the returned SCT to construct a certificate or use it directly in a TLS handshake.
TLS clients receive SCTs alongside or in server certificates, and not only do they make a validation of the certificate and its chain, but also validate the SCT by computing the signature from both the SCT data and certificate, and verify the signature using the correspondent log's public key.
Monitors are the entities that observe logs and check if they behave correctly. Monitors need to inspect every new entry in each log and they may keep copies of entire logs. The protocol that they use is the following:
1. Fetch the current Signed Tree Head (STH).
2. Verify its signature.
3. Collect all the entries in the tree corresponding to the STH.
4. Confirm that the tree made from the collected entries produces the same MTH as the one in the STH.
5. Get the current STH. Repeat until the STH changes.
6. Check the STH signature.
7. Collect all the new entries in the tree corresponding to the STH. If these entries are unavailable for a long period period, then it means the log is misbehaving.
8. Either:
    1. Verify that the updated list of all entries generates a tree with the same MTH as the one in the new STH.

Or, if it is not keeping all log entries:

2. Get a consistency proof for the new STH with the previous STH.

3. Verify the consistency proof.

4. Check if the new entries generate the corresponding elements in the consistency proof.

9. Go to Step 5.

Auditors typically perform two functions: they can verify any pair of STHs from the same log by requesting a consistency proof, and also they can request Merkle Audit Proofs to check if a specific certificate is included in the log or not.

Next, we briefly describe the Merkle Tree Hash structure, the Audit procedure and the Consistency procedure. Intuitively, we see how to build the tree, and how to both an audit and a consistency proof. A thorough description of Merkle Trees can be found in [6,7].

Merkle Hash Trees The Merkle Tree Hash's input is a list of entries that are going to be hashed using SHA-256. Thus, forming the leaves of the Merkle Hash Tree. The output is going to be a 256-bit Merkle Tree Hash (MTH).

Given an ordered list of n entries D[n] = {d(0), d(1), ..., d(n-1)}, the Merkle Tree Hash(MTH) is defined as follows:
If the list is empty, then the MTH is the hash of an empty string:

$$MTH(\{\}) = SHA - 256()$$

If the list has one element (also known as a leaf hash), then the MTH is:

$$MTH(\{d(0)\}) = SHA - 256(0x00 \parallel d(0))$$

Finally, for n >1, let k be the largest power of two smaller than n. The Merkle Tree Hash of an n-element list D[n] is then defined recursively as

$$MTH(D[n]) = SHA - 256(0x01 \parallel MTH(D[0:k]) \parallel MTH(D[k:n]))$$

where $\parallel$ stands for concatenation and D[k1:k2] represents the list {d(k1), d(k1+1),..., d(k2-1)} of length (k2 - k1).

Merkle Audit Proofs A Merkle audit path is the list of missing nodes required to compute the nodes leading from a leaf to the root of the tree. If the root computed from the audit path and the true root are a match, then the audit path is proof that the leaf exists in the tree.

Given an ordered list of n entries to the tree, D[n] = {d(0), ..., d(n-1)}, the Merkle audit path PATH(m, D[n]) for the input d(m), 0 <= m <n, is defined as follows:
The path for the single leaf in a tree with a one-element input list D[1] = {d(0)} is empty:

$$PATH(0, \{d(0)\}) = \{\}$$

For n >1, let k be the largest power of two smaller than n. The path for the element d(m) in a list of n >m elements is then defined recursively as:

$$PATH(m, D[n]) = PATH(m, D[0:k]) : MTH(D[k:n])$$

for m <k; and

$$PATH(m, D[n]) = PATH(m-k, D[k:n]) : MTH(D[0:k])$$

for m >= k,
where : stands for concatenation of lists and D[k1:k2] represents the list {d(k1), d(k1+1),..., d(k2-1)} of length (k2 - k1).

Example of audit proofs

Figure 1 shows an example of a Merkle tree. For example, the audit path for d0 is [b, h, l] because, if we have d0, then we have a. To build g we need b. Then, we need h to build k and finally we need l to generate "hash". Another example is the following: the audit path for d6 is [i, k]. This time we have d6 which means we also have j. To build l we need i and finally to generate "hash" we need k.

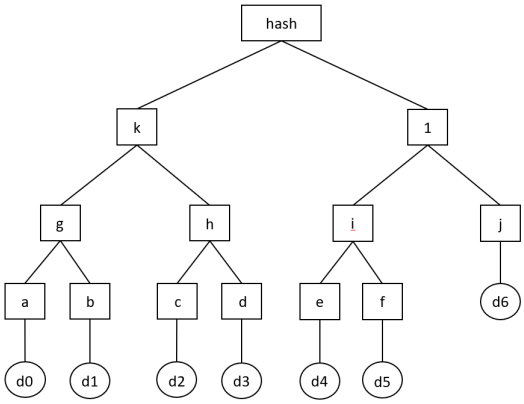

Fig. 1. Merkle Tree Example, adapted from [5]

Merkle Consistency Proofs A Merkle Consistency Proof allows you to confirm the tree's append-only property. The most recent version of the tree includes everything that was in the earlier version, in the same order, and all new elements come after the elements in the older version. If we consider m as the number of elements in the older version and n the number of elements in the most recent version, where m <= n, we can say that a consistency proof must contain a group of intermediate nodes enough to confirm MTH(D[n]), such that a subgroup of the same nodes can be used to verify MTH(D[0:m]).

The Merkle consistency proof PROOF(m, D[n]) for a preceding Merkle Tree Hash MTH(D[0:m]), 0 <m <n, considering an organized list of n tree entries, D[n] = {d(0), ..., d(n-1)}, is defined as:

$$PROOF(m, D[n]) = SUBPROOF(m, D[n], true)$$

The subproof for m = n is empty if the boolean is set to true as it indicates that there were no changes made to that particular subtree, implying that its MTH is known:

$$SUBPROOF(m, D[m], true) = \{\}$$

Otherwise, the subproof for m = n is the MTH(D[0:m]):

$$SUBPROOF(m, D[m], false) = \{MTH(D[m])\}$$

For m <n, let k be the largest power of two smaller than n. The subproof may be defined recursively as it follows:
If m <= k, the right subtree entries D[k:n] only exist in the current tree. We prove that the left subtree entries D[0:k] are consistent and add a commitment to D[k:n]:

$$SUBPROOF(m, D[n], b) = SUBPROOF(m, D[0:k], b) : MTH(D[k:n])$$

If m >k, the left subtree entries D[0:k] are identical in both trees. We prove that the right subtree entries D[k:n] are consistent and add a commitment to D[0:k]:

$$SUBPROOF(m, D[n], b) = SUBPROOF(m{-}k, D[k:n], false) : MTH(D[0:k])$$

where : stands for concatenation of lists, and D[k1:k2] represents the list {d(k1), d(k1+1),..., d(k2-1)} of length (k2 - k1).

The use of this algorithm can give clear evidence that a log is consistent, meaning that there are not modified certificates in the log and that it has not been branched before.

Example of consistency proofs
Figures 2 and 1 show an example of a Merkle tree that was built in incremental steps to show an example of consistency proof. The consistency proof between hash0 and hash is [c, d, g, l] where c and g are used to verify the old Merkle tree and d and l are additionally used to show that the new tree is consistent with the old tree.

2.1 Certificate Transparency Gossiping

CT data replication, usually designated as *gossiping* is required in the security model of CT even though it does not have any significant deployment yet. Therefore, Chuat et al. [2] proposed some Gossip protocols to detect anomalies without losing privacy and with a small overhead. The general framework of these protocols is shown in Figure 3. As we can see, in the beginning, the server submits the certificate to the log and receives the correspondent SCT. Then,

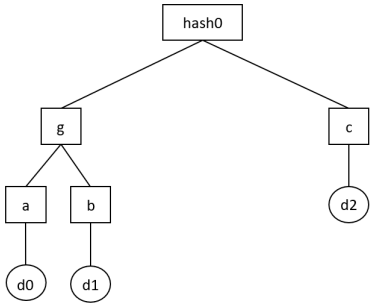

Fig. 2. Merkle Tree in first state, adapted from [5]

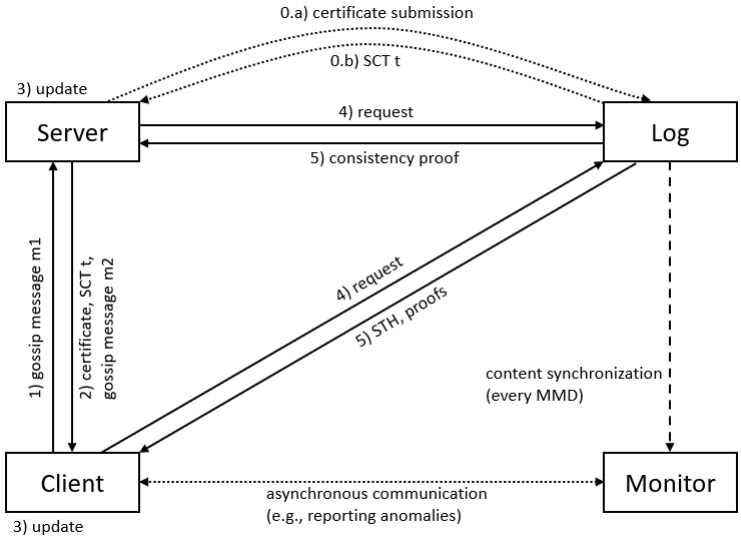

Fig. 3. General Framework for both protocols, adapted from [2]

the auditor (client) gossips the first message, m1, and the server responds with the certificate, the correspondent SCT and a gossip message m2. The content of these messages is what varies in both protocols. In the protocol where they only exchange STHs, the message is simply a valid STH. In the second protocol, they exchange STHs and consistency proofs, so the content of these messages consists in m = (Sa,Sb,Pa,b) where Sa is the STH of a tree with size a, Sb is the STH of a tree with size b and Pa,b is the consistency proof between tree sizes a and b. After this exchange of messages, both the client and the server update their local state. Depending on the content of the received message, they may

need to request a proof from the log to conclude this update. If an anomaly is discovered, the auditor reports it to a monitor.

If we have a system with multiple logs, we can execute the protocol in parallel for each log.

3 Location Certificate Transparency

3.1 Architecture

Figure 4 provides an overview of the interactions between the main entities that contribute to the issuance of a location claim and the storage of a location certificate.The entities together create the Location Certificate Infrastructure (LCI) that we need to achieve the required transparency and security.

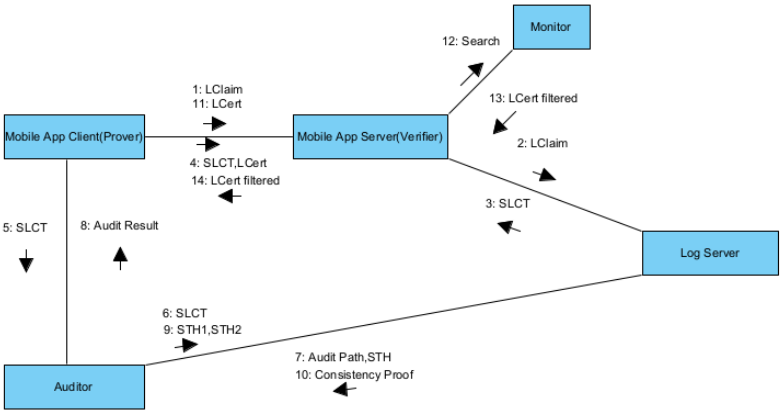

Fig. 4. Location Certificate Infrastructure

Log Server An append-only public log server (ledger) that stores logs of the Location Certificates (LCerts) so that everyone can audit them when needed. The certificates are created by the prover and sent out to the verifier and then stored into the log server. However, they are not stored immediately, the log server returns a Signed Location Certificate Timestamp (SLCT) that acts as a promise to say that this specific certificate will be stored in the log within a fixed time interval called the Maximum Merge Delay (MMD). Everything done by the log server will be verified by other trusted entities such as monitors and auditors;

Monitor A service that, continuously, watches for suspicious location certificates. It contacts the log server and downloads an entire copy of the log. It also verifies the consistency between the versions of the log. Finally, it may also be used as a search mechanism that retrieves location certificates for a criteria, for example, all location certificates emitted by a particular device. If a prover issues an invalid LClaim, monitors will detect it and nothing bad will happen to the system.

Auditor A service that verifies if the log is behaving correctly and is consistent.It sends to the log server two Signed Tree Heads (one for an old version of the Merkle tree of location certificates and one for a newer version) and the log server calculates the consistency proof and sends the result back to the auditor. The Signed Tree Heads can be obtained through the execution of gossip protocol between the auditor and the monitor where they exchange Signed Tree Heads and can check if a particular location certificate appears in the log.

Verifier An entity that is authorized by the Location-Based-Service (service provider) to verify locations claimed by the provers. It can be a mobile user in the vicinity of the claimer or a stand-alone service managed by the LBS.

Prover A mobile user who claims a certain location and subsequently has to prove the claim's authenticity. It receives one or more location endorsements from its neighbour devices (witnesses) when it visits a site. The prover then submits the received location endorsements to the LBS as part of the location claim/request.

Certificate Authority (CA) An authentication service responsible for validating the identities of the prover, verifier and witnesses that involved in the creation of the location certificates.

3.2 Interactions

Figure 4 illustrates the interactions between the entities of the Location Certificate Infrastructure (LCI). First, the mobile app client (prover) creates a location claim and sends it to the verifier. If the verifier accepts it, it generates a location certificate and submits it to the log server and then receives an SLCT, a timestamp promise from the log server to store the location certificate. Then, the verifier returns to the prover the location certificate and the also the received SLCT. A prover may also want to know if a specific SLCT that he has is already stored in the log server. The prover sends his SLCT to the auditor and the auditor forwards it to the log server. The log server calculates the audit proof and returns the result (audit path and the STH) to the auditor so that he can check if the log is telling the truth. If everything is correct, the auditor returns the audit result to the prover. Figure 5 shows the sequence of interactions between the entities.

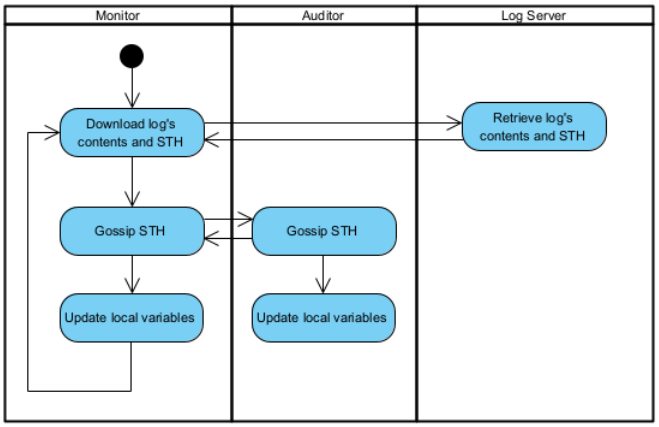

Fig. 5. Background Work

4 Implementation

The entities of the Location Certificate Infrastructure (LCI) are implemented as services using the Spring framework [3]. We used PostgreSQL as main database for storage and all the communications were done over REST calls using the google protocol buffers (protobuf) as payload and data-encoding format and the HTTPS (Hypertext Transfer Protocol Secure) to protect the privacy and integrity of the exchanged data between the entities.The structures of the messages are defined as part of the SureThing framework libraries [4].

5 Evaluation

In this section, we present practical experiments made to asses the efficiency and performance of the developed LCI. We used Hot Spots that show the time spent on each method during the tests as our metric to see what methods are taking the most of the time. The tests included creating 100 and 1000 Location Certificates, respectively, and store them all into the ledger. Then, the prover creates another LCert with a specific value on one of its fields and sends it to be stored. After that, we tested both 1) audit proof and 2) consistency proof. Finally, we make three queries to the monitor, where the first two return all the LCerts created and the third one just returns the LCert created after. It is also worth mentioning that for every Maximum Merge Delay (MMD) the Ledger emits a new STH and the Monitor downloads the new log's contents if they exist and gossips with the auditor to check if the ledger has lied or not.

The following values were taken using an AMD Ryzen 9 5900X 12-Core Processor 3.70GHz.

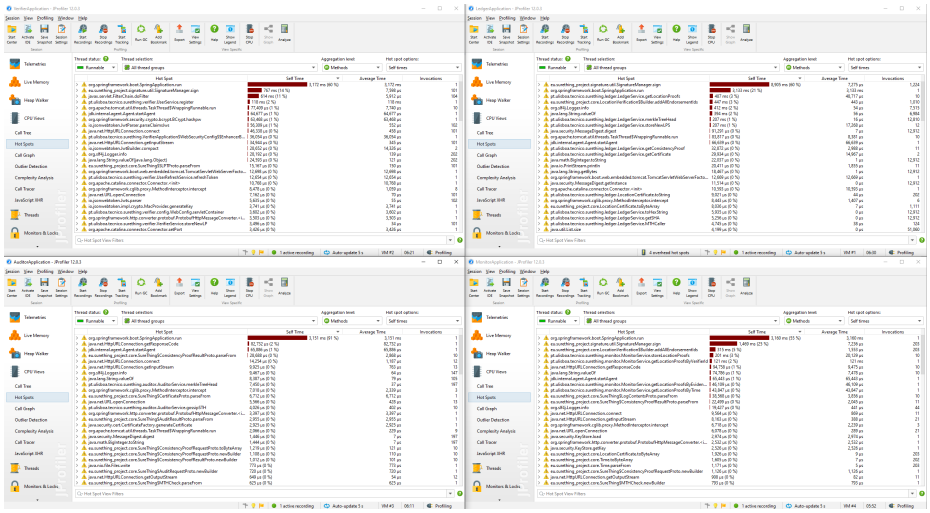

Fig. 6. Hot Spots for 100 LCerts

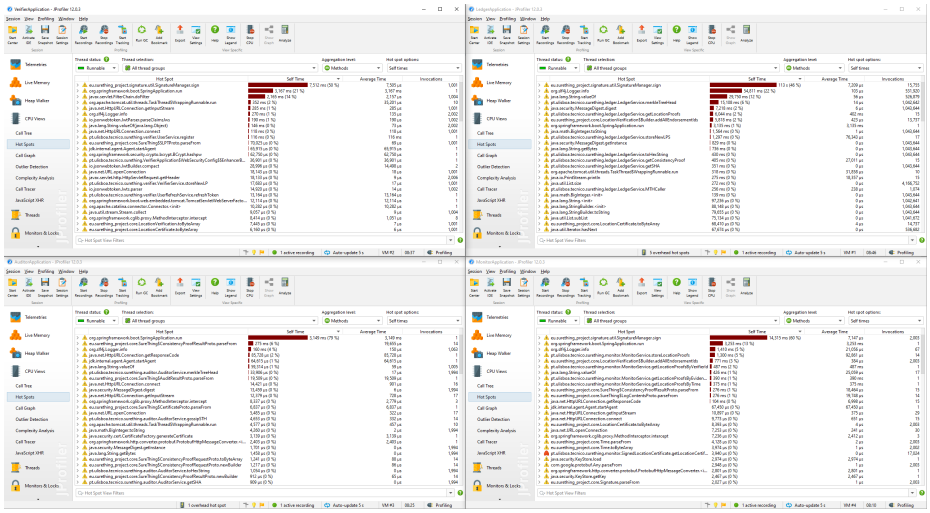

Fig. 7. Hot Spots for 1000 LCerts

5.1 Hot Spot

Looking for both figures we can observe that as the size of the sample grows, the more relative time we spend on signing objects which lets us conclude that the algorithms we implemented about the merkle trees are not the bottleneck of the system.

6 Conclusion

In this paper we studied the current state of certificate transparency in the HTTPS ecosystem and its components. Then, we proposed a solution in section 3 that consists of some entities, each with its own functionalities, and together they work to form an infrastructure that provides location proof transparency. We believe that our solution will help to check the validity of the location certificates that are being issued as they are stored in a public log. Thus, we can be more secure when using those location certificates for smart tourism or medical appointments, as these two are the current use cases of the SureThing project.

Acknowledgements

This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UIDB/50021/2020 (INESC-ID) and through project with reference PTDC/CCI-COM/31440/2017 (SureThing).

References

1. Canlar, E.S., Conti, M., Crispo, B., Di Pietro, R.: Crepuscolo: A collusion resistant privacy preserving location verification system. In: 2013 International Conference on Risks and Security of Internet and Systems (CRiSIS). pp. 1–9. IEEE (2013)
2. Chuat, L., Szalachowski, P., Perrig, A., Laurie, B., 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 (Sep 2015). https://doi.org/10.1109/CNS.2015.7346853
3. Cosmina, I., Harrop, R., Schaefer, C., Ho, C.: Pro Spring 5: An in-depth guide to the Spring framework and its tools. Apress (2017)
4. Ferreira, J., Pardal, M.L.: Witness-based location proofs for mobile devices. In: 2018 IEEE 17th International Symposium on Network Computing and Applications (NCA). pp. 1–4. IEEE (2018)
5. Laurie, B., Langley, A., Kasper, E.: Certificate Transparency. RFC 6962, RFC Editor (2013). https://doi.org/10.17487/RFC6962, `https://www.rfc-editor.org/info/rfc6962`, backup Publisher: RFC Editor Published: Internet Requests for Comments
6. Merkle, R.C.: A digital signature based on a conventional encryption function. In: Conference on the theory and application of cryptographic techniques. pp. 369–378. Springer (1987)
7. Mykletun, E., Narasimha, M., Tsudik, G.: Providing authentication and integrity in outsourced databases using merkle hash trees. UCI-SCONCE Technical Report (2003)
8. Saroiu, S., Wolman, A.: Enabling new mobile applications with location proofs. In: Proceedings of the 10th workshop on Mobile Computing Systems and Applications. pp. 1–6 (2009)
9. Zhu, Z., Cao, G.: Applaus: A privacy-preserving location proof updating system for location-based services. In: 2011 Proceedings IEEE INFOCOM. pp. 1889–1897. IEEE (2011)