RUHR UNIVERSITÄT BOCHUM

RUHR-UNIVERSITÄT BOCHUM

Signed Certificate Timestamps: A Never-Failing Promise?

Luis Wengenmair

Bachelors's Thesis – November 8, 2024
Chair for Security of Software

1st Supervisor: Prof. Dr. Kevin Borgolte
2nd Supervisor: Dr. Muhammad Talha Paracha

Contents

Abstract

Since its introduction into the web's ecosystem, Certificate Transparency (CT) logs have played an instrumental role in fostering greater trust in Certificate Authorities (CAs). Prior to this, CAs were organizations that operated with considerable autonomy, largely unaccountable to any external oversight. The presence of CT logs ensures that users are not required to place unconditional trust in CAs, as these logs document the activities of the CAs and enable the detection of discrepancies regarding the issuance of digital certificates. One might reasonably question whether there is any reason to trust the controllers of CAs.

The verifiability of CT logs by any individual or entity wishing to monitor or audit them provides a rationale for trusting their data, as it is publicly accessible. In contrast to monitors who verify the consistency of the logs with respect to newly issued certificates, auditors perform a supplementary function of checking for the proper inclusion of certificates in the logs and are thus able to detect any potential gaps in the data. While there are numerous monitors querying for consistency proofs, auditing is not commonly conducted on a large scale in the same manner as monitoring. It is therefore unclear whether the promise made by CT logs to include certificates is kept and can be trusted unconditionally.

In order to ascertain whether there is evidence to suggest that this promise should not be trusted, our research has verified the claims made by CT logs on a large scale. A comprehensive investigation was conducted into millions of promises for inclusion, with the objective of verifying their validity.

The majority of these promises were found to have been fulfilled. In the cases where verification could not be conducted, external factors were identified as the primary reason for this inability.

The findings of this investigation indicate that there is no evidence to suggest that the promises made by CT logs cannot be trusted. Nevertheless, this does not constitute definitive proof that the aforementioned guarantees can be relied upon to the exclusion of all doubt. Further research is required to gain a more comprehensive understanding of this topic. This research should take into consideration the findings presented throughout this study. It should address the obstacles presented by servers providing the issuer's certificates, expand the dataset of certificates, and/or advance

the development of code capable of efficient SCT auditing on a large scale. For the latter, our provided code can serve as a solid foundation.

1 Introduction

1.1 Motivation

In the contemporary era, the ability to communicate via the Internet has become a necessity for all individuals and entities. The use of the internet is not limited to the exchange of brief messages with friends and family; it is also employed in a variety of other contexts, including online banking, e-commerce, healthcare, and the workplace. The transfer of personal and sensitive data is a prerequisite for the provision of these services, which are conducted via the Internet. Such data may include credit card details, home addresses, medical histories, and client information. In order to guarantee the privacy and protection of this sensitive data, it is necessary to implement specific measures. One potential solution is the implementation of network security protocols. Transport Layer Security (TLS) has become the de facto standard for ensuring secure and authenticated communication over the Internet. To fulfill this purpose, TLS relies on digital certificates to establish a secure connection between two parties [Res18]. Certificates are issued by Certificate Authorities (CAs), which are tasked with ensuring that only verified individuals and institutions are granted a corresponding certificate for their domain. In order to ensure the continued security and privacy of digital communications, it is essential to guarantee the integrity and trustworthiness of the Certificate Authorities (CAs) responsible for issuing digital certificates [AL21].

The DigiNotar breach in 2011 had a significant impact on the trust placed in CAs. In this incident, attackers managed to impersonate websites and obtain valid certificates issued by DigiNotar. This resulted in significant security and privacy breaches, which in turn led to a loss of trust in the CAs. This incident underscored the necessity for greater transparency and accountability in regard to CAs [Mil11].

In response to these critical needs, Certificate Transparency (CT) logs have emerged as a potential solution. They provide public and cryptographically verifiable records of digital certificates issued by CAs. This serves the purpose of enabling the operator of a domain to search for and examine all certificates that have been issued for their domain, with the objective of verifying that no fraudulent certificates have been issued. Certificate Transparency (CT) logs are composed of append-only and cryptographically secure Merkle trees, which contain a comprehensive record of all digital certificates utilized within the public internet, at least in theory. Browsers that utilize TLS and require the corresponding digital certificate can then examine whether the certificate is present in a CT log [LLK13].

It is, however, not feasible to instantaneously update the Merkle trees when a new certificate is received, due to the considerable size of the trees and the necessity to append up to several million certificates per day. This implies that a certificate cannot be immediately verified by a browser; however, customers of CAs require certificates to be operational without delay. To meet this expectation, the CT logs promptly return a Signed Certificate Timestamp (SCT), which serves as a *promise* from the CT log provider to the certificate owner that the certificate will be included in their log within a specified timeframe. Browsers may then utilize the signature of the SCT to validate certificates instead of directly examining the logs via API, as this would be excessively time-consuming. While the SCTs ensure the prompt and efficient validation of certificates, the system's overall integrity depends on the accurate and timely inclusion of certificates in logs [Lau14].

It is imperative that a certificate with a valid SCT but without inclusion into the CT log should not occur, as it could give rise to significant trust issues regarding the CT log provider. In the event of such an occurrence, unauthorized or malicious certificates would remain undetected by the domain owner, thereby increasing the risk of exploitation. Such logging errors would result in reputational damage for the organization responsible for managing the "malfunctioning" CT log. Consequently, it would become evident that there is a need for a new supervisory authority, which would independently verify the work of the CT logs.

1.2 Goals and Contribution

The objective of this thesis is to ascertain the reliability of SCTs as an indicator of whether certificates are being logged. The objective is to ascertain whether there are any certificates that have a valid SCT but are not present in the corresponding CT log.
This study has two principal objectives. Firstly, a comprehensive dataset comprising a vast number of certificates will be constructed through the process of scanning the public internet. Secondly, the reliability of the inclusion process with regard to certificates that have obtained a promise of inclusion will be evaluated. The aim is to ascertain whether these certificates are, in fact, included in the CT logs.
In addition to the aforementioned objectives, the project will derive efficient and automated techniques for verifying the inclusion of specific certificates in the CT logs on an extensive scale. In addition to evaluating techniques for crawling the public

internet to obtain suitable certificates, our objective is to provide statistical data on the reasons why certificates are unable to be verified. Furthermore, we intend to identify the challenges encountered when auditing CT logs and to propose potential solutions for improvement.

1.3 Structure

The initial section of the paper sets forth the fundamental technical basics that are necessary for a more profound comprehension of the ecosystem that lies behind the CT logs. Subsequently, the methodology employed for the collection of certificates on a large scale is outlined, along with the procedures utilized for their verification. Thereafter, the efficacy of these approaches is evaluated through the examination of corresponding statistics. In light of these findings, the results are displayed. Finally, a conclusion is drawn regarding the trustworthiness of CT logs, notable issues are highlighted, which emerged during the course of the research and the future direction of research in this field is discussed.

1.4 Information on language support

We used Natural Language Processing (NLP)-tools as well as Large Language Models (LLMs) to create this thesis. All tools were used exclusively to improve the linguistic clarity and readability of the English language, without making any changes to the content. This support was used to make the thesis accessible to a wider readership.

2 Background

The following chapter presents the fundamental (technical) details that are essential for comprehending the subsequent approaches. It offers a succinct overview of TLS and CAs, as well as a more comprehensive examination of CT logs, which are not widely known. It is imperative to grasp the operational dynamics of the CT log ecosystem, and for that, one must possess a certain degree of background knowledge.

2.1 Transport Layer Security (TLS)

TLS is a cryptographic protocol designed to establish a secure communication channel over a network, primarily the Internet. It was developed from Secure Sockets Layer (SSL) and is primarily recognized for its role in securing data transmitted via the Hypertext Transfer Protocol (HTTP), which led to the establishment of the Hypertext Transfer Protocol Secure (HTTPS). TLS ensures the confidentiality, integrity, and authentication of transmitted data [OHR22]. To achieve these objectives, it employs the use of encryption, cryptographic hashes, and digital certificates. The utilization of encryption ensures the confidentiality of the client's and server's data, thereby maintaining the privacy of the communication. Cryptographic hashes and Message Authentication Codes (MACs) are employed to guarantee that the data has not been tampered with [NN19]. In addition, digital certificates are required to authenticate the identity of the server (and, if desired, the client) in order to prevent clients from communicating with an unintended service or malicious third party. The aforementioned factors collectively enable the convenient and secure utilization of online banking services, as well as the seamless installation of software without the concern of malware [Res18].

TLS relies on symmetric cryptography for the encryption of data transmitted between the client and the server. Symmetric cryptography is a relatively rapid process; however, it is not possible to derive a mutual key for the encryption over an unsecure channel. To avoid the possibility of an unintended third party gaining access to the key, a derivation process is employed utilizing asymmetric cryptography. The advantage of asymmetric cryptography is not only that two parties can communicate in an encrypted manner over an unsecure channel; it also allows the construction of a Public Key Infrastructure (PKI), thereby enabling the communicating parties to authenticate their partner, respectively. The primary disadvantage of asymmetric cryptography is its relatively slow processing speed, due to the extensive calculations that are required. Consequently, in order to leverage the benefits of both encryption types, TLS combines asymmetric cryptography for authentication and key derivation with symmetric cryptography for data encryption, thereby optimizing the use of both approaches [SBI13].

To be more precise, TLS employs the use of digital certificates, typically X.509 certificates, with the objective of establishing trust between communicating parties. It has become a standard practice for servers to possess digital certificates in order to establish a connection with TLS. Certificate Authorities (CAs) issue these digital certificates, which contain the public key of the server in question, along with other identifying information. During the TLS handshake, the server presents its certificate to the client, which the client can then verify using the CA's root certificate, which is distributed external to the main communication channel [Res18].

2.2 Certificate Authority (CA)

A CA is an institution that is tasked with the issuance and management of digital certificates. The aforementioned digital certificates can be utilized in cryptographic systems, such as TLS, to authenticate two communicating parties to each other and establish trust. A CA serves as a third party, verifying the identity of organizations or individuals requesting a certificate in order to establish a link between their public key and their identity. The principal function of a CA is to authenticate the identity of an entity (such as a website or a server) prior to the issuance of a digital certificate. In the majority of cases, the entity in question is required to provide proof of ownership or control over the relevant domain in order to confirm their legitimacy. Once the identity of an entity has been verified, a digital certificate is issued by the CA. The certificate comprises the public key, along with other pertinent identifying information regarding the entity in question, such as the domain name. Additionally, it includes a cryptographic signature provided by the CA. Subsequently, the entity is able to demonstrate its authenticity in TLS with the aforementioned certificate [BH17][AL21]. It is fundamental to the functioning of a CA in a cryptographic system like TLS that it can be trusted by all communication partners beforehand. This trust is established "out-of-band," which requires that browsers and operating systems have a technique to verify the CA's authority and integrity without the use of cryptography. This is achieved by distributing root certificates. To maintain this public trust, CAs must adhere to strict guidelines and refrain from any misbehavior [Chi21].

2.2.1 Root Certificate Authorities (Root CAs

A root CA represents the most fundamental level of trust within the PKI framework. These organizations distribute root certificates by installing them directly in web browsers or operating systems. The certificate contains the public key of the CA and is signed by the CA itself. Such a certificate is therefore referred to as a "self-signed" certificate. This provides the fundamental level of trust that is necessary to verify the trustworthiness of all certificates that are signed by this CA. There are numerous Root CAs that issue self-signed certificates; however, only a select few are sufficiently qualified and trusted to issue certificates that can authenticate servers on the public Internet. When a browser encounters a certificate issued to a domain name or server, it verifies that the issuing certificate is one of the aforementioned CAs [Tec10]. Nevertheless, as the trustworthiness of root CAs cannot be proven by cryptographic measures, it is essential to exercise caution and manage the trust placed in them accordingly. They wield considerable influence within the PKI system. If a root CA engages in fraudulent activities, such as issuing illegitimate certificates, it can result in a breach of trust. Consequently, it is imperative that CAs adhere to the highest standards of conduct and procedure in order to guarantee the integrity and legitimacy of their authority and the certificates they issue. In the event of a CA's misbehavior, there is a significant risk of critical security and privacy breaches, including man-in-the-middle (MITM) attacks and phishing schemes [CA-12].

2.2.2 Intermediate Certificate Authorities

In order to enhance security and limit the exposure of root CAs, the majority of CAs operate with one or more *intermediate CAs*. Intermediate CAs act as mediators between root CAs and end-entity certificates (certificates issued to organizations and individuals). The root CA issues a certificate to the intermediate CA, which then issues certificates to organizations and individuals. This structure constitutes a "chain of trust," whereby the intermediate CA is deemed trustworthy if it can be traced back to the root CA. The use of intermediate CAs offers several advantages. For instance, it constrains the potential impact of a compromised CA. In the event of an intermediate CA being compromised, the root CA can revoke the intermediate's CA certificate without directly affecting the root CA or other intermediate CAs. Additionally, intermediate CAs enable the root CA to delegate certificate issuance, allowing for a more distributed approach to certificate management [int].

2.2.3 Certificate Revocation

In certain exceptional cases, it may be necessary to revoke a certificate before its expiration date. This is typically the case when a certificate has been compromised, the domain owner has lost control over the domain, or the certificate was issued fraudulently.

Certificate Revocation Lists (CRLs)

A CRL is a list consisting of revoked certificates, which are therefore invalid. It is feasible for browsers and operating systems to routinely download and examine CRLs to ascertain whether a particular certificate has been revoked. This approach results in a certain degree of latency, as there is a delay between the revocation of a certificate, the updating of the list, and the subsequent checking of the certificate [BSP+08].

Online Certificate Status Protocol (OCSP)

OCSP is a service that enables browsers and operating systems to ascertain the validity of a certificate in real time. Upon establishing a connection with a server, a client may request the status of the server's certificate from the CA. In the event that the certificate has indeed been revoked, the service will indicate that a revocation has occurred. This approach offers a more expedient and efficient solution than CRLs; however, it does impose the necessity for the CA's infrastructure to remain continuously responsive to such requests [SMA+13].

2.3 Certificate Transparency Logs (CT Logs)

In 2011, DigiNotar, a Dutch certification authority (CA), was the subject of a cyber-attack. A significant security breach resulted in the issuance of over 500 illegitimate digital certificates [Mil11]. It is alleged that the attack was carried out by an Iranian hacker who successfully exploited vulnerabilities in DigiNotar's infrastructure. The aforementioned illegitimate digital certificates were utilized in MITM attacks, largely targeting Iranian internet users. The interception of communications included email correspondence and private browsing sessions. As a consequence of the incident, nearly 300,000 users of Google's service were potentially exposed to the risk of surveillance or impersonation by the attacker [Art11].

The incident occurred due to inadequate security measures employed by DigiNotar, which resulted in the unauthorized issuance of digital certificates going undetected. The fraudulent certificates remained undetected for approximately one and a half months before being identified. The breach had far-reaching consequences, affecting not only individual users, but also the reputation that DigiNotar had built as a CA. Ultimately, this led to the company's bankruptcy. Of greater consequence, the incident brought to light deficiencies in the extant PKI model. This demonstrated that a CA that is misbehaving or has been compromised can issue certificates without being noticed. Had there been a mechanism for monitoring all issued certificates for the public, the fraudulent certificates could have been identified significantly sooner, thereby limiting the extent of the damage [Wol16].

It was thus deemed necessary to construct a framework that would enhance the ability to identify falsely issued certificates, as security breaches are an inevitable consequence of any system, regardless of the strength of its defenses.
In response to the identified shortcomings of the PKI, CT Logs have emerged as a potential solution following the DigiNotar incident. They maintain a publicly accessible registry of all certificates issued by CAs, thereby enabling real-time monitoring of CA activities. Consequently, they make it possible to detect falsely issued or fraudulent certificates with immediate effect. In consequence, it is not necessary to place inherent trust in a CA; rather, one may examine their work. [LKL13]

CT logs achieve their objective by requiring all CAs to submit all of their issued certificates to public, append-only logs. Such logs may be monitored and audited, thereby enabling browsers, operating systems, organizations, and users to verify the legitimacy of a certificate and ascertain whether it has been tampered with. The transparency and accountability of every CA's operations is enhanced through the publishing of their work in CT logs. It provides the internet community with the ability to examine the actions of CAs in greater detail.

One of the most significant advantages of CT logs is their capacity to detect issues with certificates at the point of issuance. In the event that a fraudulent certificate is issued by a CA for a domain of which the domain owner is unaware, it can be promptly identified and conveyed to the relevant authorities. Furthermore, the absence of a log entry for a given certificate would be cause for suspicion. Consequently, it is possible to impose penalties on CAs that engage in misbehavior by reducing their reputation within the ecosystem. Browsers are capable of rejecting certificates that have not been logged, thereby ensuring that only those visible to the public are trusted [Sol19][Int13][LLK13].
For CT logs to fulfill their intended purpose, it is necessary to implement a data structure that is append-only. This structure must be able to demonstrate that a certificate has been logged and that no alterations have been made to the rest of the log.

2.3.1 Merkle Trees

Merkle Trees were identified as a suitable data structure for meeting the requirements for CT logs. Merkle Trees are capable of guaranteeing immutability and efficient verification. Merkle Trees provide a method for maintaining append-only logs while simultaneously ensuring that any alterations, deletions, or additions to the log entries are detectable by the monitoring party. A Merkle Tree is a binary tree in which each leaf node consists of the cryptographic hash of a data block. In the context of CT logs, the aforementioned data blocks are represented by digital certificates. The hash of each non-leaf node results from the combination of the hashes of its two child nodes, thus establishing a hierarchical structure in which every interior node is derived from the hashes of its children. This process continues upwards until the tree head is reached. The tree head is a cryptographic representation of the entirety of the data contained within the tree.

The properties of cryptographic hashes ensure that any alteration to a single node will affect all nodes above it, ultimately resulting in a change to the tree head. This allows for the straightforward detection of any tampering. Furthermore, this also applies to the order of the leaf nodes. In the context of CT logs, this implies that if any certificate is altered, removed, or added to the log without compliance with the prescribed logging procedure, the tree head will undergo a modification, thereby indicating a discrepancy. Merkle Trees allow for the efficient verification of log integrity at any given point in time, utilizing the tree head [Wan20].

Inclusion proof

In order to ascertain whether a specific certificate is present within the tree, it is sufficient to obtain the hash value of the certificate in question and the hashes of the sibling nodes traversed along the path from the leaf node to the tree root. The hash of the certificate can be calculated independently, and the log provides all the necessary hashes of the sibling nodes. This enables the client to perform the calculation of the tree head by themselves. If the computed tree head matches the given tree head, the client can be certain that the certificate is indeed part of the tree and that no tampering has occurred. As the operator of the Merkle Tree is only required to provide a minimal amount of data (the hashes along the path to the tree head) in order to prove their integrity, the process is highly efficient [Wan20].

Consistency proof

As previously stated, Merkle Trees also enable the generation of a consistency proof. This process serves to verify that a newly created tree, which may contain new entries, still includes all the previously logged data. In order to verify the consistency of a current log (new tree head) in relation to an older tree head, the client must have access to the latter in advance. From the log, the client retrieves the new tree head, along with the hashes of the recently added entries and the nodes required to calculate the intermediate hashes from the old to the new tree head. With this information, the client can then compute the new tree head. If it matches the actual new tree head, the client can verify the log's consistency, particularly ensuring that no entries have been altered or removed and that new entries have been added correctly [Wan20].

2.3.2 Procedure

Upon receiving a request from a domain owner for a certificate, a CA transmits the issued certificate to a CT log operator, who is typically selected by the CA itself. While CAs typically undertake this responsibility, any individual may request the inclusion of their certificate, provided that it was issued by a trusted CA. To assist with this process, CT log operators maintain lists of trusted root CAs. Upon submission of the certificate, the CT log responds with an inclusion promise, agreeing to record the certificate within a specified timeframe, known as the Maximum Merge Delay (MMD). It is imperative that this promise be embedded within the certificate, thereby enabling verifying parties to verify that the certificate has been logged in an appropriate manner. Subsequently, the certificate is returned to the domain owner.

To validate the certificate's inclusion, a party may rely on the attached inclusion promise or conduct additional verification. A cryptographic proof of inclusion can be requested directly from the CT log using the details provided in the aforementioned promise. Provided that this request is made subsequent to the Maximum Merge Delay (MMD), the CT log is obliged to confirm the inclusion. Failure to comply with these procedures may indicate that the CT log is misbehaving, which could result in consequences for the operator.[Cer24]

2.3.3 Signed Certificate Timestamps (SCT)

Certificate Transparency logs are confronted with a considerable challenge in meeting the expectations of certificate authorities (CAs) and domain owners, namely to provide immediate inclusion proof for newly issued certificates. Given the global scale of the Internet and the significant daily volume of certificate issuance, CT logs handle millions of submissions on a daily basis. Due to the technical limitations associated with instantaneous appending of every certificate to the Merkle Tree structure, SCTs provide an effective solution. SCTs serve as immediate assurances that a certificate has been accepted by a CT log and will soon be publicly available, thereby mitigating delays caused by inclusion constraints.

An SCT is a compact, cryptographically signed data structure that is produced by a CT log and issued to the CA, thereby guaranteeing the future inclusion of a certificate. Each SCT possesses a set of distinctive elements that collectively establish this assurance.

The first of these is a log ID, which serves as a unique identifier linking the SCT to the corresponding CT log that issued it. Subsequently, a timestamp indicates the precise moment at which the SCT was generated, thereby providing a reliable reference point for the initial request by the CA for logging. The inclusion of this timestamp is of crucial importance, as it is a prerequisite for the Maximum Merge Delay (MMD). Lastly, the digital signature created by the CT log serves to authenticate and verify the integrity of the SCT. The digital signature, created using the log's private key, can be validated by any party in possession of the corresponding public key, thereby ensuring the SCT's credibility.

These combined components serve as a binding commitment from the CT log to the CA, promising that the certificate will be included and accessible to the public within the MMD. The SCT is embedded into the certificate and transmitted to the client, thus enabling any recipient to view and verify it. This arrangement allows CAs to provide immediate proof of the certificate's pending log inclusion, even if the certificate has not yet been added to the Merkle Tree.
When the certificate is subsequently added to the Merkle Tree, the SCT's information is essential for confirming that the log has honored its commitment to timely inclusion [Lau14].

2.3.4 Precertificates

Another method employed by CT logs to prevent CAs from engaging in improper conduct is the utilization of precertificates. This mechanism guarantees that CAs are unable to modify a certificate in any manner after the CT log has promised to including it. In order to achieve this, the CA initially generates a certificate that contains a special poison extension. A certificate that possesses this extension is thus deemed invalid and should not be accepted as a trusted certificate. This version of the certificate, designated a "precertificate," is not intended for utilization in a TLS handshake and is typically not disseminated by the domain proprietor.

Upon issuance of a certificate to a domain owner, a CA signs the certificate, including the aforementioned extensions. Consequently, if specific extensions undergo alteration, the signature is also modified. In contrast, CT logs generate their signature by taking into account the entirety of the certificate, excluding the poison extension. The signature for the leaf certificate and precertificate, therefore, differ; however, the signature generated by the CT logs remains consistent.

The CT log then returns an SCT, calculated over the precertificate, to the CA. In order to produce a fully functional leaf certificate, the CA must first remove the poison extension, then append the provided SCT, and finally update the certificate's signature. Consequently, during a TLS handshake, the user will receive the leaf certificate instead of the precertificate.

To verify the signature in the SCT, a specific adjustment is required. In order to generally verify a signature, it is first necessary to reconstruct the content that was used to produce it. Consequently, users must remove any SCTs embedded within the certificate to accurately recreate the original content, the precertificate.

This technique ensures that, apart from the SCTs included within the certificate, no other modifications can be made by the CA without alerting users. In the event of any alteration occurring after the CT log has issued the SCT, the signature will not be verifiable, indicating that the CA is misbehaving [Wan20].

2.3.5 API

CT logs provide API interfaces to facilitate communication and interaction. These interfaces offer the ability to perform a variety of functions, including retrieving the size of the Merkle Tree, obtaining the public key associated with the CT log, requesting the inclusion of certificates, acquiring a list of all trusted root CAs, retrieving a specific log entry based on its position, and, most notably, obtaining a cryptographic proof confirming the existence of an entry when provided with the entry's hash. Each log entry consists of a number of distinct components. The entry comprises predefined constants, the timestamp at which the submitted certificate was accepted by the CT log, and the certificate itself. In the event that the entry in question is a precertificate and not a leaf certificate, the entry will also include the hash of the certificate issuer's public key.
Furthermore, the timestamp within the entry must match the timestamp of the corresponding SCT [LLK13].

To ensure the continued reliability of CT logs, rate limiting is employed when the volume of requests from individual users exceeds a specified threshold. The rate limiting and maximum response size are set according to the specifications of the log provider in question [rat22][rat23].

2.3.6 Monitoring

A notable benefit of CT logs is their verifiability for users, which is made possible through the use of Merkle Trees and the objective of ensuring 100% uptime by log providers.
Monitors observe logs to guarantee their proper functioning and track certificates of interest. In order to achieve this, it is necessary for a monitor to review each new entry in every log that it oversees. Some monitors may even choose to retain copies of entire logs.

Individuals interested in developing a monitor have the capability to do so, enabling continuous oversight of CT logs to confirm compliance on a large scale. A significant number of monitors are designed to perform consistent scans of one or multiple logs, whereby they request periodic consistency proofs for the Merkle Trees in use. Additionally, some monitors verify that certificates are included in the log in a timely manner with regard to the MMD [LLK13]. Furthermore, monitors may exchange recent updates and proofs provided by the CT log to ensure the log's information is globally consistent [Lau14].
Monitors can be deployed to achieve a variety of objectives. They may track log activity to verify that, with each update, the prior version of the Merkle Tree remains unaltered. Additionally, they can confirm that new entries correctly generate the elements necessary for the consistency proof. Some monitors also offer services tailored to domain owners, such as notifying them when a certificate is issued for their domain. Therefore, if a certificate is issued for a domain without the owner's consent, the monitor can notify the corresponding CA to revoke that certificate [mon].

2.3.7 Auditing

While monitors are responsible for ensuring that logs are append-only and consistent across the network, auditors are tasked with examining the actual content of the logs. Their role is to investigate instances of misissued certificates or the omission of expected certificates. Monitors retrieve only the certificates that have been appended to the log, and thus are unable to verify the absence of expected certificates. In contrast, auditors are tasked with not only verifying the validity of observed SCTs but also ensuring that the promises these SCTs represent have been fulfilled. This verification is accomplished by requesting inclusion proofs from the corresponding CT log. In the event that domain owners do not proactively examine for any absent certificates, undetected gaps may remain unidentified by a monitor.

The auditing of an inclusion proof for each certificate encountered by a user could potentially compromise privacy, as it would likely reveal their browsing history. Consequently, in the current CT implementation, the majority of clients avoid this crucial examination to safeguard user privacy [MDO^{+}22].

2.3.8 Handling by browsers

In order to fully utilize the enhanced security afforded by CT logs, certain browsers have instituted policies that mandate the embedding of valid SCTs within certificates. Each browser is at liberty to define its own criteria for the acceptance of a certificate. For example, Mozilla Firefox currently imposes no requirements, whereas other browsers, including Chrome/Chromium [chr], Safari [app], and Brave [bra], mandate

the presence of at least two SCTs. However, the majority of browsers only verify the SCT's signature. If the signature is deemed valid, the SCT is then interpreted as a promise of inclusion, thereby classifying the certificate as trustworthy.
Conversely, if a certificate fails to meet the established requirements, the connection is considered untrustworthy and is therefore not established. In such instances, an error message is displayed, enabling the user to determine whether to proceed with the connection or terminate the session.

A primary advantage of solely verifying the SCT signature, as opposed to requesting a comprehensive proof of inclusion from the associated CT log, is that it minimizes the time required to establish a valid connection. This is because the process of validating a signature is considerably faster than querying a server for each SCT in a certificate. Furthermore, the request for inclusion proofs may potentially compromise privacy. In addition to basic validity checks, Chrome/Chromium has initiated an audit of a limited number of certificates collected from its users, with the objective of balancing security with user privacy. The determination of whether inclusion proofs are conducted is based on the individual user's "Safe Browsing" settings [ST20][chr22]. It remains to be seen whether other browsers will adopt similar mechanisms to audit a portion of certificates for inclusion proofs from CT logs.

3 Approach

The objective of this thesis is to evaluate the reliability of STCs in guaranteeing the logging of certificates in CT logs. In order to answer this question, we present an approach in the following chapter for the collection and analysis of certificates from the open internet on a large scale. We analyze the certificates with a particular focus on the validity of SCTs and the inclusion of the corresponding CT logs.

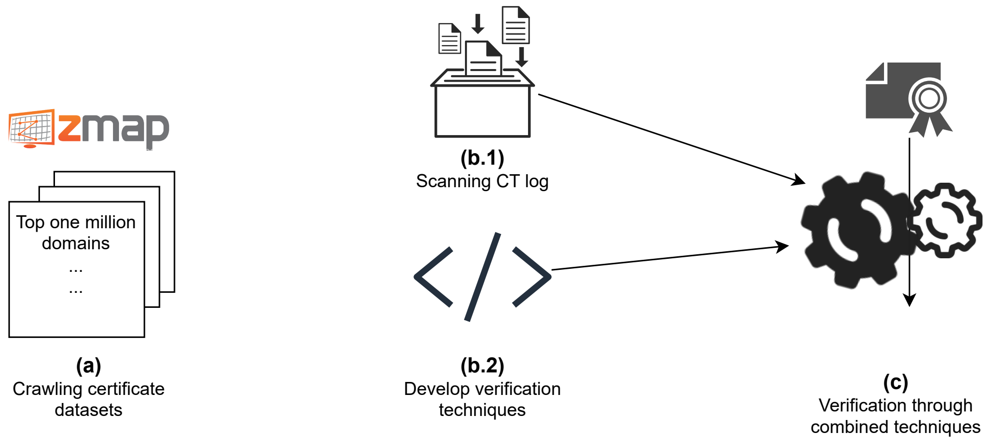
zmap
Top one million domains
...
...
(a)
Crawling certificate datasets
(b.1)
Scanning CT log
(b.2)
Develop verification techniques
(c)
Verification through combined techniques

Figure 3.1: The methodology employed to ascertain the compliance of SCTs is as follows: (a) The gathering of certificates, (b.1) the downloading of an entire CT log, (b.2) the development of a verification process based on the available data, and (c) the verification of the gathered certificates using a combination of the aforementioned processes and other relevant procedures.

The methodology presented in this thesis is outlined in three main stages, as illustrated in Figure 3.1. The initial step is to gather X.509 certificates on a large scale from across the internet, ensuring a comprehensive and representative dataset. This collection is of great importance to us, as it may lead to significant insights and, potentially, the discovery of discrepancies between SCTs and CT logs. Secondly, following the acquisition of the aforementioned certificates, a verification procedure is developed for determining the inclusion of each certificate in its corresponding CT log based on that certificate's SCT. This involves querying the CT logs to ascertain the certificate's inclusion and confirm that it has been correctly logged in accordance with the SCT. The discovery of a single certificate with a valid SCT but absent from its corresponding log would provide evidence of misbehavior on the part of the CT log. Thirdly, the effectiveness of the developed tools was evaluated in conjunction with the SCTs identified in the collected certificates.

While the absence of evidence that a CT log misbehaves does not constitute proof that SCTs are always reliable, a comprehensive investigation with no significant findings of misbehavior would provide substantial reassurance that SCTs are highly trustworthy. With this procedure, we should be able to identify any anomalies that have the potential to negatively impact trust in the CT ecosystem, ultimately contributing to the evaluation of the overall integrity of SCTs.

3.1 Gathering of certificates

In order to construct a comprehensive and representative dataset consisting of a significant number of certificates, a variety of methods were employed to scan, retrieve, and collect certificates from the Internet.

3.1.1 IPv4

To ensure comprehensive coverage of internet infrastructure, one of the methods employed for the acquisition of certificates involved the comprehensive scanning of the entire IPv4 address space. To this end, we utilize ZMap, a highly efficient network scanning tool capable of scanning the entire IPv4 address space, with the objective of identifying hosts that respond to the default port for HTTPS 443. This methodology enables the construction of a list of hosts from which a TLS certificate can be obtained. ZMap is an effective tool for rapidly identifying active servers, making it well-suited for large-scale internet scans.

Subsequently, ZGrab, an additional tool from the ZMap project, is employed to retrieve the X.509 certificates from the identified servers. ZGrab is a specifically designed application-layer scanner, which enables the downloading of TLS certificates. This is achieved by establishing a connection with the server and completing the TLS handshake. This approach allows us to collect a diverse set of certificates from a broad range of hosts, representing different organizations, geographic regions, and security configurations. The combination of ZMap and ZGrab is an effective method for gathering a comprehensive data set of certificates from across the entire IPv4 space.

3.1.2 Tranco's one million

An alternative approach is to concentrate on the most popular websites on a global scale. In this regard, the Tranco list is deemed an appropriate source of information. As a "researcher-oriented ranking," it aggregates and ranks websites based on popularity across multiple sources, resulting in a ranking of the top one million domains. Tranco is particularly well-suited to academic research as it is relatively stable and resistant to manipulation in comparison to other rankings [LPVGT$^+$19].

To establish a connection with each domain and download the corresponding X.509 certificate, we utilize OpenSSL, a widely utilized cryptographic toolset for Linux. This method allows us to ensure that we capture certificates from the most heavily visited websites. This approach offers insight into the security practices of high-traffic websites, rendering the certificates collected through this method valuable for evaluating the SCT mechanism.

By focusing on the most visited domains, this approach enhances the comprehensive IPv4 scanning by additionally capturing certificates from high-value targets.

3.2 Verifying inclusion of certificates

Once a substantial number of certificates have been obtained from a variety of online sources, the subsequent essential step is to ascertain whether the certificates have been correctly entered into the CT logs, as indicated by their SCTs. This process involves a detailed examination of each certificate and its associated SCTs. The objective is to ascertain the CT log that issued the SCT and to verify the certificate's inclusion in the designated CT logs. This approach ensures the legitimacy and validity of the SCTs.

3.2.1 Google's Go library

In order to verify the authenticity of digital certificates and their corresponding SCTs, we employ the Google Go library for Certificate Transparency. The library, developed by Google, has been designed with the specific purpose of working with CT logs. It provides a comprehensive suite of tools for their management. The library offers a variety of functions, including data encoding, interaction with CT log APIs, extensive log scanning, and the capability to host one's own CT log. The versatility of this tool renders it an immensely useful apparatus for the retrieval of CT logs and the authentication of SCTs.

CT log on local machine

The first method entails downloading the entirety of the CT logs to a local machine for subsequent examination. This allows for rapid and autonomous confirmation of the inclusion of certificates. To achieve this, the "sctscanner" tool from the Go library is employed. The tool is capable of retrieving each entry from a specified CT log, thereby allowing the user to process the retrieved certificate in a manner that suits their requirements.

To enhance the efficiency of this process, a file is created for each certificate obtained, with the certificate's hashed public key serving as the file name. The content of each file is minimal in order to utilize the least amount of storage space possible, containing only the entry number of the corresponding certificate in the CT log. This design allows for rapid storage and retrieval, enabling straightforward verification of inclusion by matching the certificate fingerprint with the locally stored files.

The verification process is as follows: when a certificate is provided, its SCT identifies the CT log in which it should be included. The certificate's hashed public key is calculated and checked against the locally stored files corresponding to that CT log. If a file with the same name as the fingerprint of the certificate exists, it can be concluded that the certificate is indeed included in the log.

sctchecker

In addition to downloading entire logs, the Google Go library provides the "sctchecker" tool, which is designed to facilitate the process of verifying SCTs from a given certificate. The tool permits the input of a PEM file, and returns the status of any SCTs identified in the certificate under examination with regard to verification.

The "sctchecker" tool employs a variety of components within the Go library to facilitate communication with CT logs and perform SCT verification against the intended log.

3.2.2 Own Python Code

As an alternative to Google's "sctschecker," we developed our own Python script to verify SCTs directly through the CT log's API. This bespoke solution enables us to manage the entire verification process independently, thereby affording us the flexibility and transparency into the underlying steps that we require. As a result, we are able to exercise greater control over the manner in which the verification is conducted. This is especially beneficial in instances where the process encounters an obstacle and further troubleshooting or investigation is necessary.

A notable benefit of custom-developed code is its capacity to provide comprehensive diagnostic information. By performing each step of the verification process independently, such as querying the log, analyzing the SCT data, and checking the inclusion status, potential issues can be identified at each stage of the process. This allows for the isolation of the source of any issues, whether they are related to the SCT, the certificate itself, or the log. This enables us to identify issues and refine the verification process.

A further advantage of this methodology is the capacity to generate statistical data and reports on the outcomes of the verification process. As we oversee the entirety of this process, we are able to obtain comprehensive data regarding each verification, including success rates and the specific types of errors encountered. This enables us to conduct a more detailed analysis and to obtain more precise metrics regarding the performance and reliability of SCTs and CT logs.

3.2.3 Censys.io

In addition to the aforementioned methodologies, we were granted research access to Censys. The Censys platform offers a comprehensive repository of internet data, including detailed information about TLS certificates. In addition to scanning the IPv4 address space, the tool also scans parts of the IPv6 address space, thereby providing a more comprehensive view of the internet's certificate infrastructure [DAM$^+$15].

By drawing upon the vast repository of data made available by Censys, we are able to examine certificate information that could not be processed by other techniques.

3.3 Advantages and Disadvantages

The following section will present an overview of the advantages and disadvantages associated with the various techniques employed during the verification process. These qualities are considered in regard to their ability to provide the information required for the purposes of this thesis.

3.3.1 CT log on local machine

The second and third approaches are dependent on the API of the CT logs, which correspond to the SCTs that require verification. Consequently, the efficacy of these approaches requires that the API of the CT log maintain optimal uptime and a reasonable response time. This is not a guaranteed condition, particularly in the case of older CT logs. Such logs frequently possess high latency and/or low uptime, or, in extreme cases, are entirely unavailable. In addition, the inclusion proof requires the issuer's certificate, which must be available for the other two techniques to be applicable. In the absence of the requisite data, it is not possible to calculate the appropriate hash for the inclusion proof request. It is imperative that the issuer's certificate be downloaded in advance or in a timely manner. This presents a significant challenge, as there is no publicly accessible repository of issuer certificates that can be downloaded in advance. Additionally, our testing machine has experienced instances of blocked traffic from the servers that provide issuer certificates.

The aforementioned issues are not present when CT logs are stored locally. In this approach, there is no reliance on an API, and requests are not sent via the network; instead, they are examined in a directory on the local machine. This represents a significant time-saving process. Furthermore, the need for issuers' certificates is obviated by the ability to identify logged certificates through alternative means, such as their corresponding fingerprint. It can be reasonably deduced that a local copy of the CT log would result in a notable reduction in the difficulties encountered. However, for this to be a viable solution, it is necessary to download the entirety of the CT log, which is a time-consuming and space-intensive process, particularly given that a single CT log may contain hundreds of millions of certificates. Consequently, it is essential to download the CT log in advance to ensure its availability for use, a process that is once again highly dependent on the API, specifically in terms of its uptime and response time.

3.3.2 sctchecker

This tool was developed by Google and is in accordance with the organization's established standards of quality. This indicates that the code has been subjected to continuous improvement and rigorous testing for a multitude of potential scenarios. It is, therefore, a highly reliable tool in this context. It can be reasonably assumed that the tool's output is reliable and accurate.

The tool was designed to be universal and to accommodate every potential scenario, which makes it a highly robust and versatile tool. However, its complexity and breadth of functionality may not be optimal for our specific use case. The output is not easily accessible and requires post-processing to yield the desired information. This process is time-consuming, as it involves converting the output to a usable format. Compared to the Python code, the sctchecker is more comprehensive, covering additional use- and edge-cases.

3.3.3 Own Python code

In the context of our research, we developed an application that is tailored to our specific requirements. This implies that the application is limited in terms of its functionality, yet it is sufficient for achieving the desired outcome for our research. As our study is conducted by a limited number of researchers, we lack the capacity to exhaustively test our own code and identify exceptional and infrequent edge cases. It is therefore reasonable to assume that the code inhibits the occurrence of some bugs.

By controlling the entire verification process, it is possible to identify precisely when problems or errors occur. This allows for the collection of sophisticated statistics about certificates, SCTs and CT log behavior. Additionally, by developing the code ourselves, we can ensure that it is as efficient as possible for our research purposes.

3.3.4 Censys.io

Censys was initially conceived as a universal tool. For the purpose of gathering certificates and verifying their inclusions. To this end, we were able to procure a researcher's license, which affords us restricted access to the comprehensive database. Unfortunately, the limitations are such that we are unable to perform either of the two functions. Our contingent of 25,000 requests is insufficient for large-scale, automated processing of certificates.

Nonetheless, we are grateful for the opportunity to utilize this service, as it allows us to enhance our code and identify solutions to certificate-related issues on a limited scale.

3.4 Combination of different techniques

As previously discussed, the various techniques possess distinct advantages and disadvantages, which, in part, complement one another. Consequently, the techniques are employed in conjunction with one another, rather than being limited to a single approach. It is evident that both our Python code and the sctchecker are capable of verifying any SCT, provided that the API is accessible. In contrast, the local CT log approach necessitates the complete download of the corresponding CT log in order to verify a single SCT. Furthermore, it should be noted that our own code is more time-efficient than the sctchecker, although it may not be capable of handling all potential scenarios and is vulnerable to instances of software malfunction.

Accordingly, the techniques are employed as follows. Firstly, our own Python code and the local CT log are employed in conjunction with one another. This entails that for each SCT that is identified and signed by a CT log that has been downloaded in advance, the inclusion of that SCT in the local log is verified. In the event that the SCT is signed by a log that has not been locally saved, a request for inclusion verification is initiated.

As the verification process is ongoing, any certificates that cannot be verified are flagged for subsequent attention. Once all certificates have been processed using the Python code, the SCTchecker is employed to attempt to verify the certificates where the initial verification was unsuccessful. Certificates that remain unverified are included in our statistical analysis.

This approach allows us to leverage the strengths of each technique while mitigating the inherent limitations of each. By employing the local log and Python code, we can rapidly verify certificates, although we may encounter unforeseen issues. To address these potential shortcomings, we utilize SCTchecker to perform a secondary examination of certificates identified as problematic in our Python code.

4 Implementation

This chapter provides a detailed examination of the practical application of the approach outlined in chapter 3. The following section provides a more detailed explanation of each step, accompanied by an overview of the manner in which the approach is conveyed in the actual implementation. In this project, the code comprises both Linux command-line instructions and Python scripts, which provide a diverse toolset. The complete code listings are available in the appendix for reference.

4.1 Gathering of certificates

In order to obtain a substantial sample of certificates, two principal methods are employed. First, we utilize ZMap to scan the IPv4 address space for servers that offer TLS connections. Once the servers have been identified, the actual retrieval of the relevant certificates is managed using ZGrab. The resulting certificates provide a diverse dataset of the broad IPv4 space. In addition, the Tranco list was also used, which ranks the top 1 million most popular websites. This approach also considers certificates from the internet's most frequently visited domains. For this collection, OpenSSL is deployed to connect to the websites and download the corresponding certificates.

4.1.1 IPv4

The initial step is to utilize ZMap, a project designed for fast and effective scanning of the IPv4 address space. This allows us to identify which hosts within the IPv4 space are active and, more specifically, which hosts have an open port 443. Port 443 is the standard port for HTTPS connections and typically provides an X.509 certificate if it is active, as it is a prerequisite for establishing secure communication via TLS. The identification of open port 443 is exclusively handled by ZMap. By passing port 443 as an argument, ZMap performs a systematic scan of the entire IPv4 address space, searching for servers that are responding on this port. Consequently, a comprehensive list of IP addresses corresponding to relevant servers is generated. Nevertheless, at this point in the process, the only information available regarding these servers is that their port 443 is not closed. No further details about their services or certificates have been identified.

The second phase of the process involves establishing a connection with each of the identified IP addresses with the objective of downloading the associated digital certificate, should one be available. This is achieved through the utilization of ZGrab. ZGrab is a tool that is capable of initiating a handshake with the server in question, thereby enabling the collection of relevant details regarding both the server and the provided service. By specifying the TLS protocol and port 443 as arguments, ZGrab attempts to perform a TLS handshake with each of the servers on the given list. In the event that the server in question provides a digital certificate, ZGrab will retrieve it and store it locally.

The results of the comprehensive scanning process are documented in a JSON file, with each scanned OP address represented by an individual entry. The aforementioned entries contain a variety of crucial data points, including the success or failure of the certificate retrieval, the unaltered X.509 certificate data, the certificate's fingerprint, and numerous other pertinent details. This structured output facilitates further analysis and processing of the certificates.

Prior to undertaking further analysis of the saved certificates, it is essential to undertake preprocessing. The raw base64-encoded certificates are initially converted into PEM format, which is widely recognized and supports a variety of verification processes. This includes all of the planned processes. Therefore, storing the certificates in this way allows for multiple validation steps to be performed on the same file without requiring repeated conversions.

4.1.2 Tranco's one million

In conducting our research, we employ the Tranco list, dated August 26, which is a regularly updated ranking of the top one million domains globally. Each entry in the list represents a single domain. The Tranco list provides a comprehensive and robust dataset for the acquisition of certificates. The precise methodology underlying the Tranco list ranking process is beyond the scope of this thesis. However, it is noteworthy that the list is specifically designed for research purposes, as it combines data from multiple sources to provide a stable and manipulation-resistant ranking of the most visited websites.

The subsequent phase of the process is to obtain the digital certificates from each domain. In order to achieve this, we utilize the OpenSSL tool, which is a well-established cryptographic tool that is commonly pre-installed on most Linux systems. OpenSSL provides secure communication by offering open-source code for the management of certificates and the performance of TLS/SSL handshakes. In the process described here, OpenSSL was used to establish a connection over TLS with each domain from the list on port 443, which is the default port for HTTPS. The X.509 certificates provided by the server were then downloaded.

Each retrieved certificate is stored in a distinct file. The filename is designated according to the domain name of the corresponding website. The certificate is stored in PEM format, which encodes the certificate in a base64 representation, facilitating subsequent processing. When feasible, OpenSSL acquires the entire certificate chain, which entails not only the leaf certificate (i.e., the server's certificate) but also the intermediate and root certificates from the associated CA. This approach ensures that the downloaded files provide a full context for validating the authenticity of the certificates.

The process of utilizing OpenSSL to download the certificate chains was initiated via a shell command, which combined multiple tools and functions. Initially, the order of domains is randomized to ensure a balanced and unbiased collection process, wherein no particular domains are favored based on their original sequence. Furthermore, multiple tasks are conducted in parallel. The implementation of parallelization has the effect of markedly enhancing efficiency, allowing for the execution of up to 16 downloads in parallel, and thus reducing the time required to obtain certificates from the given one million domains.

In the command, the OpenSSL application is instructed to attempt a TLS handshake with each domain on port 443. By employing the -showcerts flag, it is ensured that all certificates, including intermediates, are displayed. The -servername option is utilized to facilitate the Server Name Indication (SNI), thereby ensuring that certificates are retrieved correctly even from hosts that utilize multiple domain names under the same IP address. A limitation is placed on the time it takes for a connection to timeout, which helps to avoid lengthy delays with unresponsive servers. Finally, the certificate is extracted from the output and saved to a PEM file.

4.2 Verifying inclusion of certificates

To confirm the accuracy of the SCTs on each certificate, a variety of methods are employed. Some approaches are conducted using a well-known GitHub repository developed by Google employees. An alternative approach is to develop custom Python code. The final approach, which employs Censys, entails querying the online database manually when necessary.

4.2.1 Google's Go library

Both approaches to Google's Go library use the same repository. We will not describe the concrete functionality of the code, only how we use it for our pur-

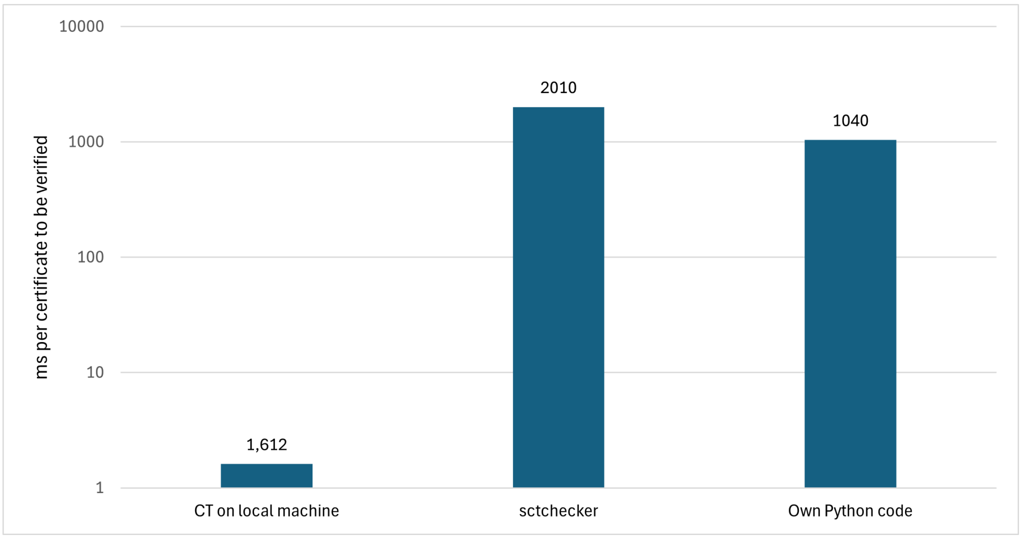

Figure 4.1: The average time required to verify a single certificate is presented in logarithmic scale for each of the techniques. It should be noted that it is unclear whether the Python code in question contains errors that could result in it taking 1.04 seconds to run without completing its intended function.

poses.

CT Log on local machine

This approach is fundamentally distinct from all others. The objective is to download the entire CT log. This results in the preservation of each distinct representation of a certificate on our own machine. This approach offers a more straightforward method for verifying the presence of a certificate within a specific CT log. However, this approach also necessitates the expenditure of a considerable amount of resources.

The download is conducted using the scanner component of the GitHub repository. The tool allows for the retrieval of each entry from a CT log, which can then be processed according to the user's specifications. However, in order to leverage the tool's versatility, it is necessary to create a Go file that will handle the fetched entries.

The program for downloading the CT log imports a number of standard Go packages for the handling of HTTP requests, cryptography, file operations, and logging, as well as packages from Google's Certificate Transparency Go library for the interaction with CT logs and the handling of certificates. The scanner is configured with a number of concurrent workers, which serve to increase the speed of the scanning process. Additionally, the option exists to only scan for precertificates. Upon detecting a certificate within the log, the program extracts the public key of the precertificate and calculates a SHA-256 hash over the key. Subsequently, the certificate's entry index is stored in a file whose name is derived from the public key's fingerprint.

To ascertain whether a certificate is included in a downloaded CT log, Python code is employed. The certificate is obtained in PEM encoding. The cryptography and PEM library is employed to load the certificate and extract the public key. Subsequently, the hashlib library is utilized to calculate the SHA-256 hash over the extracted public key.

Finally, the file with the calculated hash is checked for existence. If a file with matching name is found, it can be concluded that the certificate is included.

In practice, the average time required for the verification of the inclusion of one certificate is 1.612 milliseconds 4.1.

sctchecker

Additionally, the repository provides access to the sctchecker tool. The tool is capable of performing basic SCT verification. Upon presentation of a certificate chain, the tool attempts to ascertain the inclusion of the certificate. The output provides the number of SCTs within the certificate that were validated, as well as the reason for any unsuccessful verification attempts.

To facilitate the processing of a large number of certificates, a bash script is employed. The script reads each line from a given text file. Each line contains a file name, which represents the name of a certificate where the SCT is to be checked. If the file exists, a Go command is executed. This command utilizes the sctchecker to verify the certificate contained in the corresponding file. The progress is outputted on stderr and provides information about the verification process of every single SCT it checks. The output must then be processed manually.

In practice, the sctchecker can verify an average of one certificate in 2.01 seconds 4.1.

4.2.2 Own Python Code

Another proposed method for verifying the inclusion of certificates is the development of a custom code that can verify individual certificates through the API provided by the CT logs.

We have elected to utilize the Ray library as a parallelizer with the objective of accelerating the processing of the extensive list of certificates. Additionally, our code employs a multitude of libraries, including cryptography, pyasn1, pyasn1_modules, requests, base64, time, json, os, pem, and hashlib, to assist in the processing of the task. The initial step is to create a class that is specifically designed to handle the statistics that are gathered during the process of verifying the inclusions. This is necessary in order to handle the statistics while parallelizing the code. The program collects data regarding the number of times and reasons for which the inclusion of certificates could not be verified, as well as the average processing time for a single certificate.

Although the access to the APIs of the CT logs is publicly available, the log IDs, which can be observed in SCTs, require further information. Specifically, the objective is to determine the URL to be accessed when a specific log ID is observed in an SCT. To that end, a JSON file containing a list of CT logs, maintained by Google, is retrieved. To enhance the accessibility of the required data during runtime, a dictionary is employed to map each log ID to its corresponding URL. Some of the URLs are hard-coded in our code. In such instances, Google hosts a backup of the CT log in question, and the original log is unresponsive to requests. Subsequently, another dictionary is initialized and populated with data regarding the STH (Signed Tree Head) for each log URL. This is achieved through the use of the CT log's API, which facilitates the retrieval of the most recent Signed Tree Head (STH). The dictionary is instrumental in enabling the program to run more swiftly, as the STH is updated earliest after a period of half an hour.

Subsequently, the parallelizer, Ray, is initialized by launching a local Ray cluster, which enables the execution of tasks in parallel across multiple cores. Next, an instance of the class responsible for statistical processing is created. Afterwards, the directory in which the certificates are stored is specified, indicating the location from which the files are to be processed. The directory name is then encoded into bytes, and a list is created that includes every file present in the directory. Subsequently, counters are initialized to facilitate the monitoring of the number of failed and successfully verified files throughout the process. Then, a list is initialized, which serves to store the Ray processes. The responsibility of each process is to verify a single certificate.

In the initial stage of the verification process, a counter is set up to record the number of SCTs in a certificate that can be verified. The duration of the process is recorded for the purpose of statistical analysis and estimation of time requirements. The program then opens the file in which the certificate is stored. The file is read and parsed as PEM-encoded data using the pem library. This allows for the differentiation of multiple certificates within a single file. This is the case when a certificate chain is stored in a file. Subsequently, the parsed PEM data is converted into a list of X.509 certificate objects. The certificates are converted into a format that is readily accessible due to the utilization of the cryptography library. For future reference, a SHA-256 hash is calculated over the leaf certificate in order to obtain its fingerprint. In the event that more than one certificate is identified within a given file, it is assumed that the second certificate represents the issuer's certificate. In the event that the file contains a single certificate, an alternative method must be employed to obtain the issuer's certificate.

To exclude certificates without SCTs, a more comprehensive examination of the leaf certificate is necessary. To this end, the cryptography library is employed to identify the specific extension containing the SCTs. Typically, these are stored as part of the certificate's extensions, specifically under the Precertificate SCT extension. In the event that no SCTs are identified, the verification process is terminated and the next certificate is evaluated.

In the event that the issuer certificate is not present within the certificate chain, the Authority Information Access (AIA) extension from the given certificate is utilized, as it contains pertinent information regarding the certificate's issuer. The aforementioned extension is then examined for the purpose of locating URLs that allow the download of the issuer's certificate. In the event that a URL is identified, a subsequent step is to examine whether a local cache of the issuer certificate is available. In the event that a cached certificate is identified, it is loaded from the disk. The absence of a cached certificate prompts the system to attempt a download from the specified URL, with the objective of subsequently caching the downloaded certificate locally. The implementation of this caching strategy enables enhanced time efficiency and the avoidance of being blocked due to an excessive number of requests.

In the event of an unsuccessful outcome at any stage, an error message is generated and the verification process is marked as unsuccessful. This is due to the fact that it is not possible to ascertain the verification status unless the issuer's certificate is available.

For each SCT identified within the certificate, the program extracts the log ID, which serves to identify the CT log that issued the SCT. Additionally, the SCT's timestamp, indicating the date and time of submission to the CT log, is extracted. These two pieces of information are required for the completion of the verification process.

Subsequently, for each identified SCT, an assessment is conducted to determine if it originates from a test CT log. These logs are utilized for testing purposes and lack the necessary infrastructure and functionality to facilitate inclusion verification. Consequently, these certificates are flagged as successfully verified and the fact that they are associated with test CT logs is logged.

Afterwards, the program attempts to identify the corresponding CT log URL associated with the retrieved log ID. In the event that the log ID is not known, the verification of the specific SCT is documented as unsuccessful. In the event that the CT log URL is known, the tree size of the log is then retrieved. The tree sizes are stored in a cache for a maximum of 30 minutes. In the event that the tree size is outdated, the program initiates an HTTP request to the CT log's API with the objective of acquiring the most recent STH. In the event that the tree size could not be retrieved, the failure is documented and the process continues with the next SCT.

Subsequently, the precertificate is derived from the actual certificate by removing any SCT-related extensions. Additionally, the issuer's certificate is retrieved, its public key is obtained, and a SHA-256 hash is calculated over it. The final step is to calculate the leaf hash, which is necessary for the API to identify whether there is an entry in the CT log that corresponds to the provided certificate. The leaf hash is calculated over a byte array that includes the timestamp from the SCT, the issuer's public key hash, the precertificate, and some constants. With the requisite leaf hash in hand, an HTTP request is made to the CT log via the get-proof-by-hash endpoint. The request transmits the hash value and a recent size of the CT log.

If the response from the API is successful (status code 200), it can be confirmed that the certificate's SCT is included in the CT log. In the event of a failure, a second hash is calculated using a timestamp that has been shifted by one hour. This is done because the cryptography library employs the use of datetime objects for the interpretation of the SCTs' timestamps. In some rare cases, the conversion from a datetime object back to a timestamp in milliseconds does not function as intended, resulting in a shift of one whole hour. Subsequently, in the event of a second unsuccessful request, the certificate is deemed to be not included in the specified log. Following the inclusion check, the result (whether successful or unsuccessful) is documented.

Subsequently, the code calculates the total time taken for the verification process, with the objective of providing some statistics about performance after the program has completed. The final step in the process is the return of a successful flag, which indicates whether the inclusion of a certificate can be verified.

Ultimately, within a loop, Ray's wait function awaits the completion of at least one task and subsequently returns a list comprising both completed tasks and those that are still in progress. This approach allows for the monitoring of verification status progress. Upon completion of tasks, their results are retrieved and displayed. Once all tasks have been completed, final statistics are saved. The time required for the verification of a single certificate inclusion is measured, as are the number of verification failures and the reasons for these failures. Additionally, the number of successful verifications is recorded.

Following the processing and saving of all results, the Ray cluster is shut down to clean up resources.

In practice, we are able to verify an average of one certificate every 1.04 seconds 4.1.

5 Results

This chapter presents a detailed account of the practical performance of our implementation, accompanied by a comprehensive statistical analysis of the verification processes. We differentiate between the certificates/SCTs collected through IP scanning and those obtained through the Tranco list.

5.1 Gathering of certificates

First, we scan the IPv4 address space for hosts that might provide a TLS certificate. We use ZMap as described in chapter 4. Using this method, we generate a list of 2,934,271 unique IPv4 addresses. These addresses appear to have an open 443 port and need to be investigated.

Using ZGrab, we try to collect an X.509 certificate from each IPv4 address in the list. Of the nearly 3 million IP addresses, ZGrab was able to complete a handshake with 898,799 of the hosts and collect digital certificates from them.

Second, using the Tranco list, we have potentially 1 million domains from which to collect certificates. We use OpenSSL to attempt to store the corresponding certificates from these sites. Using this method, we are able to download 789,045 certificates from the list.

Our dataset therefore contains 1687844 certificates in total. In the following, we verify the certificates from the IPv4 scan separately from those we collected using the Tranco list.

5.2 Verifying SCTs

Due to technical and temporal constraints, it has not been feasible to verify the entirety of the collected certificates. Due to temporal constraints, only 711,349 of the certificates collected through IPv4 crawling could be processed. The Tranco list approach permitted the processing of 722,983 certificates.

5.2.1 IPv4

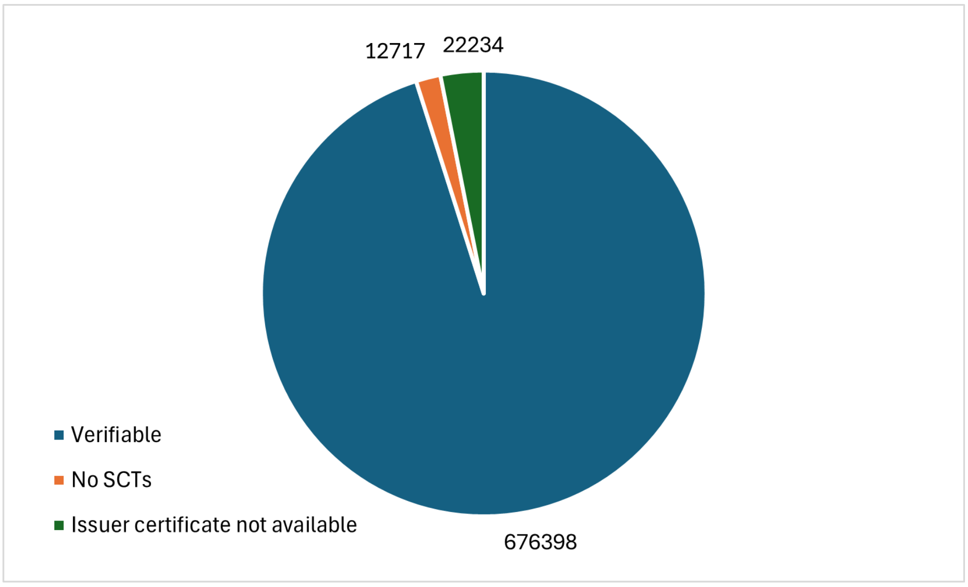

Figure 5.1: Categorization of IP scan certificates in terms of our ability to verify their SCTs. Out of all collected certificates, 12,717 have no SCT and therefore no promise of inclusion in CT logs. For 22,234 certificates the issuer certificates could not be retrieved. We can attempt to verify the SCTs of the remaining 676,398 certificates.

In Figure 5.1, we present a visual representation of our findings regarding certificates.

A subset of certificates is deemed irrelevant for the purposes of this research, as they contain no SCTs and therefore do not constitute a promise of inclusion in a CT Log. It should be noted that these certificates are not subject to further processing.

Another portion of the identified certificates can not be validated through the API. This is due to the fact that the server responsible for providing the issuer's certificate is not responding to our queries. In the absence of the issuer's certificate, it is not possible to perform the inclusion verification of a precertificate, which is a prerequisite for the construction of the corresponding Merkle-Tree hash. The inability to reach the server can be attributed to a number of potential causes. Given the extensive nature of our scanning activities, it is plausible that the server in question has implemented a block or limitation on our access. Alternatively, the server may have become unresponsive to incoming requests. It is not possible to distinguish between these two scenarios with certainty.

The remaining certificates are processed individually, with consideration given to the SCTs they provide.

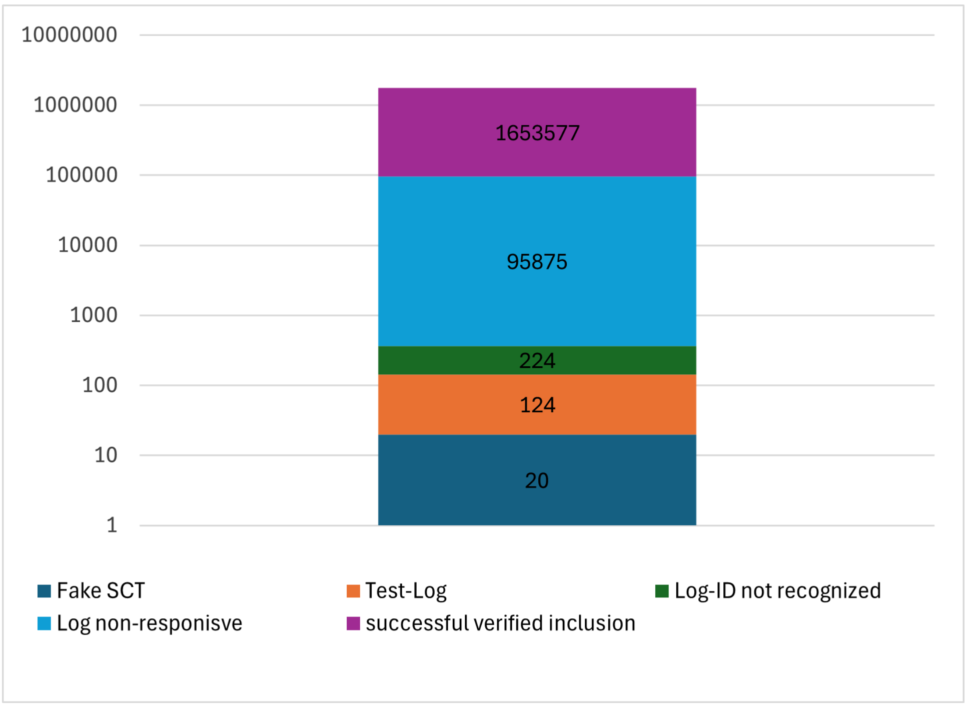

Figure 5.2: Categorization of SCTs from verifiable certificates (logarithmic scale). In total, we were able to process 1,749,820 SCTs from different certificates. 20 of these SCTs do not have a valid signature, 124 SCTs are from test CT logs, 224 SCTs contain log IDs that we do not recognize, and 95,875 SCTs are signed by logs that are no longer responding. The remaining 1,653,577 SCTs can be successfully verified by us.

The verification of individual SCTs enabled the identification of special properties associated with some of them. It is evident that there are SCTs that can be successfully verified. In contrast, we have identified a subset of SCTs for which we are unable to confirm the inclusion of their associated certificates in the CT log. There are a number of reasons why a SCT may not be able to be verified in a satisfactory manner.

Firstly, it is possible that a certificate has been included in a test CT log. The functionality of these logs is limited to the party that provides them with a certificate. Such a system provides a promise of inclusion for the given certificate but is unable to demonstrate proof of inclusion, as it was not designed with this specific task in mind.

It can be reasonably assumed that certificates containing such SCTs from Test-CT logs are also certificates issued for testing purposes.

Secondly, there are what we refer to as "fake SCTs." It is unclear for what purposes these certificates are utilized. The SCTs in question bear a signature that is defective and therefore cannot be validated. To be more precise, the signature in question cannot be verified using the public key of the CT log that is purported to have issued the SCT.

Thirdly, there are SCTs for which the corresponding CT log is not identifiable, as it is not included in the list of all known and announced logs, which is maintained by Google. For purposes of comparison and reference, the gstatic-list is utilized by us for all log IDs. Verification of these SCTs is not attainable, as the path of the API and the public key of the CT log associated with the specified log ID remain unidentified [all].

Ultimately, verification of the inclusion promise is not possible if the CT log that provided the promise is no longer accessible. Some SCTs were issued by CT logs that are no longer online, which prevents verification of the SCTs from these non-responsive logs. The distribution of properties of the corresponding SCTs on a logarithmic scale is shown in Figure 5.2, which presents the SCTs gathered while scanning the IPv4 space.

For the certificates collected while scanning the IPv4 space, Figure 5.2 shows the distribution of the properties of the corresponding SCTs on a logarithmic scale.

The presented statistics offer no evidence of any irregularities or misbehavior in the CT logs.

5.2.2 Tranco's one million

In Figure 5.3, we present a breakdown of the distribution of relevant and non-relevant certificates that we intend to investigate further. The data is presented in a format analogous to that used for the IP scan.

For the certificates that are relevant to our investigation, we extract the SCTs and subject them to further processing.

The distribution of different SCTs and their corresponding properties is provided in Figure 5.4. The structure is analogous to that employed for the IP scan.

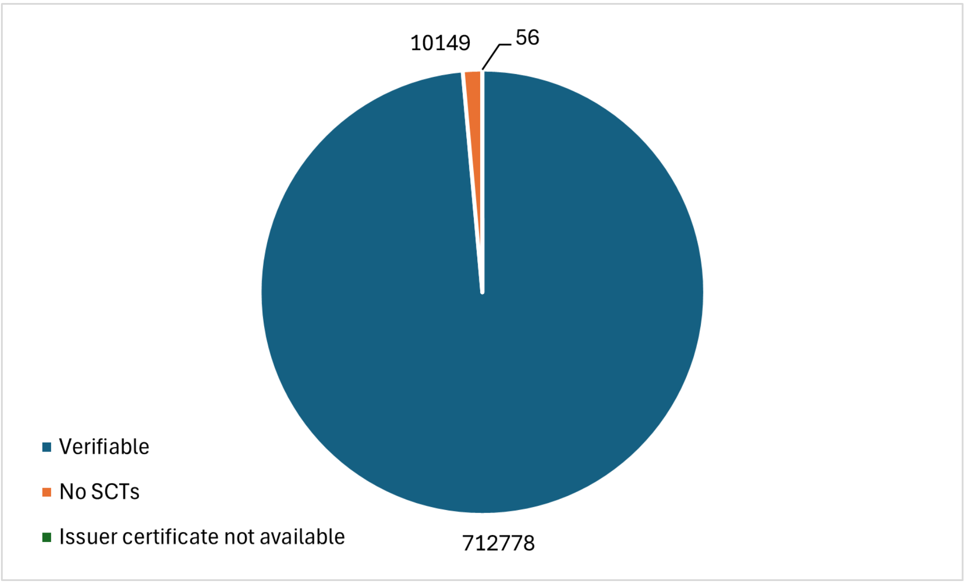

Figure 5.3: Categorization of certificates from the Tranco list in terms of our ability to verify their SCTs. Of all the certificates collected, 10,149 have no SCT and therefore no promise of inclusion in CT logs. For 56 certificates, the issuer certificates could not be retrieved. We can attempt to verify the SCTs of the remaining 712,778 certificates.

5.3 Downloading a CT Log

Although the process of verifying an SCT using a local CT log is the most temporally efficient, as evidenced by the data presented in Figure 4.1, it is not without its drawbacks. That is the process of downloading the entirety of a CT log. In order to conduct our research, we downloaded the Nessie2025 log provided by DigiCert. At the beginning of the process, the log comprised approximately 147 million individual entries. By the conclusion of the process, the number of entries had increased to approximately 189 million. It should be noted that not every single certificate was downloaded; only the precertificates were obtained. This is due to the fact that, while it is required of CAs to provide at least the precertificate, they are not obliged to include the certificate which has the SCT as an extension in the CT log. The precertificates alone comprise approximately 72% of the CT log. The download of the approximately 136 million precertificates took 30 days in total and occupies 46 gigabytes of space. It can be seen that this method is therefore limited in its usage. While there are many scenarios in which downloading is beneficial, it should be noted that a lengthy download must be completed beforehand.

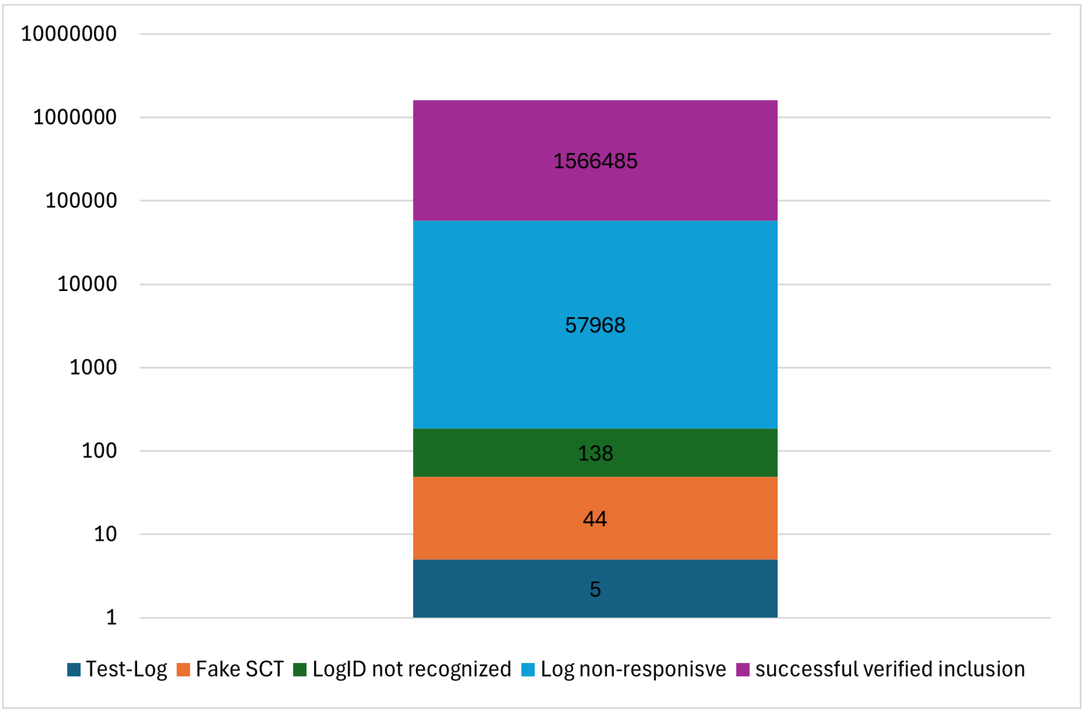

Figure 5.4: Categorization of SCTs from verifiable certificates (logarithmic scale). In total, we were able to process 1,624,640 SCTs from different certificates. 5 of these SCTs are from test CT logs, 44 SCTs do not have a valid signature, 138 SCTs contain log IDs that we do not recognize, and 57968 SCTs are signed by logs that are no longer responding. The remaining 1,566,485 SCTs can be successfully verified by us.

6 Discussion

In the course of our research, we encountered issues that we believe are worthy of further attention, as they may also affect those who are interested in pursuing this topic in greater depth. In the following section, we will provide a brief overview of some of the more notable issues.

6.1 Rate limiting and blacklisting

The issue of rate limiting and blacklisting is a common occurrence in the context of large-scale network traffic. Although this is an effective method for maintaining server stability, it does result in more time-consuming and resource-intensive auditing of critical web ecosystem components. It is also noteworthy that different providers of CT logs have implemented limits that vary in their specifications. For instance, while DigiCert and Cloudflare impose a limit of 1,024 entries per request for users accessing their CT logs, Google restricts the number of entries in each batch to 32. This has a significant impact on the rate at which an auditor is able to download a CT log. Fortunately, this limit did not present a significant challenge during the verification process of the SCTs, as the preparation of the issuer's certificate and the leaf hash required so much time in fact that we did not have to worry about rate limiting.

A more significant challenge was encountered with regard to blacklisting, as this phenomenon was observed in the context of multiple servers, specifically those responsible for distributing issuer and intermediate certificates. As previously stated in chapter 2, in order to obtain the leaf hash, it is necessary to possess the certificate issued by the relevant issuer. Blacklisting by these provisioning servers results in the inability to test SCTs of the corresponding certificate. We attempted to mitigate this by consistently saving intermediate certificates, which did reduce the frequency of occurrences but did not entirely prevent them. This significantly impacted our ability to verify a substantial number of certificates. Unfortunately, it is not immediately evident whether the server is no longer responsive or if we have been blacklisted.

6.2 Timely inclusion

Another point that we initially intended to investigate is the timing of inclusion in a CT log. Specifically, the time interval between the issuance of the certificate to the CT log provider and its actual incorporation into the log. Although the monitors are already tasked with ensuring that the MMD has been met, it would also be beneficial to investigate the amount of time that elapses between the acquisition of a promise for inclusion and the actual inclusion. It is unfortunately not possible to calculate the exact time stamp for older certificates without the information being saved by a monitor.

6.3 Independent code for using API

Despite the relevance of Certificate Transparency, there is a relatively limited repository of code utilizing the API of the logs. It is challenging to identify even basic examples of how to utilize the API of the CT logs. Moreover, there is a limited awareness of this particular aspect of the web. It is regrettable that there is a lack of available libraries. The use of multiple sources for source code can enhance the reliability and integrity of the codebase through cross-validation. Furthermore, a diversity of sources can also lead to different approaches and innovations, which in turn encourages the development of features, optimizations, and enhancements. Currently, the Google source code represents the primary available option. However, this may entail certain risks associated with centralized control.

6.4 Mirrors of CT logs

The research would have been concluded with a greater number of SCTs, for which the corresponding CT logs are no longer accessible online. However, for some of these CT logs that are already offline and do not provide any inclusion verification, there exist mirrors provided by Google. The act of mirroring offline CT logs offers certain advantages with regard to the continuity of the CT ecosystem. Primarily, even in the event that the original log is no longer accessible, the verification process can still be conducted. This enhances the reliability of the certificates that, in the absence of this measure, would fail to meet the technical requirements of browsers. It is encouraging to observe that Google is maintaining the logs for the benefit of end users.

7 Conclusion

This thesis examines the reliability and compliance of Signed Certificate Timestamps (SCTs) within the broader framework of certificate transparency. It explores the extent to which SCTs provide a trustworthy foundation for security within the modern web ecosystem. In the preceding chapters, we have presented our methodology, data, statistical analysis, and findings. This leads to the conclusion that no evidence was found to suggest that any SCTs failed its promise towards the user. Verification of the SCTs revealed that each one was indeed backed by an actual entry in the corresponding CT log.

It should be noted that not all SCTs could be fully verified for compliance. However, this is attributed to external factors, such as inaccessible services and servers. However, the absence of evidence for misbehavior does not constitute proof of absolute trustworthiness. The thesis demonstrated that a significant number of SCTs do comply and that there is no reason to doubt their individual promises.

It is imperative to exercise a critical eye towards this mechanism, given its significant influence within the modern web ecosystem. However, in light of these findings, it is reasonable for users to adopt a degree of confidence in these mechanisms.

The research presented here is limited in temporal scope. Further research may encompass larger and more diverse datasets, incorporating sources that may reveal a fundamentally different class of certificates. A further avenue for future research could be to enhance the verification process, thereby increasing its speed and reliability. This could be achieved by refining the proposed code to address potential flaws and technical issues. As an alternative, the code provided by Google's library could be edited to align it more closely with the researcher's objectives.

A Appendix

```
sudo zmap -p 443 -o results.csv
```

Listing A.1: ZMap crawling ports 443

```
sudo zgrab2 tls --port 443 --input-file results.csv --output-file certificates.json
```

Listing A.2: ZGrab saving X.509 certificates

```python
import json
def prepare_certificates_json(cert_path):
    with open(cert_path, 'rb') as cert_file:
        curr = 0
        for line in cert_file:
            curr += 1
            entry = json.loads(line.strip())
            if curr%5000 == 0:
                print(f"{curr} Lines geparsed.")
            if entry["data"]["tls"]["status"] == 'success':
                try:
                    with open(f"certificates/{entry['data']['tls']['result']['handshake_log']['server_certificates']['certificate']['parsed']['fingerprint_sha256']}.txt", "w") as f:
                        f.write(f"-----BEGIN CERTIFICATE-----\n{entry['data']['tls']['result']['handshake_log']['server_certificates']['certificate']['raw']}\n-----END CERTIFICATE-----")
                except KeyError:
                    continue

prepare_certificates_json("certificates.json")
```

Listing A.3: Pre-process certificates crawled with ZGrab

```
shuf ../domains.txt | tr -d '\r' | parallel --eta -j 16 "timeout 5s openssl s_client -connect {}:443 -showcerts -servername {} 2>/dev/null </dev/null | sed -ne '/-BEGIN CERTIFICATE-/,/-END CERTIFICATE-/p' > {}.pem"
```

Listing A.4: Download certificates from Tranco list

```go
package main

import (
    "context"
    "crypto/sha256"
    "crypto/x509"
    "encoding/hex"
    "fmt"
    "io/ioutil"
    "log"
    "os"
    "path/filepath"
    ct "github.com/google/certificate-transparency-go"
    "github.com/google/certificate-transparency-go/client"
    "github.com/google/certificate-transparency-go/scanner"
    "github.com/google/certificate-transparency-go/jsonclient"
    "net/http"
)

func calculateFingerprint(certDER []byte) string {
    hash := sha256.Sum256(certDER)
    return hex.EncodeToString(hash[:])
}

func saveEntryNumber(fingerprint string, directory string, index int64) error {
    filename := filepath.Join(directory, fmt.Sprintf("%s.txt", fingerprint))
    content := fmt.Sprintf("Entry number: %d", index)
    return ioutil.WriteFile(filename, []byte(content), 0644)
}

func main() {
    saveDirectory := "/mnt/measures/Nessie2025/content"
    os.MkdirAll(saveDirectory, os.ModePerm)

    logURL := "https://nessie2025.ct.digicert.com/log/"
    httpClient := &http.Client{}
    logClient, err := client.New(logURL, httpClient, jsonclient.Options{})
    if err != nil {
        log.Fatalf("Failed to create log client: %v", err)
    }

    opts := scanner.DefaultScannerOptions()
    opts.Matcher = &scanner.MatchAll{}
    opts.PrecertOnly = true
    //opts.NumWorkers = 30  // Increase the number of concurrent workers

    logScanner := scanner.NewScanner(logClient, *opts)

    foundCert := func(entry *ct.RawLogEntry) {
        logEntry, err := entry.ToLogEntry()
        if err != nil {
            log.Printf("Failed to parse log entry at index %d: %v",
↪ entry.Index, err)
            return
        }

        if logEntry.X509Cert != nil {
            fingerprint := calculateFingerprint(logEntry.X509Cert.Raw)
            err := saveEntryNumber(fingerprint, saveDirectory, entry.
↪ Index)
            if err != nil {
                log.Printf("Failed to save entry number for
↪ certificate at index %d: %v", entry.Index, err)
            } else {
                fmt.Printf("Saved certificate entry number at index %d
↪  with fingerprint %s\n", entry.Index, fingerprint)
            }
        }
    }

    foundPrecert := func(entry *ct.RawLogEntry) {
        logEntry, err := entry.ToLogEntry()
        if err != nil {
            log.Printf("Failed to parse log entry at index %d: %v",
↪ entry.Index, err)
            return
        }

        if logEntry.Precert != nil {
            tbsCertDER, err := x509.MarshalPKIXPublicKey(logEntry.
↪ Precert.TBSCertificate.PublicKey)
            if err != nil {
                log.Printf("Failed to marshal TBS certificate at index
↪  %d: %v", entry.Index, err)
                return
            }
            fingerprint := calculateFingerprint(tbsCertDER)
            err = saveEntryNumber(fingerprint, saveDirectory, entry.
↪ Index)
            if err != nil {
                log.Printf("Failed to save entry number for
↪ precertificate at index %d: %v", entry.Index, err)
            } else {
                fmt.Printf("Saved precertificate entry number at index
↪  %d with fingerprint %s\n", entry.Index, fingerprint)
            }
        }
    }

    err = logScanner.Scan(context.Background(), foundCert,
↪ foundPrecert)
    if err != nil {
        log.Fatalf("Error during scanning: %v", err)
    }
```

```
}
```

Listing A.5: Go code for scanning and saving certificates from Nessie2025

```python
from cryptography import x509
from cryptography.hazmat.backends import default_backend
import base64
from cryptography.hazmat.primitives.serialization import Encoding, PublicFormat
from cryptography.x509.oid import ExtensionOID
import hashlib
import time
import os

def load_certificate(cert_data):
    cert = x509.load_der_x509_certificate(base64.b64decode(cert_data), default_backend())
    return cert

def extract_scts_from_certificate(cert_pem):
    try:
        scts = cert_pem.extensions.get_extension_for_oid(ExtensionOID.PRECERT_SIGNED_CERTIFICATE_TIMESTAMPS).value
        sct_list = []
        for sct in scts:
            sct_list.append(base64.b64encode(sct.log_id).decode('ascii'))
        return sct_list
    except:
        return None

def public_key_hash(cert_pem):
    return hashlib.sha256(cert_pem.public_key().public_bytes(Encoding.DER, PublicFormat.SubjectPublicKeyInfo)).digest().hex()

def current_milli_time():
    return round(time.time() * 1000)


def inclusion_proof(cert_path):
    start = current_milli_time()

    with open(cert_path, "rb") as f:
        data = pem.parse(f.read())
        certificates = [x509.load_pem_x509_certificate(single.as_bytes(), default_backend()) for single in data]

    cert = certificates[0]

    fingerprint = public_key_hash(cert)

    scts = extract_scts_from_certificate(cert)
    if scts == None:
        end = current_milli_time()
        return False, end - start

    successful = True
    for log_id in scts:

        if log_id == "TnWjJ1yaEMM4W2zU3z9S6x3w4I4bjWnAsfpksWKaOd8=":
            if not os.path.isfile(f"/mnt/measures/Argon2025h1/content
    ↪ /{fingerprint}.txt"):
                successful = False
            continue

        elif log_id == "5tIxY0B3jMEQQQbXcbnOwdJA9paEhvu6hzId/R43jlA=":
            if not os.path.isfile(f"/mnt/measures/Nessie2025/content/{
    ↪ fingerprint}.txt"):
                successful = False
            continue

        elif log_id == "fVkeEuF4KnscYWd8Xv340IdcFKBOlZ65Ay/ZDowuebg=":
            if not os.path.isfile(f"/mnt/measures/Yeti2025/content/{
    ↪ fingerprint}.txt"):
                successful = False
            continue

    with open("Logs/log.txt", "a") as f:
        if successful:
            f.write(f"{fingerprint} wurde erfolgreich bei allen
    ↪ angegebenen CT Logs verifiziert.\n")
        else:
            f.write(f"Bei der Verifizeriung von {fingerprint} ist ein
    ↪ Fehler aufgetreten.\n")

    end = current_milli_time() - start
    return end


if __name__ == "__main__":

    dir_name = "/home/luis/ip_scan/Remaining"
    directory = os.fsencode(dir_name)

    dir_content = os.listdir(directory)

    size = len(dir_content)
    counter = 0

    timer = 0

    for file in dir_content:
        counter += 1
        filename = os.fsdecode(file)

        start = current_milli_time()

        end = inclusion_proof(f"{dir_name}/{filename}")

        timer += end
        if counter % 10000 == 0:
            with open("Logs/average.txt", "w") as f:
                f.write(f"Averagely {timer/counter} ms for
    ↪ verification of one certificate.")
```

Listing A.6: Verifying inclusion in local CT log

```bash
#!/bin/bash
count=0
# Path to the directory in which the PEM files are located
pem_dir="/home/luis/top1m/certificates"

# Text file with the PEM files
filelist="/home/luis/top1m/fails/file.txt"

# Check whether the text file exists
if [ ! -f "$filelist" ]; then
  echo "File $filelist not found!"
  exit 1
fi

# Run through each line of the text file
while IFS= read -r filename; do
  # Full path specification for the PEM file
  full_path="$pem_dir/$filename"

  count=$((count + 1))

  # Check whether the file exists
  if [ -f "$full_path" ]; then
    # Execute the go command
    go run sctcheck.go "$full_path"

done < "$filelist"
```

Listing A.7: sctchecker for failed certificates

```python
from cryptography import x509
from pyasn1.codec.der.decoder import decode as asn1_decode
from pyasn1.codec.der.encoder import encode as asn1_encode
from pyasn1_modules import rfc5280
from cryptography.hazmat.backends import default_backend
import requests
from cryptography.hazmat.primitives import serialization
import datetime
import base64
from cryptography.hazmat.primitives.serialization import Encoding,
    ↪ PublicFormat
from cryptography.x509.oid import ExtensionOID
from cryptography.hazmat.primitives import hashes
import hashlib
import time
import json
import os
import ray
import pem


@ray.remote
class GlobalState:
    def __init__(self):
        self.total_time_spent = 0
        self.number_samples = 0
        self.failing_reasons = {'1': {'count':0, 'desc':"Issuer konnte nicht ermittelt/heruntergeladen werden", 'ids':[]},
                                '2': {'count':0, 'desc':"Zertifikat besitzt keine Precertificate-SCTs"},
                                '3': {'count':0, 'desc':"Eine LogID ist nicht bekannt", 'ids':[]},
                                '4': {'count':0, 'desc':"Ein Log konnte nicht erreicht werden", 'ids':[]},
                                '5': {'count':0, 'desc':"Zertifikat ist nicht inkludiert", 'ids':[]},
                                '6': {'count':0, 'desc':"Zertifikat nutzt Test-Log(s). Wird immer als in Ordnung angesehen.", 'ids':[]}
                                }

    def update_time_and_samples(self, time_spent):
        self.total_time_spent += time_spent
        self.number_samples += 1

    def update_failing_reasons(self, reason_key, log_id=None):
        self.failing_reasons[reason_key]['count'] += 1
        if log_id:
            if(log_id not in self.failing_reasons[reason_key]['ids']):
                self.failing_reasons[reason_key]['ids'].append(log_id)

    def get_state(self):
        return self.total_time_spent, self.number_samples, self.failing_reasons



def convert2precert(cert_pem):
    certasn1 = asn1_decode(cert_pem.tbs_certificate_bytes, asn1Spec=rfc5280.TBSCertificate())[0]

    newExts = [ext for ext in certasn1["extensions"] if str(ext["extnID"]) not in ("1.3.6.1.4.1.11129.2.4.2", "1.3.6.1.4.1.11129.2.4.3")]
    certasn1["extensions"].clear()
    certasn1["extensions"].extend(newExts)
    return asn1_encode(certasn1)

def get_issuer_certificate(aia_extension):
    try:
        urls = []
        for access_description in aia_extension.value:
                if access_description.access_method == x509.AuthorityInformationAccessOID.CA_ISSUERS:
                    urls.append(access_description.access_location.value)

    except x509.ExtensionNotFound:
        return

    if len(urls) != 1 : return None

    url_hash = hashlib.sha1(urls[0].encode()).digest().hex()
    cert_path = f"./issuer_certs/{url_hash}.txt"
    if os.path.isfile(cert_path):
        with open(cert_path, "rb") as f:
            issuer_cert = f.read()
        try:
            issuer_cert_obj = x509.load_der_x509_certificate(bytes.fromhex(str(issuer_cert)[2:-1]), default_backend())
            issuer_cert_pem = issuer_cert_obj.public_bytes(encoding=serialization.Encoding.PEM).decode()

            return x509.load_pem_x509_certificate(issuer_cert_pem.encode(), default_backend())
        except:
            pass

    try:
        response = requests.get(urls[0])
        issuer_cert = response.content
        with open(cert_path, "w") as f:
            f.write(issuer_cert.hex())
        issuer_cert_obj = x509.load_der_x509_certificate(issuer_cert, default_backend())
        issuer_cert_pem = issuer_cert_obj.public_bytes(encoding=serialization.Encoding.PEM).decode()

        return x509.load_pem_x509_certificate(issuer_cert_pem.encode(), default_backend())
    except Exception as e:
        print(e)
        return

def extract_scts_from_certificate(cert_pem):
    try:
        scts = cert_pem.extensions.get_extension_for_oid(ExtensionOID.PRECERT_SIGNED_CERTIFICATE_TIMESTAMPS).value
        sct_list = []
        for sct in scts:
            sct_entry = {
                'log_id': base64.b64encode(sct.log_id).decode('ascii'),
                'timestamp': sct.timestamp
            }
            sct_list.append(sct_entry)
        return sct_list
    except:
        return None

def public_key_hash(cert_pem):
    return hashlib.sha256(cert_pem.public_key().public_bytes(Encoding.
    ↪ DER, PublicFormat.SubjectPublicKeyInfo)).digest()


def hash_leaf_precert(timestamp, issuer_key, cert):
    hash_data = bytearray()
    hash_data.append(0)
    hash_data.append(0)
    hash_data.append(0)
    hash_data.extend(timestamp.to_bytes(8, 'big'))
    hash_data.extend((1).to_bytes(2,'big'))
    hash_data.extend(issuer_key)
    hash_data.extend(len(cert).to_bytes(3,'big'))
    hash_data.extend(cert)
    hash_data.extend((0).to_bytes(2,'big')) #extensions

    return hashlib.sha256(hash_data).digest()

def find_log_url(log_id):
    try:
        return log_dict[log_id]
    except Exception as e:
        print(e)
        return None

def initialize_tree_size(log_url, timeout = None):
    global tree_heads
    try:
        sth = get_log_info(log_url, timeout)
    except:
        sth = None
    if sth == None:
        return None
    tree_heads[log_url]['timestamp'] = sth['timestamp']
    tree_heads[log_url]['tree_size'] = sth['tree_size']
    return tree_heads[log_url]['tree_size']

def get_tree_size(log_url, timeout = None):
    global tree_heads
    ts = tree_heads[log_url]['timestamp']
    if ts == None: return None
    if ((current_milli_time() - ts) < 30*60000):
        try:
            sth = get_log_info(log_url, timeout)
        except:
            sth = None
        if sth == None:
            return None
        tree_heads[log_url]['timestamp'] = sth['timestamp']
        tree_heads[log_url]['tree_size'] = sth['tree_size']
    return tree_heads[log_url]['tree_size']

def get_log_info(log_url, timeout):
    response = requests.get(f"{log_url}/ct/v1/get-sth", timeout=(
    ↪ timeout, None))
    if response.status_code != 200:
        return None
    return response.json()

def in_test_log(log_id):
    if log_id in ["w78Dp+HKiEHGB7rj/0Jw/KXsRbGG675OLPP8d4Yw9fY=",
                  "UutLIl7IlpdIUGdfI+Q7wdAh4yFM5S7NX6h8IDzfygM=",
                  "C3YOmouaaC+ImFsV6UdQGlZEa7qIMHhcOEKZQ4ZFDAA=",
                  "H8cs5aG3mfQAw1m/+WyjkTVI6GRCIGEJUum6F3T3usc=",
                  "o8mYRegKt84AFXs3Qt8CB90nKytgLs+Y7iwS25xa5+c=",
                  "aXqvyhprU2+uISBQRt661+Dq6hPSQy5unY+zefK5qvM=",
                  "+X6XuNM+96FZAqU6GeF5kOXcQGoDGCW6rZPpj5ucacs=",
                  "sMyD5aX5fWuvfAnMKEkEhyrH6IsTLGNQt8b9JuFsbHc=",
                  "YukAYASjB5VadUS01YSpYmjKHW5Fha3wkW3+X9wfBNs=",
                  "MCTOfusWiGJyS+pwLv/5ks/kVkNBkapZWyX4AibIABc=",
                  "P+HLRu1HNXmvAUH5ck2dxENHLXVuhedxnFWCSF3U4eQ=",
                  "JgI5SIdM9/zQ+2RxpD6EfrsgCubi+iQjbfbRpgZjD7E=",
                  "yEuQege+qimmFMJFhLej9mJDlGh7Jf5ig4tx7EIq0vk="]:
        return True
    return False

def current_milli_time():
    return round(time.time() * 1000)


@ray.remote
def inclusion_proof(cert_path, global_state):
    start = current_milli_time()

    with open(cert_path, "rb") as f:
        data = pem.parse(f.read())
        certificates = [x509.load_pem_x509_certificate(single.as_bytes
    ↪ (), default_backend()) for single in data]
    if len(certificates) == 0:
        with open("/mnt/measures/log_verify/log.txt", "a") as f:
            f.write(f"Die Datei unter {cert_path} enthaelt keine
    ↪ Zertifikate.\n")
        return False

    cert = certificates[0]
    issuer = None
    if len(certificates) > 1:
        issuer = certificates[1]

    cert
    fingerprint = cert.fingerprint(hashes.SHA256()).hex()

    scts = extract_scts_from_certificate(cert)
    if scts == None:
        ray.get(global_state.update_failing_reasons.remote('2'))

        with open("/mnt/measures/log_verify/log.txt", "a") as f:
            f.write(f"Es sind keine SCTs in {fingerprint}.\n")
        return False

    if issuer == None:
        try:
            issuer = get_issuer_certificate(cert.extensions.
   ↪ get_extension_for_class(x509.AuthorityInformationAccess))
        except:
            pass
    if issuer == None:
        ray.get(global_state.update_failing_reasons.remote('1',
   ↪ fingerprint))

        with open("/mnt/measures/log_verify/log.txt", "a") as f:
            f.write(f"Issuer fuer {fingerprint} konnte nicht ermittelt
   ↪ /heruntergeladen werden.\n")
        return False

    successful = True
    for sct in scts:

        try:
            if in_test_log(sct.get("log_id")):
                ray.get(global_state.update_failing_reasons.remote('6'
   ↪ , fingerprint))
                with open("/mnt/measures/log_verify/log.txt", "a") as
   ↪ f:
                    f.write(f"{fingerprint} benutzt Test-CT Log(s).\n"
   ↪ )
                return True
            ct_log_url = find_log_url(sct.get("log_id"))
            if ct_log_url == None:
                ray.get(global_state.update_failing_reasons.remote('3'
   ↪ , sct.get("log_id")))

                with open("/mnt/measures/log_verify/log.txt", "a") as
   ↪ f:
                    f.write(f"Die LogID {sct.get('log_id')} von {
   ↪ fingerprint} ist nicht bekannt.\n")
                successful = False
                continue
            if ct_log_url[-1] == "/":
                ct_log_url = ct_log_url[:-1]
            tree_size = get_tree_size(ct_log_url)

            if tree_size == None:
                ray.get(global_state.update_failing_reasons.remote('4'
    ↪ , sct.get('log_id')))

                with open("/mnt/measures/log_verify/log.txt", "a") as
    ↪ f:
                    f.write(f"Fuer {fingerprint} konnte Log {sct.get('
    ↪ log_id')} nicht erreichen.\n")
                successful = False
                continue

            hash = hash_leaf_precert(int(sct.get("timestamp").
    ↪ timestamp()*1000), public_key_hash(issuer), convert2precert(
    ↪ cert))
            params = {"hash": base64.b64encode(hash), "tree_size" :
    ↪ tree_size}
            response = requests.get(f"{ct_log_url}/ct/v1/get-proof-by-
    ↪ hash", params=params)
            if response.status_code != 200:
                hash = hash_leaf_precert(int(sct.get("timestamp").
    ↪ timestamp()*1000+3600000), public_key_hash(issuer),
    ↪ convert2precert(cert))
                params = {"hash": base64.b64encode(hash), "tree_size"
    ↪ : tree_size}
                response = requests.get(f"{ct_log_url}/ct/v1/get-proof
    ↪ -by-hash", params=params)

                if response.status_code != 200:
                    ray.get(global_state.update_failing_reasons.remote
    ↪ ('5', fingerprint))


                    with open("/mnt/measures/log_verify/log.txt", "a")
    ↪  as f:
                        f.write(f"{fingerprint} ist bei {sct.get('
    ↪ log_id')} nicht inkludiert.\n")
                    successful = False
        except Exception as e:
            ray.get(global_state.update_failing_reasons.remote('4',
    ↪ sct.get('log_id')))
            with open("/mnt/measures/log_verify/log.txt", "a") as f:
                f.write(f"Waehrend der Verifizierung von {fingerprint}
    ↪  ist folgender Fehler aufgetreten: {e}.\n")
            successful = False

    with open("/mnt/measures/log_verify/log.txt", "a") as f:
        if successful:
            f.write(f"{fingerprint} wurde erfolgreich bei allen
    ↪ angegebenen CT Logs verifiziert.\n")
        else:
            f.write(f"Bei der Verifizeriung von {fingerprint} ist ein
    ↪ Fehler aufgetreten.\n")

    end = current_milli_time()
    ray.get(global_state.update_time_and_samples.remote(end-start))

    return successful

def save_results(global_state, counter, size, fail, verified):
    (total_time_spent, number_samples, failing_reasons) = ray.get(
  ↪ global_state.get_state.remote())

    with open("./stats/failing_reasons.txt", 'w') as f:
        json.dump(failing_reasons, f, indent=4)

    if(number_samples != 0):
        with open("./stats/average_time.txt", 'w') as f:
            f.write(f"Durchschnittlich werden {total_time_spent/
  ↪ number_samples} Millisekunden gebraucht, um ein Zertifikat zu
  ↪ ueberpruefen.\nInsgesamte Zeit verbracht mit Verifizieren {
  ↪ total_time_spent};\nAnzahl Zertifikate, die in dieser Zeit
  ↪ ueberprueft wurden: {number_samples}\n")

    with open("./Logs/verified.txt", 'w') as f:
        f.write(f"{counter} von {size} geprueft. Davon {fail} failed
  ↪ und {verified} verifiziert\n")

    print(f"Saved results at {counter} certificates processed.")




if __name__ == "__main__":

    log_urls = "https://www.gstatic.com/ct/log_list/v3/all_logs_list.
  ↪ json"
    response = requests.get(log_urls)

    if response.status_code == 200:
        json_data = json.loads(response.text)
    else:
        print("all_logs_list konnte nicht geladen werden")

    with open("ct_logs.json", 'rb') as json_file:
        json_data = json.load(json_file)

    log_dict = {}

    for operator in json_data.get('operators', []):
        for log in operator.get('logs', []):
            log_id = log.get('log_id')
            url = log.get('url')
            if log_id and url:
                log_dict[log_id] = url
    log_dict["tz77JN+cTbp18jnFulj0bF38Qs96nzXEnh0JgSXttJk="] = "https
  ↪ ://ct.googleapis.com/logs/eu1/mirrors/letsencrypt_oak2023/"
    log_dict["36Veq2iCTx9sre64X04+WurNohKkal6OOxLAIERcKnM="] = "https
  ↪ ://ct.googleapis.com/logs/eu1/mirrors/letsencrypt_oak2022/"
    log_dict["BZwB0yDgB4QTlYBJjRF8kDJmr69yULWvO0akPhGEDUo="] = "https
  ↪ ://ct.googleapis.com/logs/eu1/mirrors/digicert_yeti2022_2/"
```

```python
    log_dict["QcjKsd8iRkoQxqE6CUKHXk4xixsD6+tLx2jwkGKWBvY="] = "https://ct.googleapis.com/logs/us1/mirrors/cloudflare_nimbus2022/"
    log_dict["ejKMVNi3LbYg6jjgUh7phBZwMhOFTTvSK8E6V6NS61I="] = "https://ct.googleapis.com/logs/us1/mirrors/cloudflare_nimbus2023/"
    log_dict["b1N2rDHwMRnYmQCkURX/dxUcEdkCwQApBo2yCJo32RM="] = "https://ct.googleapis.com/logs/us1/mirrors/comodo_mammoth/"
    log_dict["VYHUwhaQNgFK6gubVzxT8MDkOHhwJQgXL6OqHQcT0ww="] = "https://ct.googleapis.com/logs/us1/mirrors/comodo_sabre/"

    tree_heads = {}
    for id in log_dict:
        url = log_dict[id]
        if url[-1] == "/":
                url = url[:-1]
        tree_heads[url] = {'timestamp': None, 'tree_size': None}
    i = 0
    for item in tree_heads:
        i += 1
        initialize_tree_size(item, None)
        print(f"{i} von {len(tree_heads)} STHs gespeichert", end="\r")



    ray.init()
    global_state = GlobalState.remote()

    dir_name = "/home/talha/pems"
    directory = os.fsencode(dir_name)

    dir_content = os.listdir(directory)

    size = len(dir_content)
    fail = 0
    verified = 0
    counter = 0
    futures = []

    for file in dir_content:
        counter += 1
        filename = os.fsdecode(file)

        futures.append(inclusion_proof.remote(f"{dir_name}/{filename}", global_state))


    processed_counter = 0
    remaining = futures
    while remaining:
        done, remaining = ray.wait(remaining, num_returns=1)
        processed_counter += len(done)

        # Collect results from completed tasks
        results = ray.get(done)
        for result in results:
            if result:
                verified += 1
            else: fail += 1
            print(f"{processed_counter} von {size} geprueft. Davon {
↪ fail} failed und {verified} verifiziert")

    # Save final results
    save_results(global_state, processed_counter, size, fail, verified
↪ )

    ray.shutdown()
```

Listing A.8: Our own Python code for inclusion proving

Bibliography

[AL21] Rahul Awati and Peter Loshin. Definition certificate authority (ca), 2021. "https://www.techtarget.com/searchsecurity/definition/certificate-authority"[Online; accessed 2024-11-01].

[all] List of known and announced ct logs. "https://www.gstatic.com/ct/log_list/v3/all_logs_list.json".

[app] Apple's certificate transparency policy. "https://support.apple.com/en-ca/103214"[Online; accessed 2024-11-02].

[Art11] Charles Arthur. Rogue web certificate could have been used to attack iran dissidents, 2011. "https://web.archive.org/web/20170826175742/https://www.theguardian.com/technology/2011/aug/30/faked-web-certificate-iran-dissidents"[Online; accessed 2024-11-02].

[BH17] Jake A. Berkowsky and Thaier Hayajneh. Security issues with certificate authorities. In *2017 IEEE 8th Annual Ubiquitous Computing, Electronics and Mobile Communication Conference (UEMCON)*, pages 449–455, 2017.

[bra] TLS Policy. "https://github.com/brave/brave-browser/wiki/TLS-Policy"[Online; accessed 2024-11-02].

[BSP$^+$08] Sharon Boeyen, Stefan Santesson, Tim Polk, Russ Housley, Stephen Farrell, and David Cooper. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280, May 2008. "https://www.rfc-editor.org/info/rfc5280"[Online; accessed 2024-11-01].

[CA-12] Trustwave issued a man-in-the-middle certificate, 2012. "https://web.archive.org/web/20120313085319/http://www.h-online.com/security/news/item/

`Trustwave-issued-a-man-in-the-middle-certificate-1429982.html"`[Archived 2021-03-13; Online; accesses 2024-11-01].

[Cer24] Certificate Transparency Project. How certificate transparency works, 2024. `"https://certificate.transparency.dev/howctworks/"`[Accessed: 2024-11-01].

[Chi21] Hung-Yu Chien. Dynamic public key certificates with forward secrecy. *Electronics*, 10(16), 2021. `"https://doi.org/10.3390/electronics10162009"`[Online; accessed 2024-11-01].

[chr] Chrome certificate transparency policy. `"https://googlechrome.github.io/CertificateTransparency/ct_policy.html"`[Online; accessed 2024-11-02].

[chr22] How does the certificate transparency check in chrome work?, 2022. `"https://groups.google.com/a/chromium.org/g/ct-policy/c/FddjjCNIrLo"`[Online; accessed 2024-11-02].

[DAM$^+$15] Zakir Durumeric, David Adrian, Ariana Mirian, Michael Bailey, and J. Alex Halderman. A search engine backed by Internet-wide scanning. In *22nd ACM Conference on Computer and Communications Security*, October 2015.

[int] Intermediate Zertifikat. `"https://ssl.de/ssl-glossar/intermediate-zertifikat.html"`[Online; accessed 2024-11-01].

[Int13] Digicert announces certificate transparency support, 2013. `"https://www.darkreading.com/cyber-risk/digicert-announces-certificate-transparency-support"`[Online; accessed 2024-11-02].

[Lau14] Ben Laurie. Certificate transparency. *Communications of the ACM*, 2014. `"https://dl.acm.org/doi/fullHtml/10.1145/2659897"`[Online; accessed 2024-11-02].

[LKL13] Adam Langley, Emilia Kasper, and Ben Laurie. Certificate transparency. *Internet Engineering Task Force*, 2013.

[LLK13] Ben Laurie, Adam Langley, and Emilia Kasper. Certificate Transparency. RFC 6962, June 2013. `"https://www.rfc-editor.org/info/rfc6962"`[Online; accessed 2024-11-02].

[LPVGT$^+$19] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczynski, and Wouter Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. In *Proceedings 2019 Network and Distributed System Security Symposium*, NDSS 2019. Internet Society, 2019. `"https://arxiv.org/abs/1806.01156"`[Online; accessed 2024-11-02].

[MDO$^+$22] Sarah Meiklejohn, Joe DeBlasio, Devon O'Brien, Chris Thompson, Kevin Yeo, and Emily Stark. Sok: Sct auditing in certificate transparency, 2022. "https://arxiv.org/abs/2203.01661"[Online; accessed 2024-11-02].

[Mil11] Elinor Mills. Fraudulent google certificate points to internet attack, 2011. "https://web.archive.org/web/20111008200937/http://news.cnet.com/8301-27080_3-20098894-245/fraudulent-google-certificate-points-to-internet-attack/"[Online; accessed 2024-11-02].

[mon] The list of existing monitors. "https://certificate.transparency.dev/monitors/"[Online; accessed 2024-11-02].

[NN19] Kerry McKay (NIST) and David Cooper (NIST). Guidelines for the selection, configuration, and use of transport layer security (tls) implementations. Technical Report NIST Special Publication (SP) 800-52, Rev. 2, National Institute of Standards and Technology, Gaithersburg, MD, 2019. "https://doi.org/10.6028/NIST.SP.800-52r2"[Online; accessed 2024-11-01].

[OHR22] Chaeyeon Oh, Joonseo Ha, and Heejun Roh. A survey on tls-encrypted malware network traffic analysis applicable to security operations centers. *Applied Sciences*, 12(1), 2022.

[rat22] Yeti 2022-2 rate limits, 2022. "https://groups.google.com/a/chromium.org/g/ct-policy/c/AJ7msx2aWac/m/oz9kh8HVAgAJ"[Online: accessed 2024-11-02].

[rat23] Google ct log - getting entries, 2023. "https://groups.google.com/g/certificate-transparency/c/MOMI6kLYooM"[Online; accessed 2024-11-02].

[Res18] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018. "https://www.rfc-editor.org/info/rfc8446"[Online; accessed 2024-10-24].

[SBI13] Ijaz Ali Shoukat, Kamalrulnizam Abu Bakar, and Subariah Ibrahim. A generic hybrid encryption system (hes), 2013. "http://dx.doi.org/10.19026/rjaset.5.4793"[Online; accessed 2024-11-01].

[SMA$^+$13] Stefan Santesson, Michael Myers, Rich Ankney, Ambarish Malpani, Slava Galperin, and Dr. Carlisle Adams. X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP. RFC 6960, June 2013. "https://www.rfc-editor.org/info/rfc6960"[Online; accessed 2024-11-01].

[Sol19] Ben Solomon. Introducing certificate transparency monitoring, 2019. `"https://blog.cloudflare.com/introducing-certificate-transparency-monitoring/"`[Online; accessed 2024-11-02].

[ST20] Emily Stark and Chris Thompson. Opt-in sct auditing, 2020. `"https://docs.google.com/document/d/1G1Jy8LJgSqJ-B673GnTYIG4b7XRw2ZLtvvSlrqFcl4A/edit?tab=t.0#heading=h.4f9946en7wca"`[Online; accessed 2024-11-02].

[Tec10] Microsoft TechNet. What are ca certificates?, 2010. `"https://technet.microsoft.com/en-us/library/cc778623(v=ws.10).aspx"`[Online; accessed 2024-11-01].

[Wan20] Tingmao Wang. How certificate transparency works, exactly, 2020. `"https://blog.maowtm.org/ct/en.html"`[Online; accessed 2024-11-02].

[Wol16] Josephine Wolff. How a 2011 hack you've never heard of changed the internet's infrastructure, 2016. `"https://slate.com/technology/2016/12/how-the-2011-hack-of-diginotar-changed-the-internets-infrastructure.html"`[Online; accessed 2024-11-02].