

Certbat: Accessible Certificate Transparency Log Monitoring for Domain Owners

certb.at

Philip Stoop
Dominic Wenger

Bern University of Applied Sciences

Abstract

In the X.509 PKI system, any trusted Certificate Authority may issue a valid TLS certificate for any domain. Certificate Transparency (CT) makes such issuances observable, but domain owners still have to monitor the logs to detect wrongful or malicious certificates. This is a nontrivial and inaccessible task, especially for domain owners with limited resources. This report presents Certbat, free software intended to help make self-hosted CT log monitoring more accessible. Certbat discovers domains to monitor from common web servers, crawls the logs for matching certificates, compares them to deployed certificates and alerts if anything suspicious is found. It optionally stores log history using Bloom filters to allow configuring additional monitored domains at a later time without having to re-download the entire history of every log. Based on this, this report also proposes extending current CT log APIs to provide downloadable Bloom filters in order to reduce the amount of data monitors need to download.

Contents

1. Introduction	3
1.1. Motivation	3
1.2. Scope	4
1.3. Contributions	4
1.4. Design goals	4
1.5. Structure	4
2. Background	5
2.1. History	5
2.2. Certificate transparency	6
2.3. Monitoring	12
3. Related work	14
3.1. Cert Spotter	14
3.2. Other monitoring solutions	18
3.3. Related research	18
4. Design of Certbat	19
4.1. Overview	19
4.2. Log discovery	20
4.3. Domain discovery	21
4.4. Certificate retrieval	21
4.5. Certificate lifecycle	22
5. Implementation of Certbat	23
5.1. Code structure	23
5.2. Domain discovery	24
5.3. Crawler	24
5.4. Certificate retrieval	30
5.5. Persistence	30
5.6. Alerting	31
5.7. Historization	32
5.8. Configurability	34
5.9. Statistics and visualization	35
5.10. Distribution	35
6. Evaluation	36
6.1. Design goals	36
6.2. Feasibility of monitoring current logs	41
7. Discussion	43
7.1. Limitations of Certbat	43
7.2. Limitations of current CT log APIs	43
7.3. Proposed changes to CT log APIs	43
8. Conclusion	47
Acknowledgements	48
Bibliography	49

1. Introduction

In the X.509 Public Key Infrastructure (PKI) system, any trusted Certificate Authority (CA) may issue a valid Transport Layer Security (TLS) certificate for any website [1]. Given this, the security of the entire system is bounded by the practices of the weakest or least secure trusted CA, a *weakest link* problem [1]. Certificate Transparency (CT) is a system for monitoring the issuance of certificates wherein CAs log certificates in public ledgers referred to as CT logs [2], [3]. Logging a certificate results in a timestamp signed by the CT log.

This doesn't solve the weakest link problem but can help mitigate potential consequences of it by introducing transparency and observability. As CT logs are public, any interested party can observe them. Clients such as browsers can verify whether certificates have been logged by validating the signed timestamps using logs' public keys and may otherwise reject them. For domain owners, such verifications are however not sufficient, as they only prove that certificates have been correctly logged, not whether they were issued legitimately and correctly. For domain owners to be able to detect wrongful or malicious issuances, they must monitor all CT logs for all certificates concerning their domains. For various reasons, this is a nontrivial, inaccessible task in practice.

This report aims to improve the situation by presenting the design and implementation of *Certbat*, free software aimed at making accurate and efficient CT log monitoring more accessible and easier for domain owners.

1.1. Motivation

As alluded to, CT log monitoring requires downloading all valid entries from prevalent logs and doing so in a timely manner. Due to the high volume of logged certificates and rate limits of log operators, this can be especially hard to do for domain owners with limited resources; for example, those operating a single server and monitoring from a single IP address. A domain owner also may have the required experience and ability to issue and manage valid certificates but not the technical ability to additionally implement CT log monitoring for them. The first part of this is helped by the existence of *Certbot*, an application by the Electronic Frontier Foundation that helps make automated certificate renewal easy. Though only loosely related, the name *Certbat* is a nod to it as it tries to address the second part.

It should be noted that a range of CT log monitoring solutions already exists, including both hosted services and self-hostable¹ applications. One existing solution is *Cert Spotter* by SSLMate, which is available both as a hosted commercial service and as self-hostable free software [4]. Prior to this work, an evaluation of *Cert Spotter*² by the thesis advisor identified potential improvements that address accessibility issues and suggested the development of an application either built on or inspired by *Cert Spotter* to realize them. These suggestions include integration with web server software, improved alerting, comprehensive packaging and distribution, performance optimizations and storage space optimizations. Rather than forking *Cert Spotter*, we developed our own implementation called *Certbat*.

¹We define “self-hostable” to mean software that can feasibly be run and maintained on infrastructure under the user's own control, using resources and expertise that are reasonable for its intended scale.

²The self-hostable, free software version of it, not the hosted service.

1.2. Scope

Since Cert Spotter is the closest relevant free and open-source implementation, the evaluation in this report focuses on comparisons with Cert Spotter, despite its broader range of functionality. Further comparisons with other CT log monitoring solutions are outside the scope of this work. Section 1.3 summarizes the contributions of this report and Section 1.4 lists the design goals that define what is in scope.

1.3. Contributions

This report makes three contributions.

1. We present the design, implementation and evaluation of Certbat, a free software CT log monitor for domain owners (Section 4, Section 5 and Section 6). Certbat discovers the domains to monitor from the configuration of common web servers, crawls all prevalent CT logs while respecting their rate limits, compares matching certificates with the certificates actually deployed and alerts only if something suspicious is found. It is packaged for easy installation and ships with a sane default configuration. The design goals it was built to meet are listed in Section 1.4.
2. We assess whether current prevalent CT logs can be monitored at all from a single, resource-limited vantage point, that is, whether they grow faster than they can be scraped within their rate limits (Section 6.2).
3. Based on Certbat’s use of Bloom filters to store log history, we propose extending the CT log APIs to serve downloadable Bloom filters, which would reduce the amount of data every monitor has to download (Section 7.3).

1.4. Design goals

Certbat was designed to meet the following goals. Section 6 revisits each of them.

- G1** Be *easily* installable and configurable and provide sane default configuration.
- G2** Support *automatic discovery* and manual configuration of monitored domains.
- G3** Retrieve and process entries from CT logs *efficiently*.
- G4** Store downloaded entries efficiently and avoid unnecessary re-downloads.
- G5** Respect and adhere to the rate limits of CT logs.
- G6** Identify certificates corresponding to the configured domains.
- G7** Retrieve deployed certificates to compare them with the identified certificates.
- G8** Alert timely and reasonably, *avoid alerting unnecessarily*.
- G9** Run with reasonable performance assuming deployment alongside a web server.
- G10** Be free and open-source, licensed under the GNU AGPL.

1.5. Structure

Section 2 provides historical and technical background information. The intention of it is not to be comprehensive and complete but rather to provide enough context to understand the subsequent sections. Section 3 covers Cert Spotter and related work. Section 4 covers the design of Certbat, Section 5 its implementation and Section 6 its evaluation and the comparisons to Cert Spotter. Section 7 discusses results, limitations and future work, including the proposed changes to the CT log APIs. Section 8 concludes the report.

2. Background

2.1. History

In 2011, there were several catastrophic security incidents involving CAs, acting as a catalyst for addressing problems with the PKI system. A breach of a registration authority in Italy affiliated with the company Comodo occurred in March of that year³. The party responsible for the attack was able to mis-issue certificates for domains such as *login.live.com*, *mail.google.com*, *login.skype.com* and others [7], [8], [9]. This breach was detected within hours and the certificates subsequently revoked [8]; however, the incident broke the chain of trust of existing PKI, leaving users vulnerable during that period. In August of that year, an intrusion at Dutch CA DigiNotar came to public attention through a forum post submitted by an Iranian user titled “*Is this MITM Attack to Gmail’s SSL?*” [10], [11]. This incident was worse, as at least 513 fraudulent certificates were mis-issued and actively abused [12], [13]. As part of it, a man-in-the-middle (MITM) attack using a mis-issued wildcard certificate for *google.com* was conducted against approximately 300’000 users of Gmail located in Iran [12], [13]. The breach stayed undetected by DigiNotar for more than one month [13].

The CA and PKI system had been subject to criticisms before these incidents. Ellison and Schneier [14] explored and published a list of risks in 2000. Guttman [15, p. 21] pointed out in 2002 that “Any CA can usurp a certificate issued by any other CA” and “Overall security is that of the least trustworthy CA”. The DigiNotar incident had demonstrated this weakness in practice. Around the time of these incidents, Google had begun addressing problems by starting to develop and offer early mechanisms to monitor certificate issuance. It was the certificate pinning features of Google’s Chrome browser that led to a warning being shown to the user who authored the DigiNotar forum post [13]. In a Google Security Blog post from 2011, Laurie [16] demonstrates how to query the Google Certificate Catalog, a (since defunct) database of crawled certificates published in DNS. For a certificate to qualify for the catalog, it required a valid signature as well as a correct domain name, matching the domain where it was crawled from [16]. This approach had limitations in practice, as it only captured certificates seen by Google’s crawlers and was also overly reliant on Google as an additional party to be trusted.

In 2012, Laurie *et al.* [17] published the first draft for Certificate Transparency, describing a system around a public, verifiable, append-only log of certificates with the aim of detecting mis-issuance. It subsequently evolved into RFC 6962 which is further explained in Section 2.2. By 2013, Google had been running two logs in production and had added support for it in Chrome [12]. In 2015, Chrome required all extended validation certificates to be logged and starting from issuances after April 2018, all new certificates are required to be logged or are rejected as untrusted [18]. At the time of writing (2026), there are more than 45 logs run by 8 different providers and major browsers⁴ support CT [20].

Holz provides a more detailed discussion of the motivating incidents, general problems of the PKI system, and Certificate Transparency itself [1].

³A DEF CON 19 talk presented by Marlinspike [5], [6] provides historical context and documents how Comodo’s reaction to the incident was met with critical reception at the time.

⁴Major browsers refers to Google Chrome, Apple Safari, Mozilla Firefox and Brave according to [19].

2.2. Certificate transparency

Certificate Transparency describes an ecosystem, in which certificates can be logged to public, verifiable, append-only ledgers referred to as Certificate Transparency logs [3]. Binary Merkle hash trees are used to enable efficient inclusion proofs and provide the append-only property through consistency proofs [17]. CAs log certificates at the time of issuance such that anyone can observe what certificates have been issued for any domain and such that clients like browsers can reject any non-logged certificates [17].

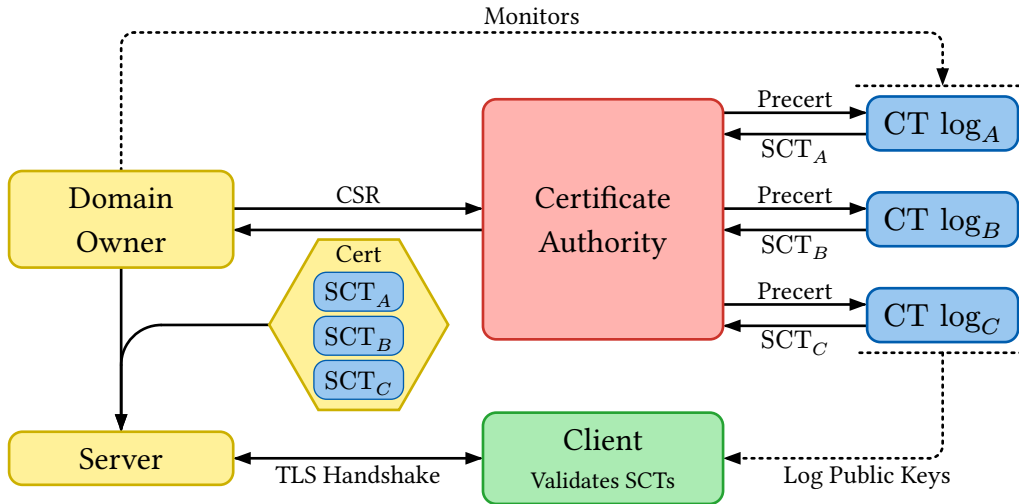


Figure 1: Example and overview of Certificate Transparency.

When a CA logs a certificate, the CT log returns a Signed Certificate Timestamp (SCT) in response. An SCT consists of a version, a log id identifying which CT log produced it (the SHA-256 hash of the log’s public key), a timestamp, a (presently unused) extensions array and the signature (shown in Figure 2) [17]. The signature covers the mentioned fields as well as the submitted certificate and is created using the CT logs private key [17]. The SCT acts as a signed promise that the CT log creates a log entry based on the submitted certificate within a Maximum Merge Delay (MMD) [17]. In addition to signing SCTs, the CT log also periodically adds new entries to its Merkle tree and signs the Merkle root [17].

When a client tries to establish a TLS session with a domain owner’s server, it obtains SCTs as part of the TLS handshake [17]. The SCTs are transmitted either through a X.509 v3 extension (this requires a precertificate, explained in Section 2.2.1), through a TLS extension or using Online Certificate Status Protocol (OCSP) stapling [17]. The client thus receives SCTs alongside or within server certificates and can then validate them in order to assert that the certificates have been submitted to a log [17]. For this it needs to know the public keys of the corresponding CT logs as well as the current time in order to reject SCTs with a timestamp in the future [17]. Note that this is assuming that the client trusts the logs (see Section 2.2.3). Different clients can have different CT policies and obtain CT logs public keys in different ways, something which is not mandated by RFC 6962 [17]. For instance, the Mozilla Firefox browser has a list of CT logs including public keys compiled into its binary executable and requires, depending on lifetime, at minimum 2 or 3 valid SCTs issued from distinct logs in this list in order for a certificate to be accepted [21].

Figure 1 shows an example and overview of CT. A domain owner requests a certificate from a CA, which logs it in three logs *A*, *B* and *C*. The logs create the SCTs and return them. The CA embeds the obtained SCT_A , SCT_B and SCT_C in the final certificate and returns it to the domain owner. The domain owner deploys it on their server. The client connecting to the server can verify that the CA logged the certificate, by using the public keys of the logs to validate the SCTs embedded within the certificate. In this figure, it is also implied that the domain owner monitors the logs. Note that the domain owner is just the chosen example, as anyone can monitor the logs. Monitoring and auditing are covered in Section 2.3.

2.2.1. Precertificates

Because the signature of an SCT covers the submitted certificate, trying to embed the SCT into it using a X.509 v3 extension introduces a cryptographic circularity. The CT log has to include the certificate in the signature to create the SCT but the certificate cannot be finalized until it contains the SCT. To address this, RFC 6962 [17] introduces the concept of a precertificate. A precertificate is created from the certificate to be issued by adding a special poison extension which prevents it from being used in practice [17]. The CA signs this precertificate before submitting it to the CT log, including the poison extension in that signature [17]. The CT log does not include the poison extension in the signature it creates when creating the SCT [17]. Instead, the log signs over the certificate body without the poison extension, along with a hash of the issuing CA's public key, binding the SCT to this specific issuer [17]. After receiving an SCT for a precertificate, the CA creates a new certificate without the poison extension, embeds that SCT and then signs it [17].

2.2.2. Log lists

Something that RFC 6962 [17] does not define is which specific CT logs a CA should submit a certificate to. Both CAs and monitors need to know which logs exist, the CAs in order to choose logs to submit to and the monitors to be able to monitor comprehensively. In practice, a CA could make the choice of where to log based on the intersection of logs trusted by most clients. Apple and Google offer curated log lists for their clients, which contain log information such as endpoint, public key and state. At the time of writing, these are the two primary lists of logs and other clients derive from them. Apple and Google have defined policies which log operators must adhere to in order to have their logs added to the lists. For instance, Apple requires an MMD of at most 24 hours, both Apple and Google require uptimes of 99% or above [22], [23]. Logs included in their lists are assigned a state of *pending*, *qualified*, *usable*, *readonly*, *retired* or *rejected* based on lifecycle and adherence to the policies [23], [24]. Google's policy explicitly prohibits the use of Chrome's CT log list for CT enforcement in other TLS clients without written permission [22]; however, the list is made available for submitters, monitors and auditors [22].

2.2.3. Logs

Conceptually, a log can be thought of as an ordered list of entries, with entries consisting of a timestamp and a Distinguished Encoding Rules (DER) encoded certificate or precertificate. In the case of a precertificate, entries additionally contain the hashes of the issuing CAs public keys. However, a log is really a store of entries and additional data (certificate chains), a merkle tree made up of hashes and an Application Protocol Interface (API) to interact with it.

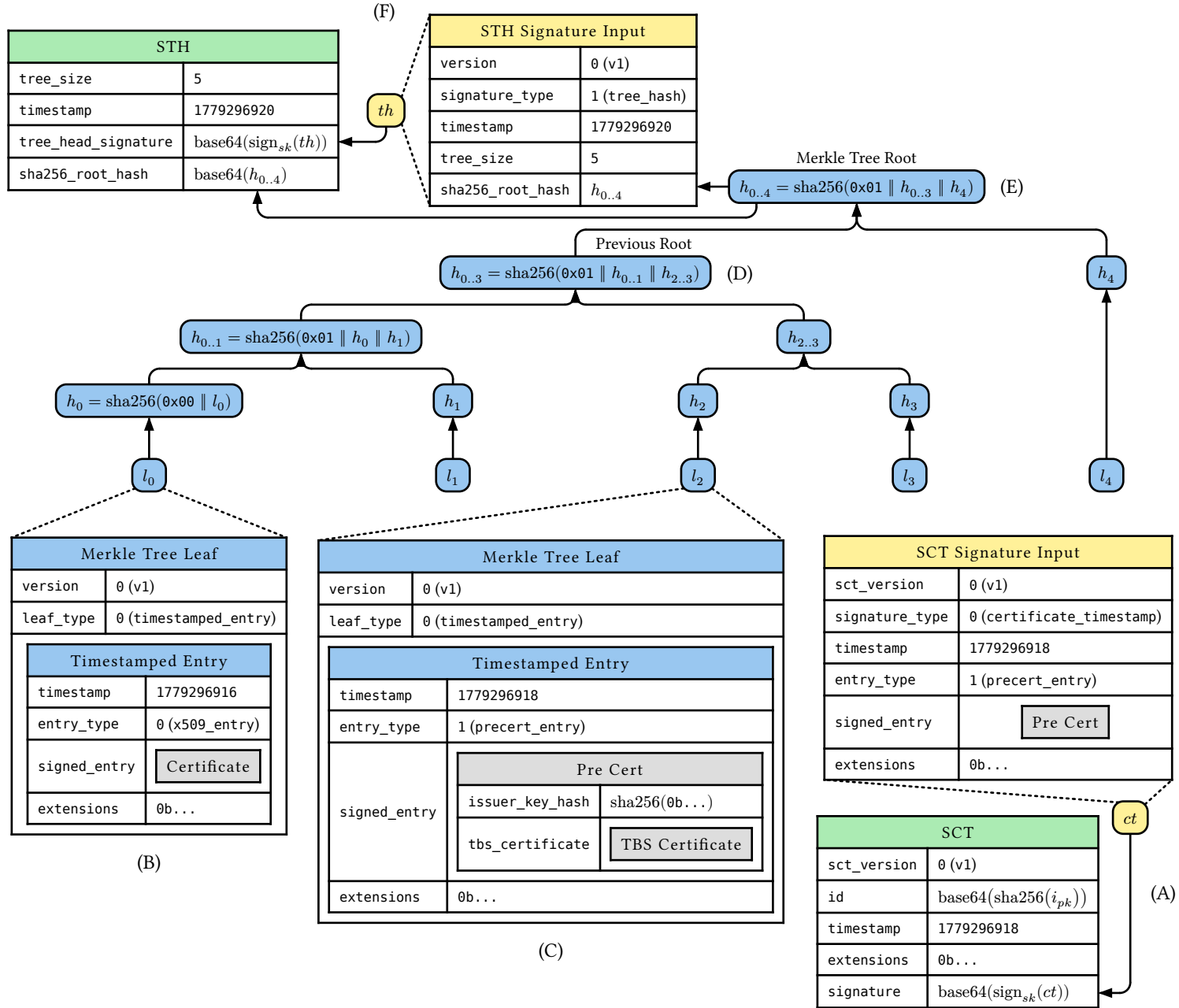


Figure 2: Merkle tree and core data structures of RFC 6962.

Figure 2 shows both the core data structures of RFC 6962 [17] as well as how the Merkle tree is formed. Combining these into the same figure should help better understand which parts are hashed and signed.

Part (A) shows an SCT and what the signature covers. This SCT is for the precertificate of the entry shown in part (C). Note that (sk, pk) is a key pair consisting of the secret key and public key of the log. Also, i_{pk} is the DER encoded Subject Public Key Info (SPKI). The text $0b...$ is used to denote arbitrary bytes. Note that though the field is present, there are currently no extensions. Parts (B) and (C) are the two possible entries, the hashes of them are the leaves of the tree. The entries for the other leaf hashes are omitted in the figure. Part (B) is for a regular X.509 certificate and part (C) for a precertificate. In the Merkle tree, leaf hash inputs are prepended with bytes $0x00$, other hash inputs with bytes $0x01$. Part (D) is the previous root, part (E) the current root after the most recent entry l_4 , hashed as h_4 , was submitted. Hashes are also referred to as Merkle Tree Hashes (MTH) [17].

The root hash, size of the Merkle tree and a timestamp are signed with the logs' private key. This is called the Signed Tree Head (STH). Part (F) shows how the hash is part of the inputs of the signature and how the signature is part of the final STH that one can obtain from a log using its API. The previous instance of the tree with size 4, without h_4 and with part (D) as root produced a different STH.

As previously mentioned, there are two kinds of Merkle proofs that can be done, inclusion proofs and consistency proofs. An inclusion proof proves that some entry was logged. For example, to prove inclusion of the precertificate leaf l_2 of part (C) in the current tree with root hash $h_{0..4}$, the log returns the set of hashes $\{h_3, h_{0..1}, h_4\}$ (Notably, it doesn't have to return all entries). To verify, one hashes the leaf and recomputes up to the root hash using this set and the size taken from the STH. The computed root hash must be equal to the root hash from the STH, shown in Equation 1.

$$\begin{aligned}
h'_2 &= \text{sha256}(0x00 \parallel l_2) \\
h'_{2..3} &= \text{sha256}(0x01 \parallel h'_2 \parallel h_3) \\
h'_{0..3} &= \text{sha256}(0x01 \parallel h_{0..1} \parallel h'_{2..3}) \\
h'_{0..4} &= \text{sha256}(0x01 \parallel h'_{0..3} \parallel h_4) \\
h'_{0..4} &= h_{0..4}
\end{aligned} \tag{1}$$

A consistency proof proves that the tree at some point in time contains all entries of an earlier tree and in the same order, thus proving the append-only property. Part (D) is the root of the previous tree of size 4, while part (E) is the root of the current tree of size 5. In this example, the old tree is exactly the left subtree of the new root. The consistency proof from size 4 to size 5 is therefore the hash h_4 . To verify, one needs the previous STH and the newer STH. Starting from the previous STH's root hash $h_{0..3}$, combining it with h_4 and recomputing $h_{0..4}$. The computed hash must be equal to the root hash from the newer STH, shown in Equation 2.

$$\begin{aligned}
h'_{0..4} &= \text{sha256}(0x01 \parallel h_{0..3} \parallel h_4) \\
h'_{0..4} &= h_{0..4}
\end{aligned} \tag{2}$$

For arbitrary tree sizes, the proof contains the intermediate hashes needed to recompute root hashes for both tree sizes from a shared set of nodes. To verify, one can check the signatures of the STHs and compare the recomputed roots to the root hashes contained in them. When observing a log, this check can be repeatedly carried out between the observed STHs, verifying that the observed sequence stays append-only.

A remaining issue is that a log could present different, conflicting views of the Merkle tree to different parties. To address this, RFC 6962 [17] proposes something called *gossiping*. The idea is that everyone who is auditing is *globally gossiping* by exchanging and comparing their retrieved STHs in order to notice any conflicts [17]. However, RFC 6962 [17] doesn't describe how this should work, declaring that the exact mechanism for this would be described in a separate document and listing it under *Future Changes* [17]. At the time of writing, no standardized and widely deployed gossip protocol for CT appears to exist.

2.2.4. Version 1 (RFC 6962)

RFC 6962 [17] defines the API a CT log should provide. The API consists of HTTP endpoints accepting GET and POST requests for submitting certificate chains, retrieving entries, retrieving proof data and retrieving metadata. All responses and parameters for POST requests are encoded as JavaScript Object Notation (JSON) [17]. Binary data is base64 encoded [17]. Parameters for GET requests are passed as query parameters. Endpoints follow the form:

`https://<log server>/ct/v1/<endpoint>`

Table 1 lists the endpoints, summarizing the purpose, input and output of each. For specifics, refer directly to Section 4 of RFC 6962 [17].

PURPOSE	KIND	ENDPOINT	INPUT	OUTPUT
Log certificate chain	POST	add-chain	Certificates	SCT
Log precertificate chain	POST	add-pre-chain	Precertificates	SCT
Retrieve latest STH	GET	get-sth		STH
Retrieve consistency proof between two STHs	GET	get-sth-consistency	First and second tree sizes	Consistency proof (list of nodes)
Retrieve inclusion proof by leaf hash	GET	get-proof-by-hash	Leaf hash and tree size	Leaf index and audit path (list of nodes)
Retrieve entries by indices	GET	get-entries	Start and end indices	Entries (list of leaf nodes and chains)
Retrieve accepted root certificates	GET	get-roots		List of root certificates
Retrieve entry and inclusion proof by index (for debug purposes)	GET	get-entry-and-proof	Leaf index and tree size	Entry and audit path (list of nodes)

Table 1: RFC 6962 API endpoints.

For monitoring, only `get-sth` and `get-entries` are needed. A monitor can periodically poll the `get-sth` endpoint to detect when new entries have been added to the log and use the `get-entries` endpoint to retrieve the entries [17].

2.2.5. Version 2 (RFC 9162)

In 2021, RFC 9162 [25] was published to define Certificate Transparency Version 2.0 and obsolete RFC 6962 [17]. It adds enhancements such as hashing and signature algorithm agility, a new precertificate format, OID-identified logs and more [25]. We are not aware of any adoption or implementation of RFC 9162 [25] in the wild at the time of writing [26]. Given this, we will not further explain it here. However, an alternative to the RFC 6962 [17] API has emerged independently of RFC 9162 [25]. Referred to as the Static Certificate Transparency API or Static CT API, it is recognized by both Google’s and Apple’s log programs [22], [23].

2.2.6. Static CT API

The Static CT API was originally specified along Sunlight, a CT log implementation sponsored by Let’s Encrypt offering partial compatibility with the API defined in RFC 6962 [17], [27], [28]. It provides the same submission endpoints, SCTs and STHs as the RFC 6962 [17] API but provides different read endpoints which serve easily cacheable and compressible binary tiles as static assets [29]. The chosen architecture requires no database or dynamic endpoints, allowing tiles to be stored, for example, on strongly consistent object storage such as S3 [29]. It also allows for tighter requirements regarding MMD. For example, Apples policies allow regular logs to specify an MMD of up to 24 hours but shortens this requirement to only up to 1 minute for Static CT API logs [23].

A CT log implementing the Static CT API contains endpoints under two URL prefixes, the submission prefix for write APIs and the monitoring prefix for read APIs [27]. The submission prefix exposes RFC 6962-compatible endpoints for submitting certificates, while the monitoring prefix exposes the log as static assets. Table 2 lists the endpoints of the Static CT API. For specifics, refer directly to the Static CT API documentation [27].

PURPOSE	PREFIX	ENDPOINT	CONTENT
Submit certificate chain	Submission	/ct/v1/add-chain	Certificates and SCT
Submit precertificate chain	Submission	/ct/v1/add-pre-chain	Precertificates and SCT
Retrieve accepted root certificates	Submission	/ct/v1/get-roots	Root certificates
Retrieve latest STH	Monitoring	/checkpoint	Checkpoint with tree size, root hash and signatures
Retrieve Merkle hash tiles	Monitoring	/tile/<L>/<N>[.p/<W>]	Merkle tree hashes
Retrieve entry data tiles	Monitoring	/tile/data/<N>[.p/<W>]	Logged entries
Retrieve issuer certificates	Monitoring	/issuer/<fingerprint>	Issuer certificate by hash

Table 2: Static CT API endpoints.

The /tile endpoint serves *tiles*, which are sequences of consecutive hashes where <L> is the level in the tree and <N> is the index within that level [27]. The /tile/data endpoint serves *data tiles* containing entries matching the tiles at level 0 with <N> being the index. Each full data tile contains 256 entries [27]. Issuer certificates referenced by data tiles are served separately by fingerprint [27]. The .p/<W> suffix is for partial tiles, with <W> being the width.

It is worth describing how specifying a tile index with <N> works, as one cannot just provide a single decimal integer but has to split it into multiple path elements. A non-negative decimal integer is split into 3-digit path elements, with all but the last path element prefixed by x. For example, index 1000 is encoded as x001/000, index 1234067 is encoded as x001/x234/067 [27].

A good reason to serve hashes as tiles using the /tile endpoint instead of arbitrary inclusion and consistency proofs is that this shifts work from the log servers to the clients. It lets the clients choose the exact tiles needed for the proofs and fetch them in parallel, instead of the servers having to calculate the set of required nodes when proofs are requested.

2.2.6.1. Names tiles

There exists an optional *names tiles* extension for the Static CT API. It originates from the Sunlight CT log implementation and is *not* part of the Static CT API specification [27], [30]. It adds another endpoint:

```
https://<monitoring prefix>/tile/names/<N>[.p/<W>]
```

It is meant for clients that are only interested in a feed of certificate names [30]. Data tiles as well as the `get-entries` endpoint of the RFC 6962 API return full certificates, which results in monitors potentially downloading much more data than needed, something which is further discussed in Section 7.2. The /tile/names endpoint accepts the same parameters as the /tile/data endpoint but returns newline delimited JSON objects containing only the names corresponding to the certificates of the data tiles. Names are taken from the Common Name field and Subject Alternative Name extension of certificates.

The extension was announced and discussed on the Google CT policy mailing list, where it was met with mixed reception [31]. At the time of writing, we are aware of three other Static CT API implementations in addition to Sunlight (TesseraCT⁵, Azul⁶ and Itko⁵). By analyzing their source code, we found that these three implementations do *not* implement the names tiles extension. We additionally probed all tiled logs of Google's current log list for support⁶. We found that only *Let's Encrypt*-operated logs currently serve names tiles, whereas all other logs, operated by *Google*, *Cloudflare*, *TrustAsia* and *IPng Networks*, do not.

2.3. Monitoring

According to Google-maintained CT ecosystem documentation [3], monitors make sure all logged certificates are visible in the log, watch for suspicious certificates and are run by companies and organizations, offered as subscription services for domain owners and certificate authorities, or run by individuals [3]. Because the term *monitor* is not always used with an explicit scope, it can be unclear what exactly is meant by it. Section 5.3 of RFC 6962 describes monitors as watching logs both to check that they behave correctly and to detect certificates of interest, with a monitor at least needing to inspect every new entry in each watched log [17]. In the steps following this description, this includes verifying STH signatures, checking that fetched entries correspond to an STH and verifying consistency proofs [17]. Section 5.4 additionally introduces *auditors*, which verify partial evidence about a log without needing

⁵Itko seems to be discontinued, as the project has been archived on GitHub.

⁶Probed on June 10th, 2026, by requesting the first and a recent names tile from each log.

to download the whole log [17]. For example, an auditor can verify consistency between two STHs or verify that a certificate accompanied by an SCT was included after the MMD [17].

In this work, we distinguish between three related activities. *SCT validation* checks that a log signed a valid promise to include a certificate⁷. *Log verification* checks cryptographic properties of the log, such as STH signatures, inclusion proofs, consistency proofs and, in the broader ecosystem, detection of split views through gossiping. *Domain monitoring* means inspecting logged certificates for domain names of interest in order to detect unexpected issuance. Unless stated otherwise, we use *monitoring* in this third sense going forward.

Consider an undesirable situation, in which there exists a mis-issued certificate by a compromised, misbehaving or careless CA. This certificate has either not been logged and cannot provide any SCTs or it has been logged and can thus provide one or more SCTs. In the first case, because of CT, connecting user-agents will simply reject it and not establish a connection. In the second case, an attacker can theoretically abuse the certificate without detection for at most the duration of the MMD [17]. With an MMD of 24 hours, this means a mis-issued certificate could be used for an attack such as MITM interception before it becomes visible to monitors. For Static CT API logs this is reduced to 1 minute, as mentioned in Section 2.2.6. Furthermore, with Google’s policies requiring log availability of 99%, measured over a 90-day rolling average, a monitor’s ability to observe new entries may be further delayed by up to ≈ 21 hours [22].

After this window, however, *someone* still has to be actively monitoring the logs for certificates of affected domains in order to catch the mis-issued certificate. RFC 6962 [17] alludes to this, mentioning that CT logs do not detect mis-issued certificates and that interested parties such as domain owners may want to monitor the logs to do so [17].

For a domain owner wanting to do this independently, monitoring not being clearly scoped to domain monitoring possibly affects the feasibility and accessibility of it. Potentially as a result of how it is described in RFC 6962 [17], available self-hostable monitoring software, including Cert Spotter, couples domain monitoring with broader log verification work. This results in more implementation and resource costs for domain owners than necessary, assuming that other parties already correctly do the log verification work. Certbat is explicitly scoped to do domain monitoring and do so in a way that hopefully makes it more accessible for domain owners.

⁷Note that a valid SCT itself does not prove that a certificate was logged. To prove actual inclusion, an inclusion proof against an STH dated after SCT timestamp + MMD is needed.

3. Related work

3.1. Cert Spotter

Cert Spotter is a CT log monitoring solution which is available both as a hosted commercial service and as self-hostable free software⁸ [4]. Cert Spotter was created by Andrew Ayer and is marketed as part of SSLMate, a product of Opsmate, Inc. [34]. As a free and open-source solution, Cert Spotter was evaluated by the thesis advisor prior to this work. This resulted in ideas for potential improvements which are reflected in the design goals in Section 1.4. Since Certbat is evaluated against Cert Spotter in Section 6, Cert Spotter is discussed here in sufficient detail for that comparison.

Cert Spotter not only does domain monitoring but also checks log consistency. In the README file [4], it is described as not just a log monitor but also a log auditor. Though, aligned with the distinction we made in Section 2.3, we can say that it does log verification work. As further mentioned there, such functionality is out of scope for Certbat.

3.1.1. Operation

Cert Spotter is implemented in the Go programming language and can be installed using `go install` [4]. Alternatively, packages are available for Debian, for Ubuntu, for Arch Linux through the Arch User Repository (AUR) and for FreeBSD through the FreeBSD Ports Collection [35], [36], [37].

By default, the directory `~/.certspotter` is used to store configuration and state, though this can be overridden using environment variables [4]. For example, the Debian package sets the configuration directory to `/etc/certbat`. There, one can create file `watchlist` containing the list of domains to be monitored, file `email_recipients` containing email addresses alerts are sent to and directory `hooks.d`, containing executable scripts [4]. When a certificate for a configured domain is found, email alerts are sent to the email addresses using `sendmail` and the hook scripts are executed [4]. The `certspotter` binary should be run as a daemon [4]. For example, the Debian package installs Systemd unit and service files for this.

During startup, Cert Spotter fetches a list of logs to monitor, by default from the URL `https://loglist.certspotter.org/monitor.json`; the `-logs` option can be used to override the URL. It periodically re-fetches this list. By default, it scrapes the logs from start to end, first going through the backlog and only then monitoring newly logged certificates in real-time. Depending on log sizes, it may take a significant amount of time until the tips of the logs are reached. The option `-start_at_end` makes it start at the tips of logs and not download entries from the past.

Cert Spotter doesn't require a database. It saves state inside the state directory using four directories `certs`, `logs`, `errors` and `healthchecks`. For example, matching certificates are saved to the `certs` directory in PEM format. Note that these directories are not organized to be easily human readable; for example, SHA-256 hashes are used as file names.

⁸The code is licensed under the Mozilla Public License 2.0 (MPL 2.0), a license which, according to the Free Software Foundation (FSF), is compatible with the GNU General Public License (GPL) [32], [33].

3.1.2. Architecture

This section provides an analysis of how Cert Spotter works based on the latest version of the source code at the time of writing. Said version is 0.24.1 at the commit with short hash 68a01db, tagged v0.24.1 [38]. Since Cert Spotter is implemented in Go, *goroutines* refers to Go's lightweight user-space threads used for concurrent execution.

As mentioned, the list of logs is periodically re-fetched. After each refresh, a pseudo-random interval between 30 and 90 minutes is selected to determine when the next refresh happens. For each log, Cert Spotter starts several goroutines working through a *monitoring pipeline*. There are $3 + 2n$ goroutines per log, which we have assigned labels here for clarity: one *fetcher*, one *batcher*, n *downloaders*, n *processors*, and one *saver*.

One architectural choice is that the log list not only contains log information and metadata but also parameters controlling how logs are scraped. Variable n from the previous paragraph, which is effectively the download and processing concurrency per log, is also influenced by these parameters. Listing 1 shows a truncated excerpt of a log list containing the parameters `certspotter_download_qps`, `certspotter_download_jobs` and `certspotter_download_size`.

```
"logs": [  
  /* ... */  
  {  
    "key": /* ... */,  
    "mmd": 86400,  
    "url": "https://wyvern.ct.digicert.com/2027h1/",  
    "log_id": "ABpdGhwtk3W2SFV4+C9xoa5u7zl9KXyK4xV7yt7hoB4=",  
    "description": "DigiCert Wyvern 2027h1",  
    "temporal_interval": {  
      "end_exclusive": "2027-07-01T00:00:00+00:00",  
      "start_inclusive": "2027-01-01T00:00:00+00:00"  
    },  
    "certspotter_download_qps": null,  
    "certspotter_download_jobs": 2,  
    "certspotter_download_size": 256  
  },  
  /* ... */  
]
```

Listing 1: Excerpt of the fetched log list JSON provided and used by Cert Spotter.

If `certspotter_download_qps` is set, $n := 10$. Otherwise, if `certspotter_download_jobs` is set, n is set to its value. If neither field is set, $n := 1$. In the log list retrieved at the time of writing, `certspotter_download_qps` is set for only four logs. The field `certspotter_download_jobs` is set for all logs, varying only between 1 and 4.

The fetcher periodically retrieves the log's latest STH every five minutes, verifies its signature using the log's public key, and communicates it to the batcher. The batcher defines contiguous ranges of log indices called batches based on saved STHs and the newly observed tree size and communicates these to the downloaders. The downloaders fetch the batches, then the

processors parse the entries in them and match them against the configured watch list. Lastly, the saver advances the persisted state.

The batcher compares the known STH tree sizes with the current persisted download position. If the current position is p and a known STH has tree size s , then the monitor must still download the entries in the range $[p, s)$, which we refer to as *the backlog*. The backlog is split into bounded batches. Starting from the current position, the batcher repeatedly computes the next batch. For RFC 6962 logs, the default batch size is 1000 entries, overridable through `certspotter_download_size`. For static logs, the batch size is 256 entries, corresponding to one tile. If one or more known STHs fall before the next batch boundary, the batch ends at the largest such STH size. Otherwise, the batch ends at the next regular batch boundary. Cert Spotter preserves these STH boundaries because it locally computes the Merkle root hashes and compares them against the root hashes from the fetched STHs.

The downloaders fetch the entries for batches. Because RFC 6962 logs are allowed to return fewer entries than requested, the remaining part of a batch is repeatedly fetched until the full batch has been downloaded. The processors parse every downloaded certificate or precertificate and compare the extracted domains against the watch list. Cert Spotter uses its own certificate parsing implementation instead of relying only on Go's standard X.509 parser. The README file [4] justifies this in the name of increasing robustness: certificates shouldn't be skipped due to being malformed or encoded unexpectedly. The parser keeps most certificate structures as raw ASN.1 and just extracts identifiers. It also handles identifiers containing null bytes, IP addresses and other special cases. Although downloading and processing are parallelized, the state advances sequentially. Processed batches can complete out of order, so Cert Spotter places them into a sequencer. The saver consumes batches from this sequencer, computes the local Merkle tree and advances the persisted position only after all earlier batches have also been processed. This prevents gaps in the local state, but means that a slow earlier batch can temporarily block later completed work. State is persisted per log. The main state file for a log, `state.json`, contains the download position, verified position, latest verified STH, and last-success timestamp. Fetched STHs that have not yet been verified are stored per log, as files in a directory. Once such an STH is verified, it is removed from the unverified STHs and recorded as the latest verified STH in `state.json`.

Error handling is retry-based. Requests to CT logs are wrapped in an exponential backoff loop starting at 100ms and capped at 30s. Fetching STHs and downloading entries are retried until Cert Spotter is stopped. HTTP responses with status codes other than 200, malformed responses, invalid STH signatures, and failed downloads are treated as errors and retried. Rate limiting is also per log. If `certspotter_download_qps` is set for a log, Cert Spotter uses it as request-rate limit (QPS meaning Queries Per Second). If it is not set, no request-rate limit is applied and concurrency is controlled only by `certspotter_download_jobs`.

Overall, we think it is fair to say that its architecture is fundamentally designed around the log verification work rather than treating this as an optional layer. A design focusing on domain monitoring could potentially achieve better performance and lower resource use by default, something we hope to achieve with Certbat.

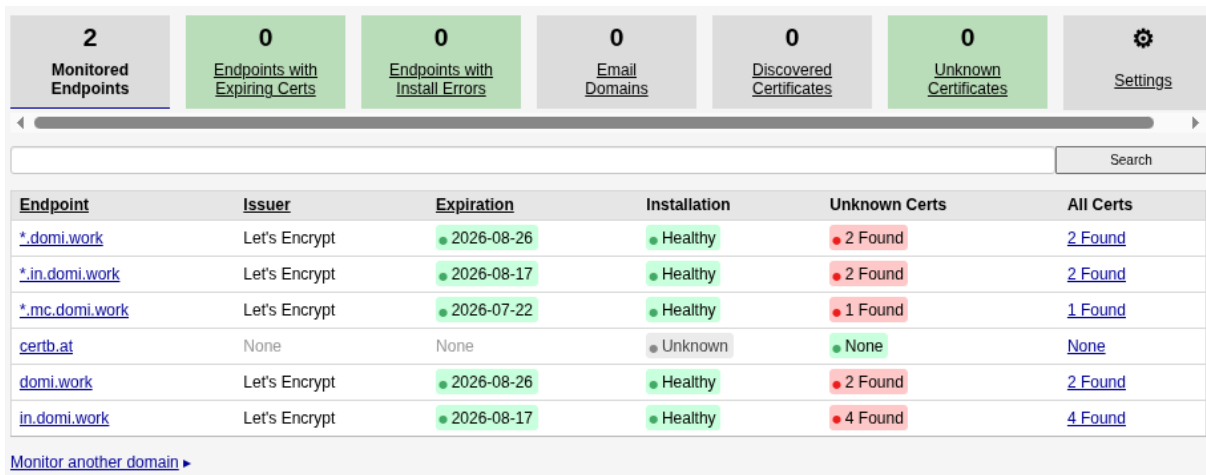
3.1.3. Hosted service

The hosted version of Cert Spotter provides more features than the self-hostable version and is also worth mentioning. It is a commercial product available as different plans. The plans and their feature sets are shown in Table 3.

FEATURE	HOBBYIST	STARTUP	BUSINESS
Monthly price	\$15	\$100	\$500
Monitored endpoints	Up to 20	Up to 150	Up to 1000
Certificate health checks	Hourly	Every 15 minutes	Every 5 minutes
Ports and IPs per endpoint	443, 1 IPv4, 1 IPv6	Unlimited	Unlimited
DNS/registrar integration	No	Yes	Yes
Webhook notifications	No	Yes	Yes
Monitoring locations	Default	Default	10 worldwide locations
Customer support	Standard	Fast	Priority
Multiple email recipients	No	Yes	Yes
Audit log retention	30 days	90 days	Unlimited

Table 3: Hosted Cert Spotter plans.

The hosted version allows excluding subdomains, authorizing CAs and public keys, using CAA records to determine if certificates are authorized, discovering certificates to monitor using DNS records and alerting using email, slack, webhooks and browser push notifications. It fetches deployed certificates periodically.



The screenshot shows the Cert Spotter Web Dashboard. At the top, there are seven status boxes: '2 Monitored Endpoints', '0 Endpoints with Expiring Certs', '0 Endpoints with Install Errors', '0 Email Domains', '0 Discovered Certificates', '0 Unknown Certificates', and a 'Settings' gear icon. Below these is a search bar. The main table has columns: Endpoint, Issuer, Expiration, Installation, Unknown Certs, and All Certs. The table lists several endpoints, mostly from 'Let's Encrypt', with their expiration dates, installation status (Healthy or Unknown), and the number of certificates found. A link 'Monitor another domain' is at the bottom left.

Endpoint	Issuer	Expiration	Installation	Unknown Certs	All Certs
*.domi.work	Let's Encrypt	2026-08-26	Healthy	2 Found	2 Found
*.in.domi.work	Let's Encrypt	2026-08-17	Healthy	2 Found	2 Found
*.mc.domi.work	Let's Encrypt	2026-07-22	Healthy	1 Found	1 Found
certb.at	None	None	Unknown	None	None
domi.work	Let's Encrypt	2026-08-26	Healthy	2 Found	2 Found
in.domi.work	Let's Encrypt	2026-08-17	Healthy	4 Found	4 Found

Figure 3: Cert Spotter Web Dashboard

The service provides a simple Web Dashboard at <https://sslmate.com/console/monitoring/>, shown in Figure 3.

It is unclear how and how much code of the self-hostable version of Cert Spotter is shared with the hosted version. Note that with the self-hostable version, some of the added features can be replicated by using the *hook scripts* feature [39].

3.2. Other monitoring solutions

We could not find a complete list of CT log monitoring solutions online. The Google-maintained CT documentation [40] lists some, none of which are self-hostable or free software. Table 4 lists the options we found which do domain monitoring, including Certbat and both versions of Cert Spotter. Besides Certbat and Cert Spotter, Silent CT is the only other self-hostable free software option. Its author describes it as a prototype and not yet stable [41]. It also currently doesn’t support the Static CT API.

MONITOR	DEPLOYMENT	LICENSE
Certbat	Self-hostable	AGPLv3
Silent CT [◦]	Self-hostable	BSD 2-Clause
Cert Spotter [◦] (open-core)	Self-hostable	MPL 2.0
Cert Spotter [◦] (hosted)	Hosted	Proprietary
CertKit [◦]	Hosted	Proprietary
Cloudflare CT Monitor [◦]	Hosted	Proprietary
DigiCert CT Log Monitor [◦]	Hosted	Proprietary
Keytos EZMonitor [◦]	Hosted	Proprietary
Hardenize [◦]	Hosted	Proprietary
Red Sift Certificates [◦]	Hosted	Proprietary
Stellastra [◦]	Hosted	Proprietary

Table 4: CT monitoring solutions.

The table is not necessarily complete. The hosted services all require account creation, some providing free subscription plans offering limited functionality. Their feature sets mostly overlap with that of the hosted version of Cert Spotter.

3.3. Related research

We did not find any research focused on the implementation of CT log monitoring software itself. The most relevant work we found evaluates the reliability of existing monitors in the wild. Li *et al.* presented a study of CT monitors and found that ordinary domain owners face impractical data volumes, while queried third-party monitors did not guarantee complete certificate sets for domains of interest [42], [43]. This supports our claim that monitoring is not sufficiently accessible for domain owners.

There is relevant ongoing work, referred to online as *Verifiable indexes* [44]. It tries to address one of the core accessibility issues, that of a domain monitor having to download large amounts of data. Briefly summarized, the idea is to allow targeted lookups into large transparency logs with cryptographic evidence that the returned results are complete and untampered with [44]. There is currently no paper or draft standard, only an introductory talk which was given at the *Transparency.dev Summit 2025* [45].

Addressing the same issue, Dahlberg and Pulls proposed Verifiable Lightweight Monitoring (LWM), a CT extension that allows domain owners to verify the completeness of certificate notifications without downloading all log entries [46]. The focus of this research is on newly added certificates rather than an entire log. One of the authors, Rasmus Dahlberg, is also one of the creators of *Silent CT*, which is listed in Section 3.2 [41].

4. Design of Certbat

4.1. Overview

For Certbat, the main idea is that it is domain monitoring software that can easily be installed on a single server, quickly be configured and then largely be forgotten, requiring little to no ongoing maintenance. It is aimed primarily at administrators of web server software and meant to run next to such software, as it can interact with it to automatically discover the domains to monitor for (this is optional, one can also configure domains manually). Based on the discovered and configured domains, Certbat periodically obtains the deployed certificates by performing TLS handshakes for each domain. During the scraping of the logs, the idea is to have a balance between freshness and completeness, by fetching newly added entries as they are logged and working through the backlogs to find certificates that were logged before monitoring started. Potentially running next to web server software or on modest hardware, CPU and RAM utilization cannot be excessive, something that Certbat is designed to optimize for. Found certificates are compared to those obtained using TLS handshakes and mismatches result in alerts being sent.

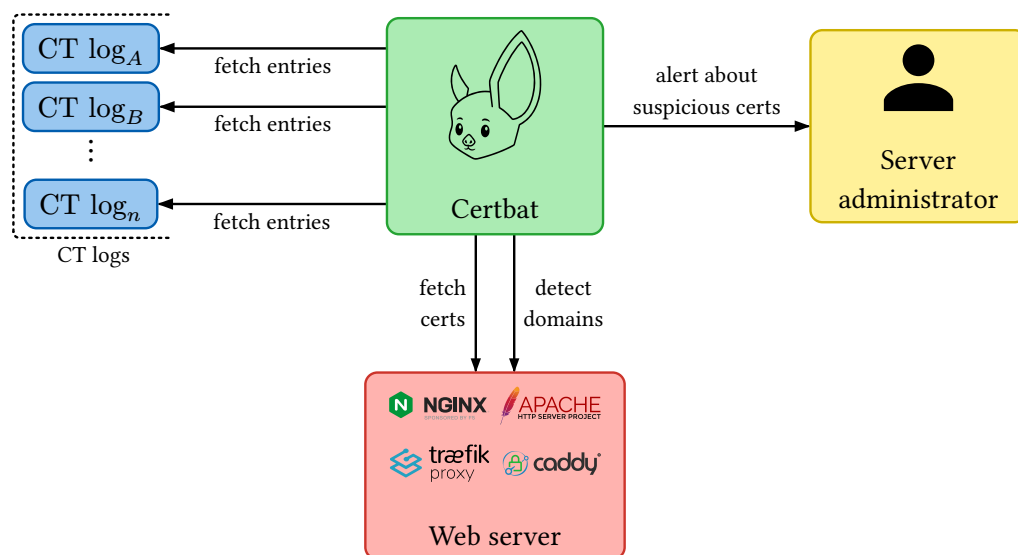


Figure 4: Overview of Certbat

Since the backlogs are processed, historical data can optionally be saved in order to avoid re-downloading all of it in case new domains to monitor are added some time after Certbat is started. Certbat doesn't need to store full certificates for this. Instead, to optimize storage space, it only stores Bloom filters. It does so per log and per range of log entries, derived from the domain names extracted from certificates.

A domain monitor's behaviour should take into consideration that certificates expire. Certbat does so wherever it can, by skipping logs that only contain expired entries, ignoring expired certificates during processing, removing expired state on disk and managing historical data as short- and long-lived Bloom filters. As a result, it also doesn't alert for expired certificates if it encounters those while working through the backlogs.

Figure 4 provides a simplified overview of Certbat's main responsibilities and interactions.

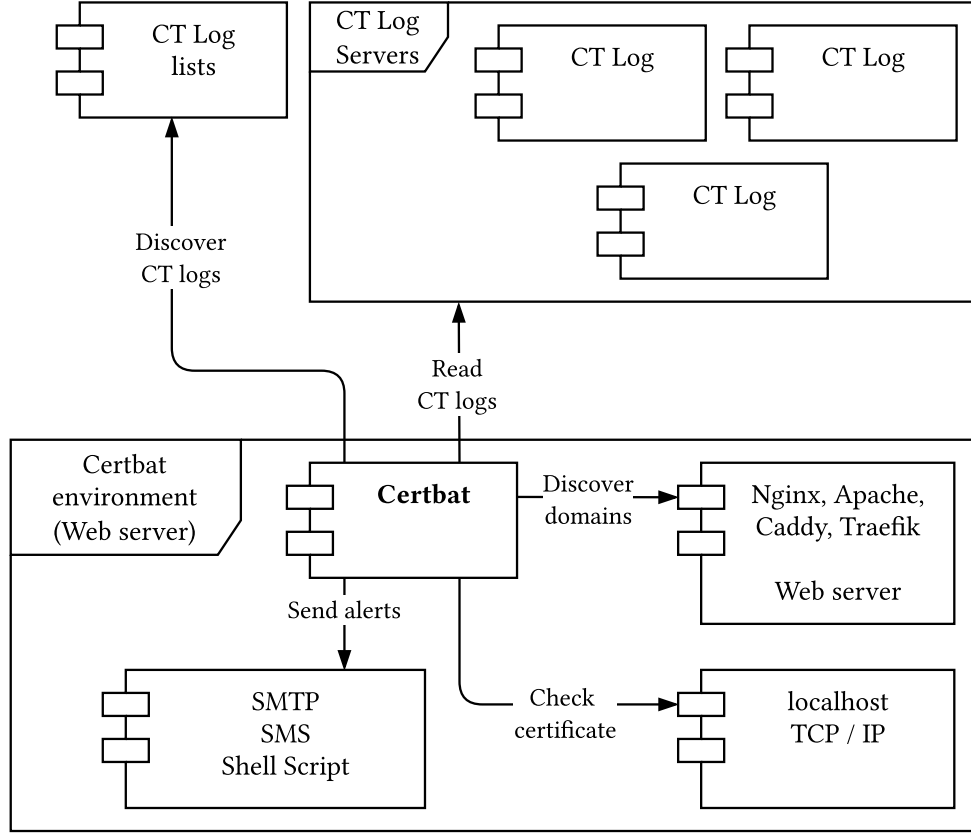


Figure 5: Big picture overview

Figure 5 further shows the *big picture* and how Certbat may be installed on a server next to web server software. To discover active domains from local web server software, it inspects their configuration. The idea there is to support a few common web servers but also provide a generic interface by letting users specify a shell script that returns domains.

4.2. Log discovery

As alluded to in Section 2.2.2, it is important to be able to determine a good (and complete) set of CT logs. Certbat periodically fetches the Apple and Google CT log lists, keeps only qualified, usable, or readonly logs whose temporal interval has not ended, merges them with manually configured additional logs by URL, and finally removes all logs whose URL matches a configured ignore pattern. Figure 6 shows how this results in L_M , the final set of logs to be monitored.

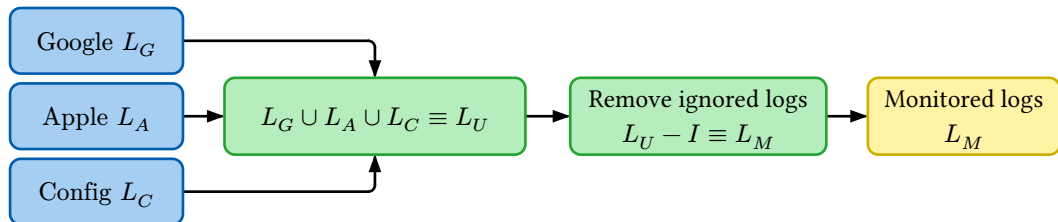


Figure 6: Construction of Certbat's monitored CT log set

The reason for choosing these two provider's log lists is that we couldn't find any other lists that weren't also already derived from them.

4.3. Domain discovery

Certbat treats the monitored domains as the union of multiple named *domain sources*, so domains can be configured from multiple running web servers, which we refer to as *providers*. Manually configured domains make up the *static* domain source and can coexist with automatically discovered domains. Each automatic source owns its own set of domains. Updates from one provider therefore do not accidentally affect the domains from another provider or from manual configuration. For example, if a domain gets removed from one web server and configured for another, the end result is that this domain is still monitored. The effective monitored domain set is $D_M = \cup_{s \in S} D_s$, where S is the set of configured sources and D_s is the current domain set of source s .

The supported providers are *Nginx*, *Apache*, *Caddy*, *Traefik* and *Custom*. The four web servers are chosen due to being two popular traditional web servers and two popular modern reverse proxies. Current design tradeoffs are that Certbat only considers domains that are expected to be served through TLS on port 443 and ignores routing constructs that cannot be resolved to concrete domains in a straightforward way. Due to that, *some* complex but valid configurations are missed. We have plans to change that and describe a solution approach for it in Section 7. How the domains are retrieved from the individual providers is implementation specific and described in Section 5.

4.4. Certificate retrieval

With *certificate retrieval* we refer to retrieving deployed certificates for concrete monitored domains using TLS, not retrieving certificates from the CT logs. For a given concrete monitored domain, Certbat periodically connects to domain:443 using TCP, sets domain as the Server Name Indication (SNI), initiates a TLS handshake, and retrieves the leaf certificate. Since Certbat connects to domain:443, not necessarily to localhost, certificate retrieval may make a round trip. As such, Certbat gets the certificates external clients would see. Certificate verification is skipped during this step as it is not Certbat's responsibility. A tradeoff of the current design is that port 443 is hard-coded. We deem this acceptable *for now*, as it covers the common HTTPS deployment case and is caused by the design tradeoffs of domain discovery.

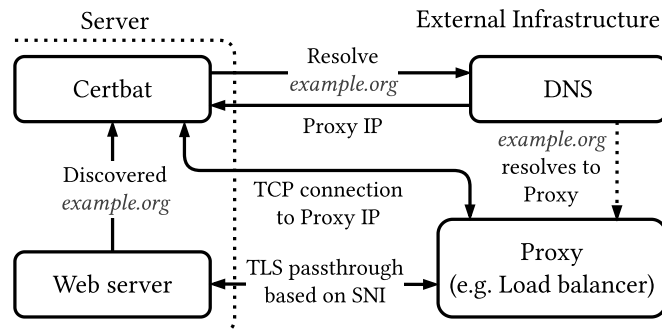


Figure 7: Example of roundtrip during certificate retrieval.

Figure 7 shows an example where the discovered domain resolves to an external proxy, which forwards the TLS connection to the local web server based on SNI. Certbat therefore follows the same path as an external client to retrieve the certificate. Importantly, this works only under the assumption that this path is not subject to a MITM attack!

4.5. Certificate lifecycle

Each certificate that is fetched from the CT logs goes through different states. Before even writing any code, we modeled this *lifecycle* as a state machine, shown in Figure 8. Note that if a previously unknown domain is later added for monitoring, matching passive certificates are potentially re-downloaded, which results in them entering the **New** state.

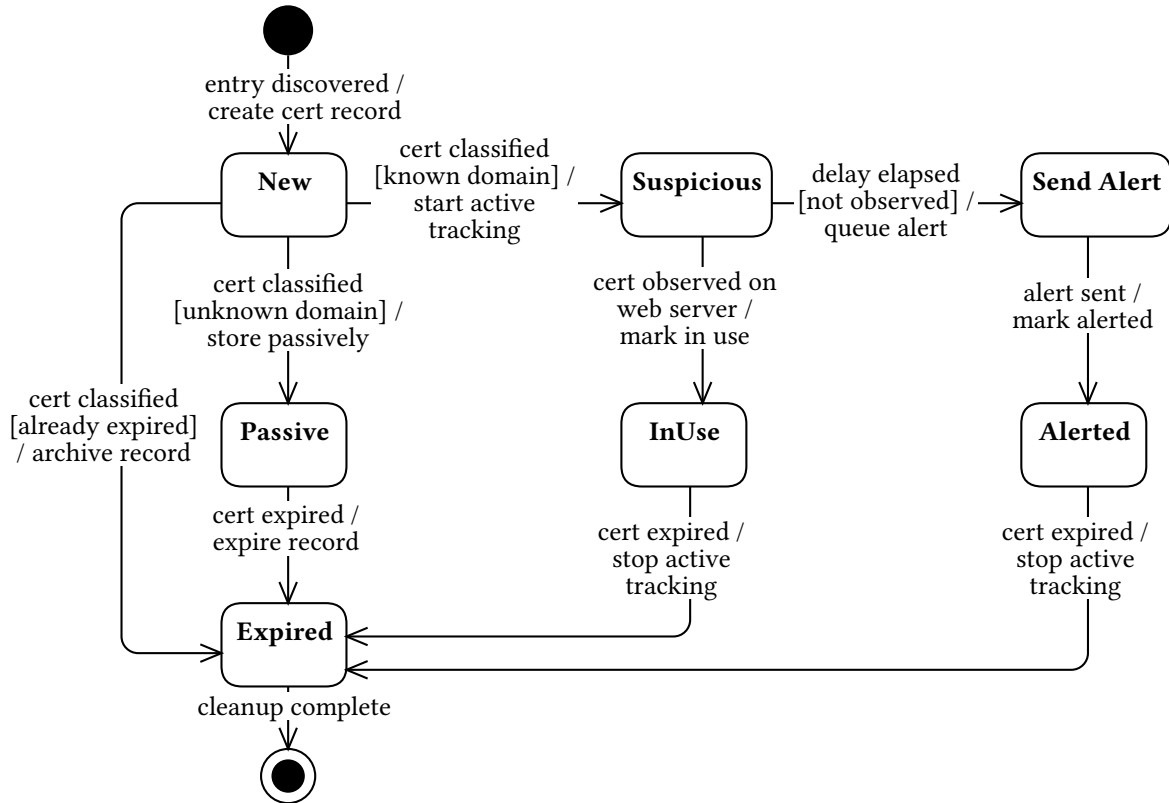


Figure 8: Certificate lifecycle

A newly discovered certificate is first classified by validity period and by the domains it contains. It is a conscious design decision that expired certificates should never result in alerts, as they cannot be used for new TLS connections anyway.

If an unexpired certificate does not match any monitored domain, it is stored passively (see Section 5.7). If a certificate matches a monitored domain, it becomes actively tracked, starting out as *suspicious*. This means that Certbat has seen the certificate in a CT log but has not yet observed it as deployed. A suspicious certificate can become *in use* if Certbat observes it as deployed. If the certificate is not observed as deployed after a configurable delay (by default 10 minutes), an alert is queued and the certificate marked as *alerted*. The delay is a tradeoff: alerting immediately would reduce detection time, but it would also create false positives during normal issuance and deployment, where a certificate may appear in CT logs before being installed on a server.

Though in Figure 8 we use the simplified *alert sent*, alerting is actually split into two parts. When Certbat decides that a certificate should cause an alert, it doesn't send one directly but adds it to a list. Delivery from the list of alerts is handled separately and alerts are only removed once sending succeeds. The list is persisted, thus pending alerts survive restarts of the software. How alerting is implemented is described in Section 5.6.

5. Implementation of Certbat

5.1. Code structure

Certbat is written in the Go programming language and compiled and deployed as a single statically linked binary. Its internal package organization can be seen in Figure 9. The package names follow Go naming conventions.

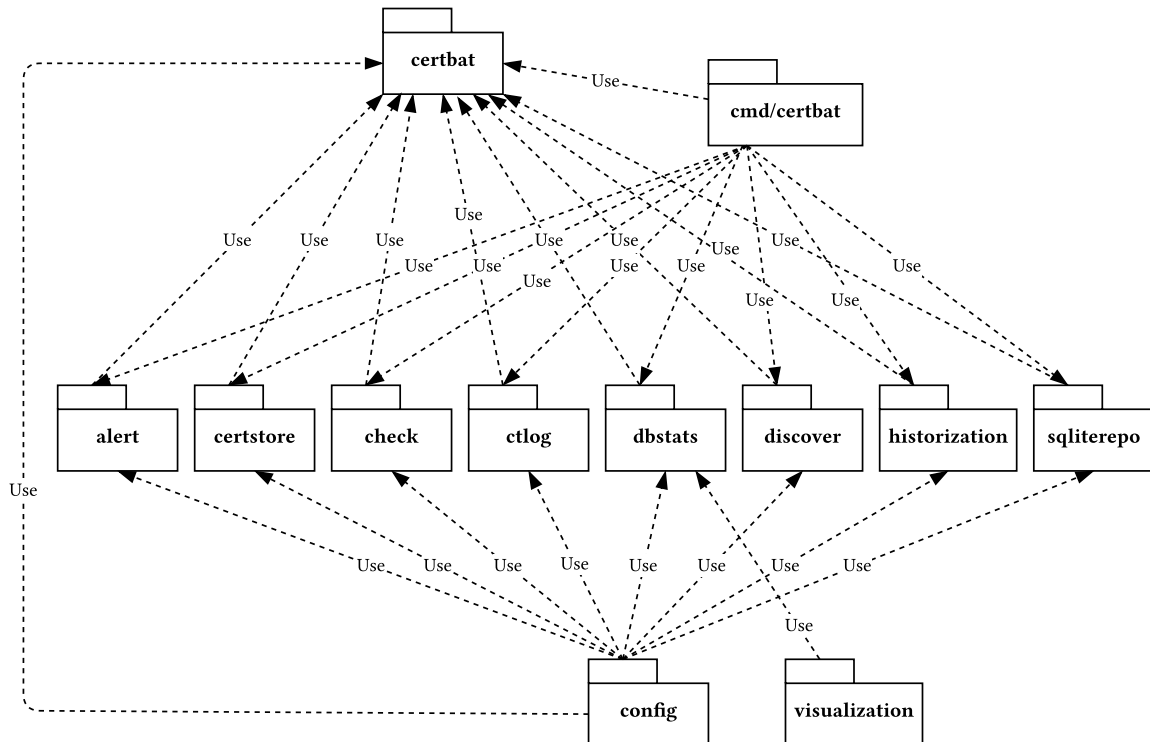


Figure 9: Internal package structure

The system is built around the `certbat` package, which defines core interfaces between subsystems. In addition, the `certbat` package contains the core business logic, written against these interfaces. The optimized parsing logic is also located in this package. The intended dependency direction is that subsystem packages may import `certbat`, while `certbat` itself does not import subsystem packages. Configuration-related logic lives inside the `config` package. It knows the interfaces of `certbat` and the subsystems and contains the logic for parsing the TOML configuration file. The main package in `cmd/certbat` contains the main function, which is the entry point of the program. It uses the `config` package to load the configuration, initializes the subsystems, and then starts the `certbat` runner. The subsystems are passed to `certbat` through interfaces.

5.1.1. Subsystems

The `alert` package contains logic to send alerts. The `certstore` package is responsible for organizing and storing suspicious certificates. The `check` package contains logic for retrieving deployed certificates using TLS. The `ctlog` package contains the crawler which discovers and crawls CT logs. The `dbstats` and `visualization` packages extract and store statistics from Certbat and then render an HTML page visualizing them. The `discover` package contains logic for domain discovery. Its subpackages implement the specific domain discovery providers (see Section 4.3). Lastly, the `sqliterepo` package contains SQLite database interaction logic.

5.2. Domain discovery

The design of the domain discovery is described in Section 4.3. Implementation-wise, we have the `DomainProvider` interface which is implemented by each provider. During startup, Certbat stores the manually configured domains as the static source and starts one goroutine for each configured provider.

The provider implementations live in subpackages of the `discover` package. The Nginx provider parses the Nginx configuration using the `nginx-go-crossplane` library and extracts `server_name` values from all TLS-enabled server blocks. The Apache provider uses the output of `apachectl -S` and only keeps names from virtual hosts on port 443. Caddy and Traefik are queried through their APIs; For Caddy, its full configuration is retrieved as JSON, while for Traefik, HTTP and TCP router rules are such as `Host(...)` and `HostSNI(...)` are parsed. The custom provider executes a configured command and reads one domain per line from its output. Across the providers, domains are converted to ASCII using IDNA handling and duplicates are removed. Entries that cannot be interpreted as concrete domain names, such as regular-expression based host rules, are currently skipped.

5.3. Crawler

The purpose of the crawler is to download certificates from CT logs reasonably fast while not unreasonably affecting CPU and RAM use, i.e. not exhausting the servers resources. Downloading from the CT logs is challenging for multiple reasons: First, there are many active logs (~60 in total at the time of writing). Secondly, the logs see a relatively high growth rate with roughly 300 certificates logged per second per log [47]. Thirdly, inefficient APIs provided by the logs that only provide downloads of full certificates result in large amounts of data to download. This further negatively affects CPU and RAM use, as the full certificates need to be parsed to extract sought after information. Lastly, rate limiting and varying availability of the log servers must be taken into consideration.

5.3.1. Rate limiting

The rate limiting implemented by CT logs is done using standard HTTP 429 status codes. Chosen rate limits are tolerable if one is only interested in *tailing the logs*, downloading new certificates as they are logged. However, as mentioned in Section 4.1, Certbat aims for both freshness and completeness, thus not only needs to download new certificates but also all existing valid certificates in a log, the backlog. On a fresh install, there are thus many certificates to download and check, resulting in logs often responding with rate limit responses. These responses in many cases do not contain useful information, such as accurate `Retry-After` headers to specify how to handle them.

To reasonably deal with this for Certbat, we make the assumption that log operators in general do constant request per second rate limiting. For each log, we keep a variable which specifies the *delay between requests* (we call this *ird*, inter-request-delay). If, for a given interval, there was no rate limiting response, this delay is lowered linearly. If there are many rate limiting responses, the delay is doubled. The inspiration for this implementation is the Additive Increase/Multiplicative Decrease (AIMD) algorithm used by TCP [48].

With this strategy, we are able to keep rate limiting responses down to one per minute over all CT logs we are interacting with.

5.3.2. Resource consumption and prioritization

The bandwidth consumed by Certbat is dictated by the size and number of certificates in the CT logs. With the current prevalent protocol and API as described by RFC 6962 [17], there is not much improvement possible. What can be done is to try to make good decisions on what to download first, *prioritization*. How Certbat prioritizes is affected by the following assumptions: Larger CT logs are especially relevant, as those are where most certificates are logged. Users of the software want parameters such as *number of workers* to be configurable. Reading the CT logs at multiple positions is beneficial for downloading efficiency. The crawler implementation is based on these.

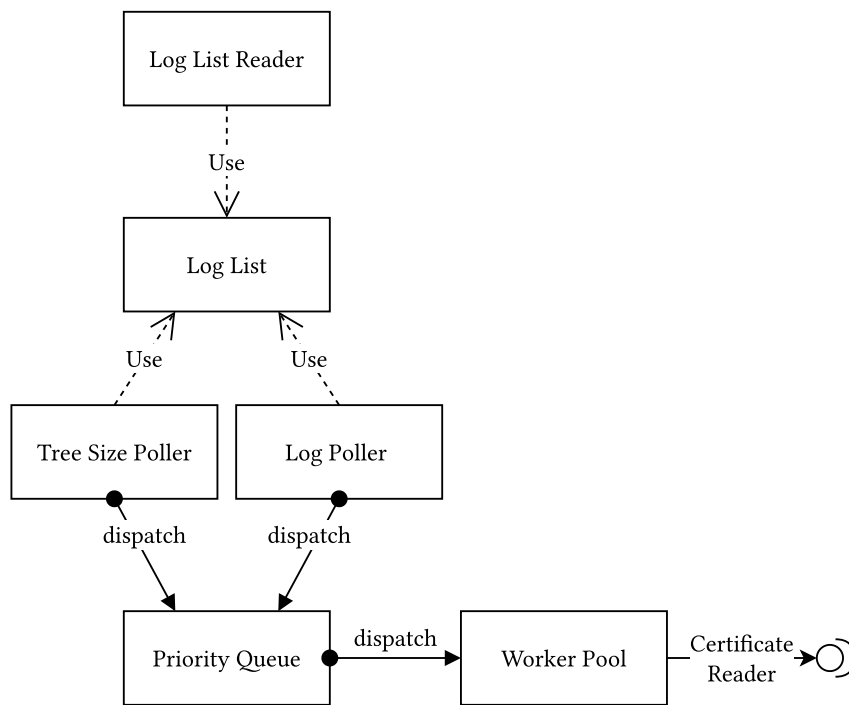


Figure 10: Crawler overview

Figure 10 shows an overview of the implementation. The *Log List Reader* periodically fetches the lists of CT logs. It populates the *Log List*, a shared data structure where all logs, including information about rate limiting and read progress, are stored. This *Log List* is then used by the *Tree Size Poller* and the *Log Poller*. The *Tree Size Poller* periodically adds a request to fetch the tree size to the *Priority Queue*. The *Log Poller* adds read requests to the *Priority Queue* and keeps up to 32 pending read requests per log. The *Priority Queue* orders the requests by highest effective priority. At first, we only used the absolute entry index at which a read request starts as its priority. It turned out that using only this start index resulted in starvation of smaller logs, such that new entries published there could be missed.

To fix this, we tried different priority formulas and arrived at the following. Let t be the current tree size, p the first log entry index covered by the read request, and a the number of seconds the request has already waited in the *Priority Queue*:

$$\left(0.25 + 0.75 * \frac{p}{t}\right) * \log(t) + a \quad (3)$$

When p is close to t , the request covers a range near the current end of the log; when p is smaller, it covers an older range further back. This favors recently added entries over older ranges. Multiplication by $\log(t)$ gives larger logs higher priority but only sublinearly, to avoid the starvation of smaller logs we saw with absolute indices. The constant 0.25 keeps older ranges relevant even when $\frac{p}{t}$ is small. The age term a makes requests gain one priority point per second spent in the queue. Tree size requests are not described by this formula, they always get maximum priority.

The requests in the *Priority Queue* are then dispatched to the *Worker Pool* (in our measurements we used a pool of size 48). The *Worker Pool* is a set containing a configurable number of goroutines that process requests from the *Priority Queue*. The requests are then processed by the workers. In case of a tree size request, the *Log List* is updated directly by the worker. In case of a read request, the received certificates are written to a channel, from which they can be read using the *LogReader* interface.

Note that the per-log rate limiting delay is applied before a request reaches the *Priority Queue*. For each log, Certbat stores the earliest time at which the next request should be sent. When a request is created, this time is advanced by the current *ird*. If the resulting request is still too early, it is only added to the *Priority Queue* after the remaining delay. Also, workers support both RFC 6962 and Static CT API tile requests, but both are internally represented as the same kind of read request.

Compared to our initial implementation which used multiple per-log goroutines without coordination, this implementation reduced RAM consumption from 5 GB on average to around 500 MB on average. With historization disabled, RAM consumption drops even lower, to around 150 MB on average.

5.3.2.1. Certificate parsing

Though these prioritization optimizations in the crawler improved RAM consumption, CPU consumption was inadequate in this version of the implementation. On our test machine with 96 CPU cores and 256 GB of RAM, Certbat used up to 60% ($60\% * 96 = 5760\%$) of CPU time. We however realized that most of the CPU time was spent while parsing certificates. As we are only interested in the *NotAfter* field and the domain names, we switched to an optimized parsing routine for X.509 v3 certificates containing the *SubjectAltName* extension. When this optimized parsing routine fails, Certbat falls back to the full parsing routine.

For comparison, we created a benchmark comparing the certificate parsing of Cert Spotter with our fast and slow implementations. For this, the following setup was used: We extracted around 0.5 M certificates using our crawler and built a Go benchmark to compare how long it takes to parse this data set. The benchmark compares the library we used, Google's *certificate-transparency-go* (*ct-go* in the following figures), the Go standard library (*Stdlib*), our

fast implementation (*Fast*), and Cert Spotter's implementation (*Cert Spotter*). The benchmark was run 5 times on two different machines. On an x86-64 Intel Core i7-8750H 8 core CPU with 4 performance cores and on an arm64 Apple M2 8 core CPU with 4 performance cores and 4 efficiency cores from a 2023 MacBook Air. In each run, the benchmark parsed all 0.5 M certificates.

Here are the results:

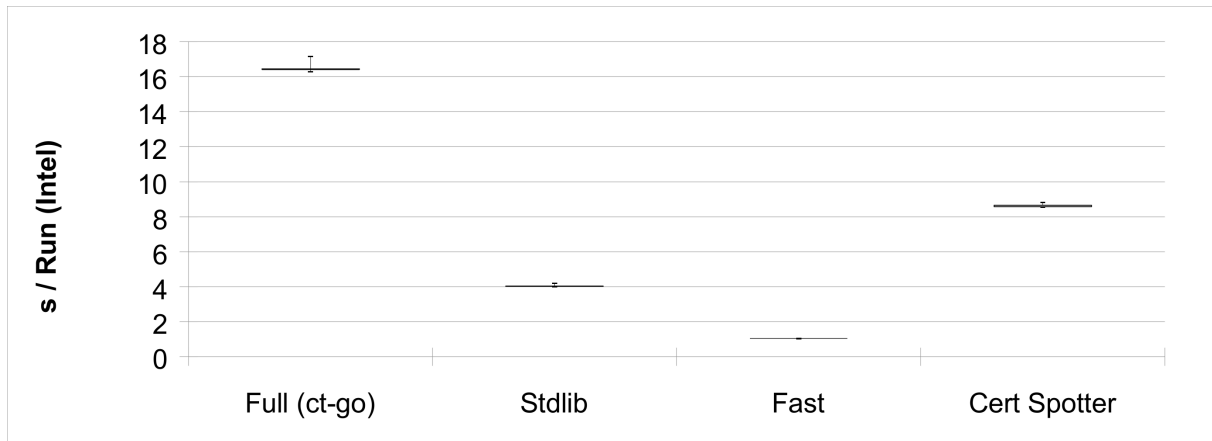


Figure 11: Boxplot of the runs on Intel

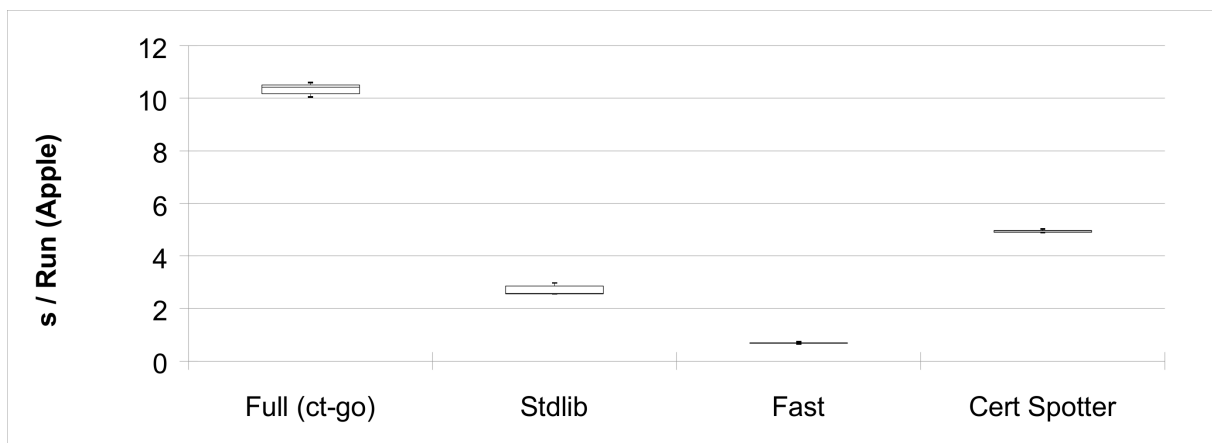


Figure 12: Boxplot of the runs on Apple

On the following diagram we set the median time used by our fast implementation to 1. This allows us to compare the relative performance of our implementation to the other ones.

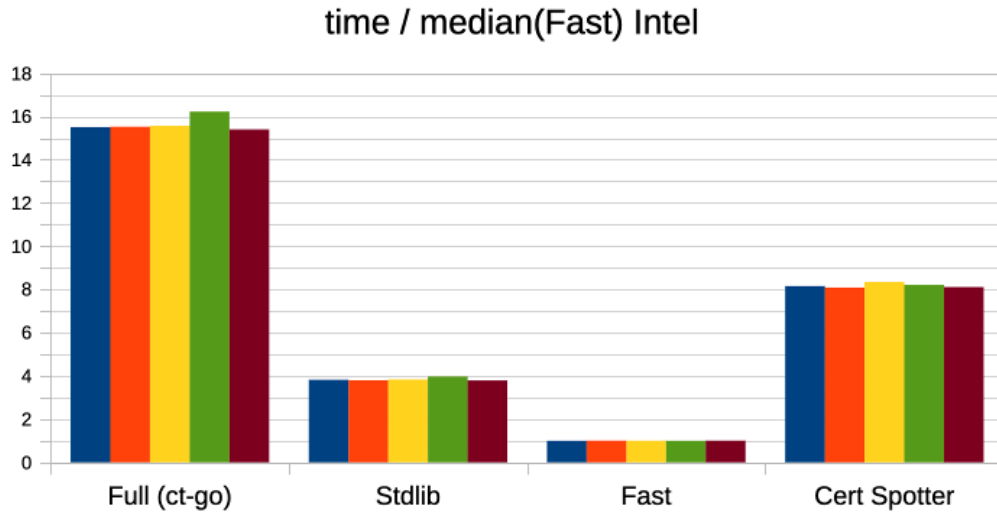


Figure 13: Benchmark runs on Intel

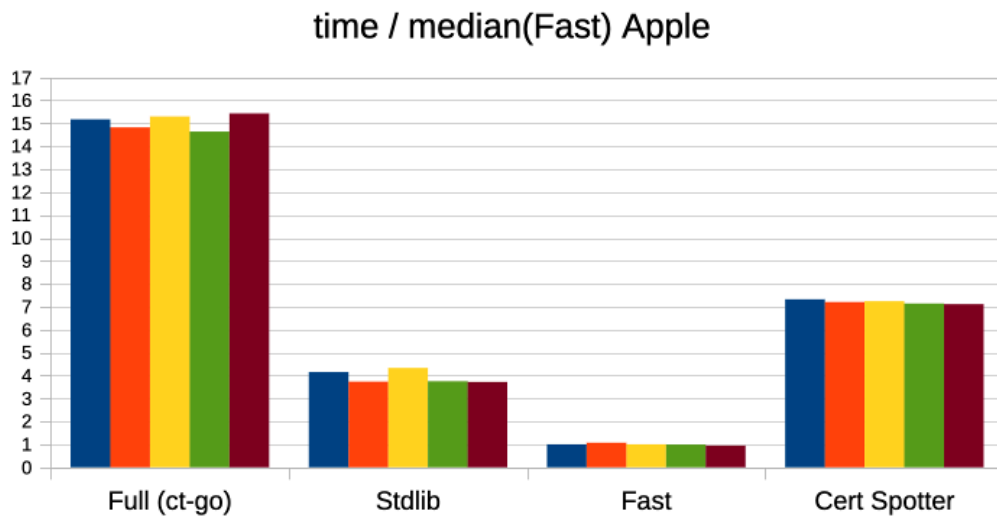


Figure 14: Benchmark runs on Apple M2

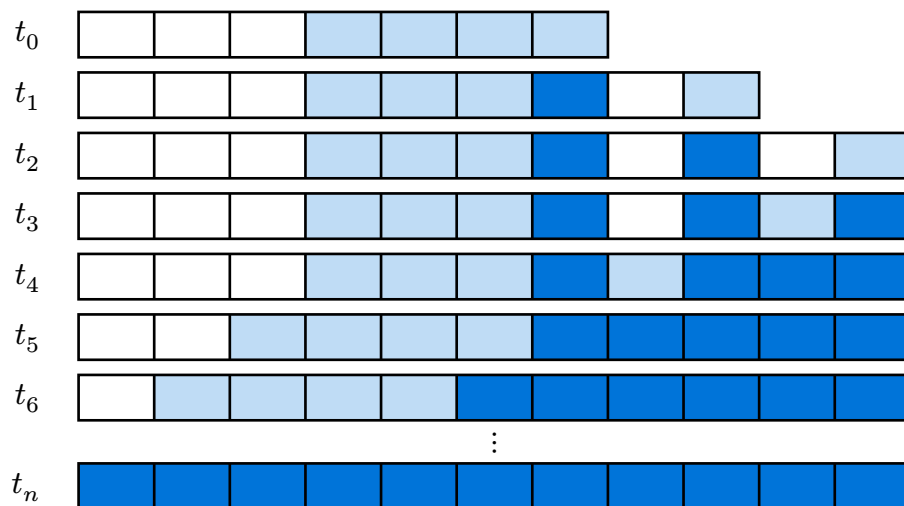
From this data we can see that “certificate-transparency-go” takes 12-15 times more time to parse a certificate, the standard library around 3 times, and Cert Spotter around 6 - 8 times (depending on the machine).

5.3.3. Index filters

A simple way to crawl CT logs is to read from the beginning, or backwards from the end, and store one cursor with the last read index. This is also how our first implementation worked. The downside is that a single cursor makes it hard to revisit parts of a log while continuing to follow new entries at the end. This matters for balancing freshness and completeness, and also for the historization approach described in Section 5.7, where already covered log ranges may need to be inspected again.

Our solution is a data structure called *index filter*, which stores covered ranges of a log. It is a list of half-open index ranges that mark which parts of a log are already covered. The crawler can then ask for the next uncovered index instead of following one global cursor. If a part of the log has to be visited again, the corresponding interval can be removed from the filter,

making it available to the crawler again. Note that a range is already marked as covered when a read request for it is queued. If the request fails or only returns a partial result, the affected range is uncovered again.



5.3.4. Tuning

One reason why it is implemented like this is that, also due to time constraints, we had to work as pragmatically as possible. Over time, focus narrowed on obtaining a practically useful crawler with acceptable CPU and RAM usage. Another difficulty in this is the long feedback loop. After any adjustments, we had to observe resulting crawler behavior for multiple days to see whether a change improved it. In practice, optimizing the crawler therefore somewhat became a pragmatic search through combinations of parameters, with long reaction times between adjustments.

5.4. Certificate retrieval

The design of certificate retrieval is described in Section 4.4. In the implementation, this is represented by the CertChecker interface, which is implemented by the check package. The implementation relies mostly on the Go standard library: `crypto/tls` is used for the TLS connection and `net` is used to connect to the domain on port 443. Every five minutes, Certbat reads the monitored domains from the database and retrieves the deployed certificates. Retrieved certificates are hashed using SHA3-224 over the `tbsCertificate` part and stored as `knowncert` entries together with the corresponding domain. Before hashing, embedded SCT lists and CT poison extensions are removed, such that a certificate and its corresponding precertificate result in the same hash.

5.5. Persistence

To store persistent state such as crawler progress, active certificates, known deployed certificates, domains, alerts, and statistics, we rely on an SQLite database. The database logic is contained in its own Go package, `sqliterepo`. Code that interacts with the database is written against repository interfaces defined in the `certbat` Go package. This allows us to test the database implementation and the code interacting with it independently. Figure 16 depicts the database schema used.

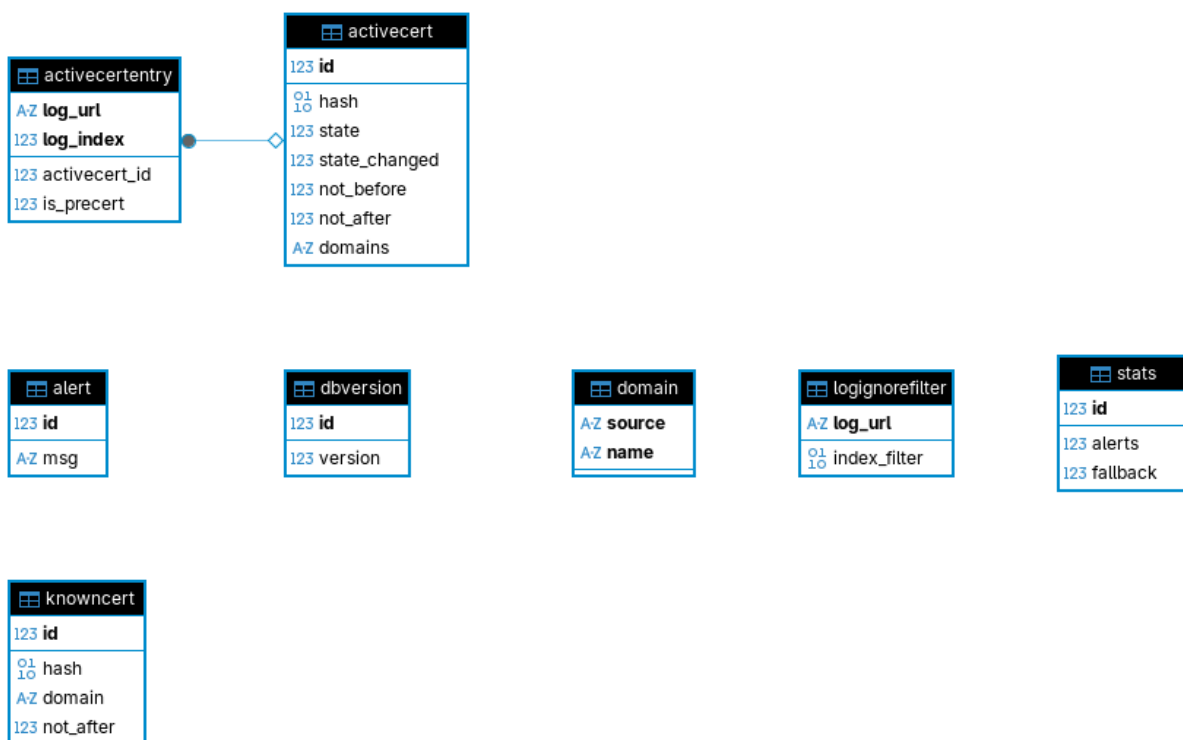


Figure 16: Database schema

The table `activecert` contains one row for every active certificate. Since multiple logs can contain multiple certificate entries related to the same issuance (for example, precertificate and certificate), a separate table `activecertentry` containing a row for every log entry referencing a given certificate is used. All alerts generated by Certbat are stored in the table `alert`. Once they have been delivered, they are removed from this table (see Section 5.6). All observed domains are stored in the `domain` table. The source stores from which provider a given domain

originated (e.g. Nginx). Read positions are stored in `logignorefilter`. It contains a serialized version of the index filter described in Section 5.3.3. The `stats` table is used to keep track of how many alerts have been delivered and how many times fallback parsing as described in Section 5.3.2.1 happened.

Suspicious certificates found in logs are stored as files in a directory on disk, so that users can more conveniently see and analyze them, than if they were saved as blobs in the database. These files are grouped by certificate hash and filenames are derived from the log URL and index.

5.6. Alerting

Alerting mainly needs to be reliable and configurable. It should be possible to deliver alerts using any suitable technology, though users probably especially want to use email. As mentioned in Section 4.5 and Section 5.5, alerts are stored in the SQLite database and a background task reads them from there to deliver them. We also included a test mechanism that sends the contents of a configured test file through the same alert sender and removes the file after successful delivery.

To allow for flexibility, delivering alerts is currently implemented such that Certbat executes a configurable external program or script and passes the message to be sent. The alert message is written to the standard input of the executed program.

For any alert, delivery is retried by the background task until the executed program terminates with exit code 0, assumed to indicate successful delivery. Once this has happened, the alert is removed from the table in the database.

The example configuration in Listing 2 shows the default behaviour, delivering alerts by email to `root@localhost`.

```
[alert]
command = "mail"
args = ["-s", "Certbat Alert", "root@localhost"]
```

Listing 2: Default alert configuration

The `test_file` key is optional and for the mentioned test mechanism. If this specified file exists, Certbat sends an alert with the message found in it and then removes it. This happens directly through the alert sender, without interaction with the database. The example configuration in Listing 3 shows how alerts can instead be written to a file using a shell one-liner.

```
[alert]
command = "bash"
args = ["-c", "echo \"$(date --rfc-3339=seconds): $(cat -)\" >> data/
certbat.alerts"]
```

Listing 3: File alert configuration

5.7. Historization

When crawling through the logs, only certificates related to monitored domains are kept as active certificates. But, what happens if the user configures a new domain, or a new domain is discovered some time after first setting up Certbat? Normally, all the logs would have to be re-crawled in order to correctly handle this, which is too inefficient.

To avoid this, Certbat supports an optional *historization* feature. The naive implementation would be to store and keep all downloaded certificates and the corresponding domains. Assuming this can be done using 50 bytes per certificate and with a backlog of ~30 billion entries, this approach would result in 1.5 TB of used disk space. Even though this feature is optional, this is an impractically large amount of space used given the goals for Certbat.

To optimize this, we decided to trade off some querying efficiency for better storage efficiency. Instead of storing all certificates, observed domains are added to Bloom filters belonging to ranges of CT log entries. This allows Certbat to later find log ranges that may contain certificates for a newly monitored domain. A match in a Bloom filter can therefore cause additional crawling work, but it avoids having to store all certificates.

In our implementation, we first conceptually split a CT log into blocks of $2^{16} = 65,536$ entries. For each block, we then create and keep two Bloom filters, containing the domain names found in the certificates within that block. The first Bloom filter, called *short*, is for certificates with a validity period that is shorter than 4 months. The second Bloom filter, called *long*, is for all other certificates in the block. This split is done to allow more easily deleting Bloom filters for short-lived certificates once the last one in a given block has expired, while preserving long-lived certificates.

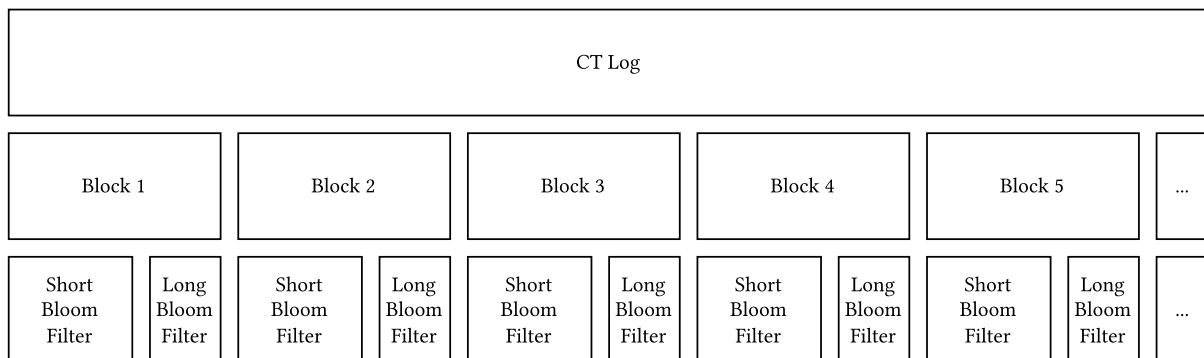


Figure 17: Historization overview for one log

When a certificate is processed for historization, Certbat first determines the block from the certificate's index in the CT log. It then chooses either the *short* or *long* Bloom filter for that block, depending on the certificate's validity period. Finally, the domain names extracted from the certificate are added to the selected Bloom filter.

When a new domain is added, Certbat checks the Bloom filters of all logs for this domain. If a Bloom filter reports a match, the corresponding block is scheduled to be re-crawled. Since Bloom filters can produce false positives, this may cause *some* unnecessary re-crawling, but it should not cause matching certificates to be missed.

To find suitable parameters for the Bloom filters, we used the estimation feature of the used Bloom filter library⁹. It estimates the required Bloom filter size and number of hash functions based on an expected number of distinct entries (n , domain names in our case) and a desired false positive rate (p). Internally it uses the following formula:

$$m = \left\lceil \frac{-1 * n * \log(p)}{\log(2)^2} \right\rceil \quad k = \left\lceil \frac{\log(2) * m}{n} \right\rceil$$

Where m is the size of the Bloom filter in bits and k the number of hash functions used.

At first, we tried to use different Bloom filter sizes for short-lived and long-lived certificates. In particular, we tried to use a smaller Bloom filter for long-lived certificates. However, the ratio between short-lived and long-lived certificates is not constant across logs and blocks, so this fixed split did not work reliably. As an alternative, we tried to keep the domain names of partially read blocks and periodically persist them in the SQLite database. This would have allowed us to count the number of distinct domain names in a completed block and create a Bloom filter with a more fitting size. In practice, however, persisting like that caused long delays, in some cases up to two minutes during a five minute interval.

Our final approach is to use an estimate of ~120'000 distinct domains per block for both short- and long-lived filters and additionally compress the filters using gzip. For blocks containing fewer distinct domains than the estimate, gzip reduced the filters to approximately one third of their uncompressed size. Bloom filters for partially read blocks are kept in memory and periodically flushed to disk. After flushing, the in-memory Bloom filters are removed to avoid an unbounded cache of partial blocks. If a block already has a Bloom filter on disk, the in-memory filter is merged with the existing one before writing it back.

To arrive at the ~120'000 estimate, we used our crawler to read 100 blocks and counted the number of distinct domains found in each block. We plotted the results and concluded that ~120'000 should work well for most blocks, shown in Figure 18.

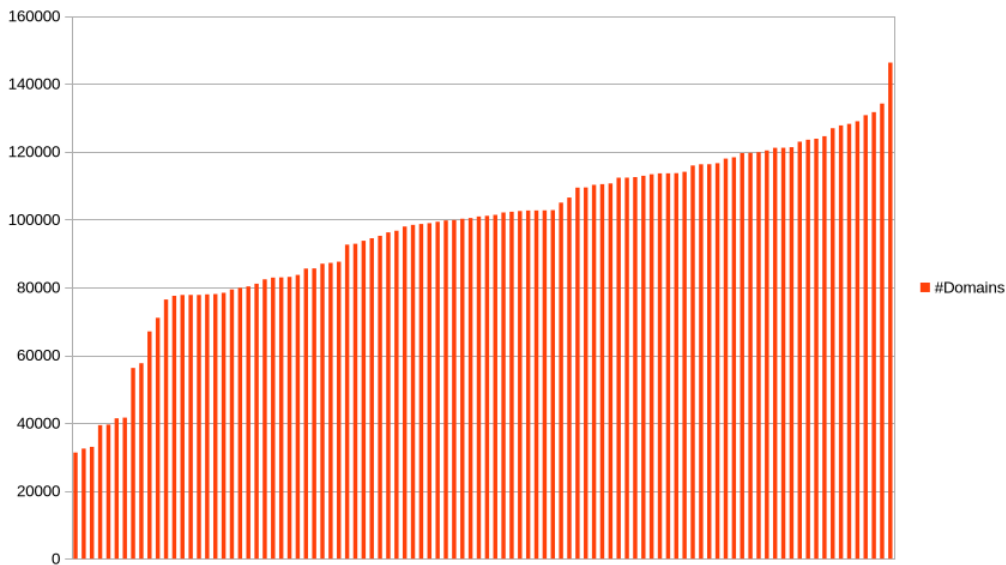


Figure 18: Domains per block

⁹The library used is called `bits-and-blooms/bloom`.

The value for p is configurable and defines the desired false positive rate of one Bloom filter. Using the estimate from above, a smaller value for p reduces false positives when querying the history, but increases storage use. Ignoring rounding errors, the raw size of the two Bloom filters per block in bytes is:

$$2 \cdot \left(\frac{-120 \cdot 10^3 \cdot \ln(p)}{\ln(2)^2 \cdot 8} \right) \quad (4)$$

With $p = 0.0001$, which is what Certbat uses by default, this results in approximately the following number of bytes per block:

$$2 \cdot \left(\frac{-120 \cdot 10^3 \cdot \ln(0.0001)}{\ln(2)^2 \cdot 8} \right) \approx 575'104 \quad (5)$$

The Bloom filter library used performs the rounding and derives the corresponding number of hash functions. For this default, this results in 14 hash functions. At the time of writing, all CT logs combined contain roughly $35 \cdot 10^9$ entries. Since one block contains $2^{16} = 65'536$ entries, this corresponds to approximately the following number of blocks:

$$\frac{35 \cdot 10^9}{2^{16}} \approx 534'057 \quad (6)$$

If every block contained both Bloom filters and no compression or cleanup were taken into account, the Bloom filters alone would therefore require the following number of bytes:

$$575'104 \cdot 534'057 = 307'138'316'928 \quad (7)$$

This is approximately 307 GB and is the upper bound for raw historization data. In practice, storage use is lower due to the compression and automatic removal of expired filters.

After reading 14B certificates, we have around 55GB of data.

5.8. Configurability

Certbat can be configured using a TOML configuration file. The path to this file can be passed using the `-config` command line flag. State such as the SQLite database and certificates is kept in the *data directory*, specified with `-datadir`. The command line interface is intentionally kept small; operational behaviour is configured in the configuration file and documented in the manual page `certbat.toml(5)`.

The configuration is parsed in the `config` package and converted into smaller configuration structures used by the individual subsystems. During parsing, Certbat enforces some validation rules. Unknown keys are rejected, provider names must be present and unique, and specified durations must use the Domain Specific Language (DSL) that can be parsed by Go's `time.ParseDuration` function (e.g. "20m"). An invalid configuration results in an early crash during startup.

Domain discovery providers, as described in Section 4.3, are configured in the same file. Provider sections are decoded by type, and for each provider there are provider specific options such as polling interval, API URL, configuration path or command arguments.

5.9. Statistics and visualization

To make Certbat's behaviour easier to observe over time, a statistics collection mechanism was implemented. Statistics collection is optional and statistics are stored in a separate SQLite database. They are periodically collected from subsystems. From the crawler, Certbat records per-log statistics such as the number of requests, downloaded entries, errors, rate limits, downloaded bytes and current tree size. From the historization subsystem, it records the storage used by Bloom filters per log.



Figure 19: Screenshot of visualization, showing how four logs were scraped over time.

Another optional feature is the visualization of some of these statistics. When enabled, Certbat renders a small HTML dashboard containing SVG charts and saves it as a file. If configured, this file is additionally served over HTTP. Figure 19 shows an example of the visualization.

5.10. Distribution

For distribution, the *GoReleaser*¹⁰ release automation tool is used. It builds the certbat binary, generates man pages and creates native Linux packages. Currently it is configured to build packages for Debian-based distributions (`.deb`), RPM-based distributions (`.rpm`), Alpine Linux (`.apk`) and Arch Linux.

The packages install the executable binary, man pages, a systemd service unit and a default configuration file at `/etc/certbat/certbat.toml`. The systemd service runs Certbat with `-config /etc/certbat/certbat.toml` and `-datadir /var/lib/certbat`. Systemd is responsible for starting Certbat and restarting it on failure. Also, the following hardening options are configured in the systemd unit: Certbat is run as a dynamic user with restricted access to the filesystem. It has write access to the data directory and a private temp directory. The installed default configuration file documents available options and defaults as commented examples.

¹⁰An open-source tool which is often used in Go projects to make the packaging and release process easy.

6. Evaluation

Section 1.4 lists the design goals of Certbat. In Section 6.1, we assess whether these goals are met. Some can be verified directly based on the implementation and packaging. For others we have defined and carried out manual tests. Lastly, for goals that are quantitatively assessable, we provide measurements and calculations. Section 6.2 then assesses whether current prevalent CT logs can be monitored within their rate limits at all.

6.1. Design goals

Goal G10 is met by construction: Certbat was implemented as free and open-source software and is licensed under the GNU Affero General Public License, version 3. The license is provided with the source code and license headers have been added to the files containing the code. The source code is available on Codeberg¹¹. The remaining goals are assessed in turn.

6.1.1. Installation and configuration

G1: Certbat should be easily installable and configurable and provide sane default configuration.

The distribution packages that get built as described in Section 5.10 contain manual pages and a default configuration with commented out example sections for all options. The default configuration provides what we objectively deem to be a sane starting point. In order to start using Certbat based on it, the minimal required addition needed by a user is one or more domains to be monitored. The manual and examples provided help configure this. Prebuilt packages are made available next to the source code on Codeberg.

6.1.1.1. Manual test: Installation on Debian

To test whether the installation package works, the generated dpkg package was copied to a freshly installed Debian virtual machine. Then, it was installed using `dpkg -i`. After that, a domain to be monitored was added in the file `/etc/certbat/certbat.toml`. With this set, certbat was started using `systemctl start certbat`. Lastly, `systemctl status certbat` reported Certbat as running and showed some recent logs.

6.1.2. Domain discovery and configuration

G2: Certbat should support automatic discovery and manual configuration of monitored domains.

Automatic discovery of domains is given by the provider implementations for different web servers described in Section 5.2. Manual configuration is also implemented, how the set of monitored domains is obtained is described in Section 4.3.

6.1.2.1. Manual test: Web server containers

This was manually tested during the implementation. For this, existing container images for the four supported web server providers were used with a basic configuration. To confirm that domain configuration changes are discovered, Certbat's database was inspected.

6.1.3. Resource consumption and comparison with Cert Spotter

G3: Certbat should retrieve and process entries from CT logs efficiently.

¹¹<https://codeberg.org/certbat/certbat>

To make “*efficiently*” measurable, we use Cert Spotter as introduced in Section 3 as a baseline. For this evaluation, we consider Certbat more efficient if it processes *more* log entries in the same time while using less CPU time and memory. In order to measure this, we provisioned identical servers using the *Hetzner Cloud*^o service.

PROPERTY	VALUE
Server type	CCX23 (General Purpose)
vCPUs	4
RAM	16 GB
Storage	160 GB
Operating system	Ubuntu 26.04 LTS
Kernel	Linux 7.0.0-15-generic
Architecture	x86_64
Location	Nuremberg, Germany
Network zone	eu-central

Table 5: Server configuration used for evaluations.

Table 5 shows the server configuration used. Note that this configuration was also chosen based on what we believe are approximately the resources a domain owner hosting a personal website would have access to.

Using three such servers, we ran Certbat with and without the historization feature on the first and second server and Cert Spotter on the third server. During the measurement period, all three servers were used exclusively for this comparison. CPU and RAM consumption was measured using the `pidstat` command line utility for Linux. To measure the progress of Certbat, the statistics mechanism as described in Section 5.9 was used. Cert Spotter records its per-log positions in a text file. We parsed this file to extract the reported positions and associated them with timestamps from the log output. A limitation of this method is that Cert Spotter does not report every log position at regular intervals. We created two little utilities to obtain the data points plotted in Figures 20 to 22. They can be found under `certspottercmp` in the source code.

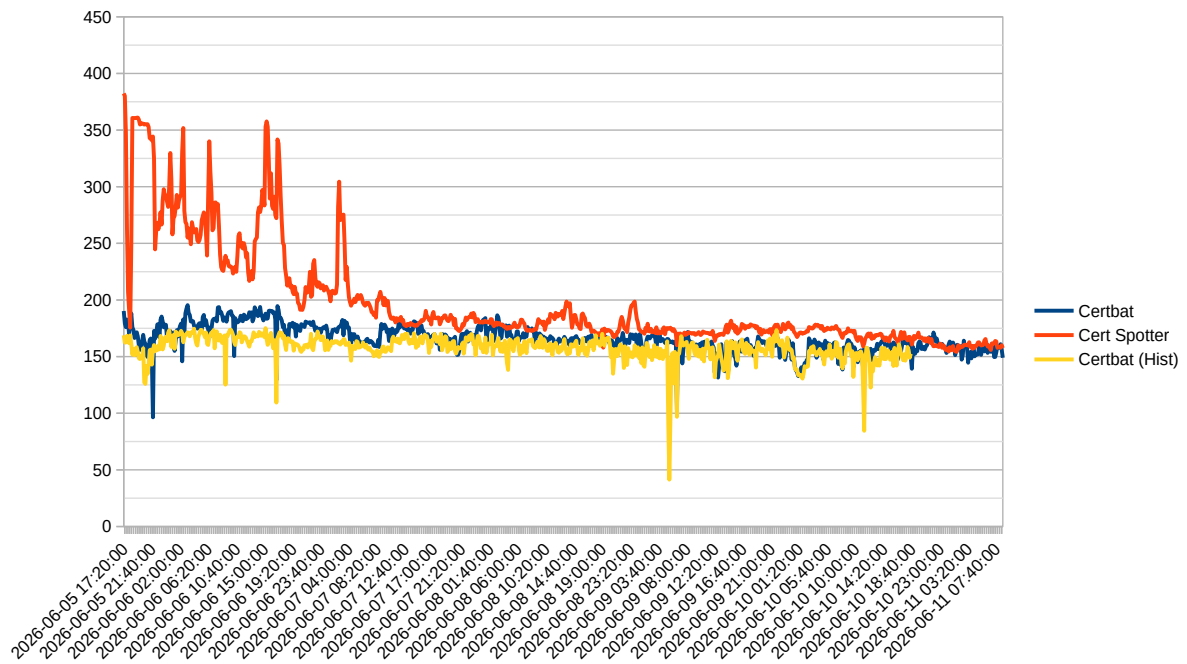


Figure 20: Comparison CPU consumption of Certbat and Cert Spotter

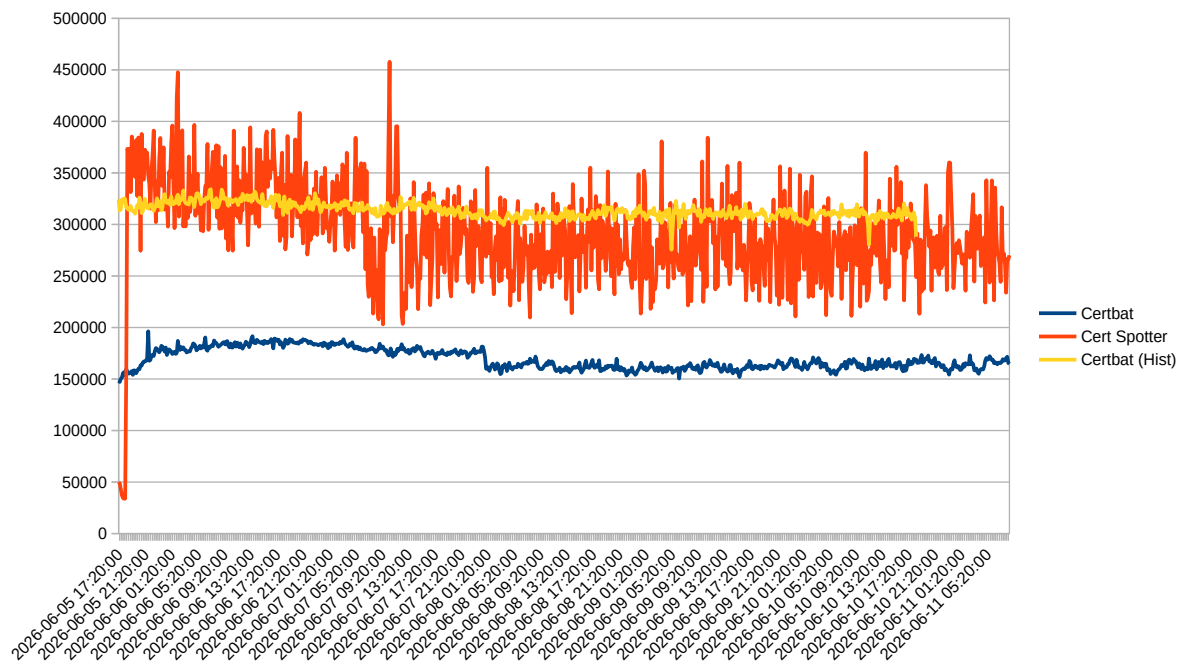


Figure 21: Comparison RAM consumption of Certbat and Cert Spotter

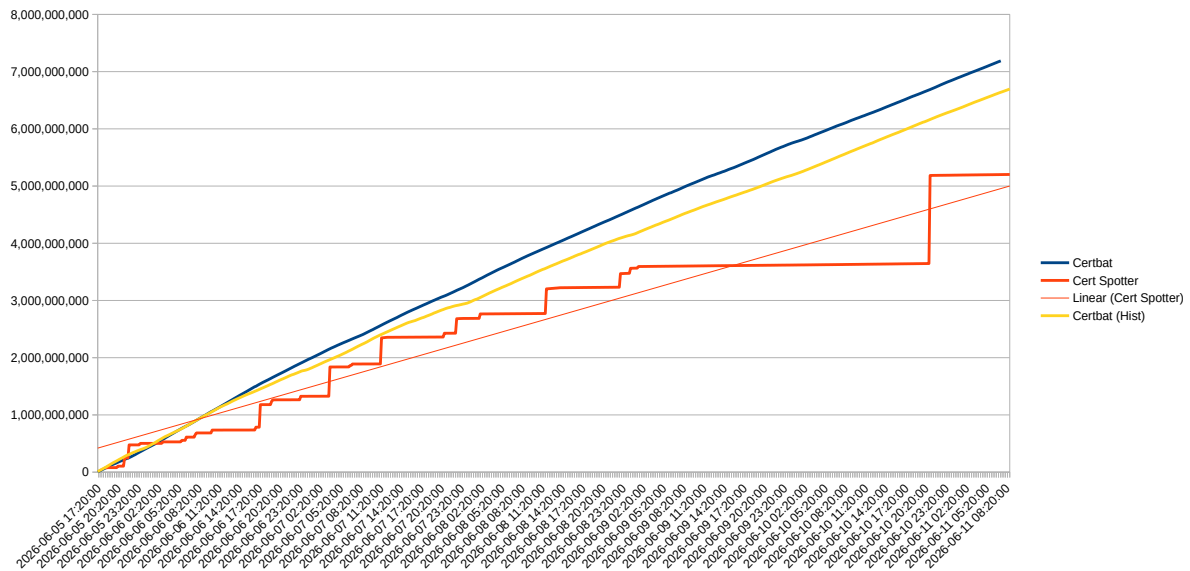


Figure 22: Comparison of consumed Log Parts of Certbat and Cert Spotter

Figure 20 shows that Certbat used less CPU than Cert Spotter over the measured period. Figure 21 shows lower memory usage when historization is turned off and comparable memory usage when it is turned on. Figure 22 shows that Certbat advanced faster through the logs, with and without historization. The jagged line for Cert Spotter in Figure 22 is caused by the irregular intervals in which Cert Spotter reports its positions, as explained above. To make the overall trend easier to compare, a linear approximation was added for Cert Spotter. This approximation is a visual aid to show how Cert Spotter is projected not to catch up.

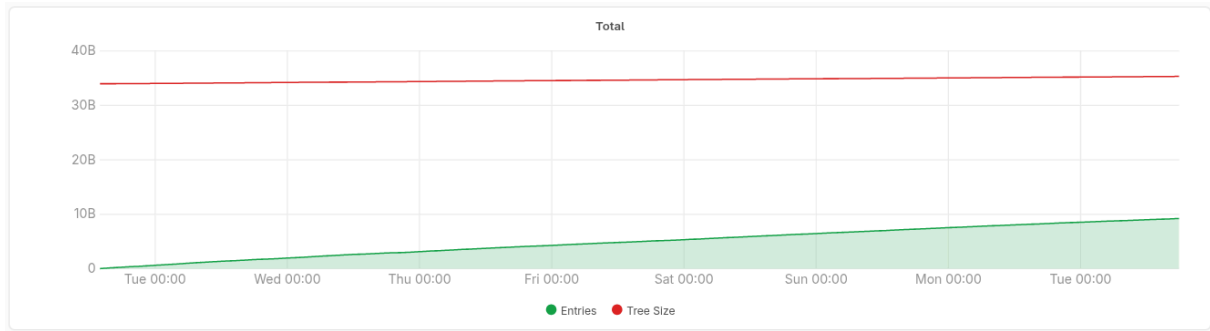


Figure 23: Crawl progress across logs.

Figure 23 shows how Certbat with historization enabled was able to crawl approximately 10 B entries in 7 days on one of the Hetzner servers. If crawl performance stays more or less constant, this means that it takes 1 month to fully catch up.

6.1.4. Storage efficiency

G4: Certbat should store downloaded entries efficiently and avoid unnecessary re-downloads.

This is covered by the implementation of historization and index filters, as described in Section 5.7 and Section 5.3.3. Instead of storing all downloaded certificates, Certbat stores compact Bloom filters for log ranges and uses index filters to mark which ranges have already been covered.

6.1.5. Rate limits

G5: Certbat should respect and adhere to the rate limits of CT logs.

This is covered by the crawler’s per-log rate limiting behavior, described in Section 5.3.1. Certbat reacts to HTTP 429 responses by increasing the delay between requests for the affected log and reduces the delay again when requests succeed. In long-running measurements, this kept rate limit responses low while allowing the crawler to continue making progress, so we consider this goal met.

6.1.6. Certificate matching

G6: Certbat should identify certificates corresponding to the configured domains.

Certbat extracts domain names from certificates downloaded by the crawler and compares them with the configured and discovered domains. Matching certificates are then handled according to the lifecycle described in Section 4.5.

This was tested manually in the same test used for G8 (Section 6.1.8.1).

6.1.7. Deployed certificates

G7: Certbat should retrieve deployed certificates to compare them with the identified certificates.

Certbat periodically connects to the monitored domains over TLS and stores hashes for the certificates observed there, as described in Section 5.4. Then, comparisons are done with the certificates found in CT logs according to the lifecycle in Section 4.5.

This was tested manually in the same test used for G8 (Section 6.1.8.1).

6.1.8. Alerting

G8: Certbat should alert timely and reasonably and avoid alerting unnecessarily.

This goal is addressed by two parts of the design. First, the crawler prioritizes the tail of each log, so newly logged certificates are processed before older backlog entries. This allows newly issued certificates to be detected quickly, resulting in timely alerts for them. Second, Certbat compares certificates found in CT logs with the certificates currently deployed on the monitored domains. If a certificate is already deployed, it is treated as expected and no alert is sent. If it is not observed as deployed, it first remains suspicious for a configurable delay before an alert is sent, as described in Section 4.5. This delay avoids unnecessary alerts during normal issuance and deployment, where a certificate may appear in CT logs before it is installed on the web server. See also Section 5.6.

6.1.8.1. Manual test: Redundant certificate issuance

To verify that this works correctly, we issued a *redundant* certificate for one of the configured monitored domains without deploying it. We configured the domain books `.domi.work` to be monitored and issued a wildcard certificate containing `*.domi.work` and `domi.work`.

We issued the domain from Let's Encrypt using a DNS-01 challenge at 12:07. In order to do the challenge, we used RFC 2136 DNS Dynamic Updates. Certbat sent out an alert for it at 12:21, 14 minutes later. We also inspected the certificate files Certbat saved on disk. At 12:17, it had found the precertificate in the Tuscolo2026h2 CT log and saved it. Later, between 12:20 and 12:24, it found the final certificate containing two SCTs in multiple logs (note that despite containing only two SCTs, Let's Encrypt submitted it to more than two logs).

6.1.9. Performance alongside a web server

G9: Certbat should run with reasonable performance assuming deployment alongside a web server.

For the assessment of this, refer to Section 6.1.3. Although for the measurements there no web server software was run next to Certbat, both the measured CPU times and RAM usage were consistently low enough to not tangibly affect web server software running on the same server.

6.2. Feasibility of monitoring current logs

6.2.1. Log growth versus rate limits

Do current prevalent CT logs grow faster than they can be scraped within their rate limits?

During development, Certbat was run and observed for long periods. Based on these observations and on manual tests, we give the following assessment. In our opinion, each log can be scraped faster than new entries are added to it, if the log is scraped in isolation. This means that we did not observe a log for which the configured rate limits would prevent a monitor from eventually catching up. The harder problem is scraping all prevalent logs at the same time, because request capacity, CPU time and memory have to be distributed across them. For this, careful scheduling as implemented in Certbat's crawler is necessary. Even with such scheduling, some logs take very long to be fully scraped because their rate limits are restrictive and they serve small batch sizes of entries. Catch up is still possible, just very slow.

Logs supporting the Static CT API improve the situation substantially. Assuming that all logs eventually switch to the Static CT API, monitoring should become more practical.

6.2.2. Reliability of monitoring

Is monitoring of current prevalent CT logs feasible to guarantee correct operation of Certbat?

In our opinion, based on running Certbat for longer periods of time, monitoring is feasible to guarantee correct operation of it. It must be stated though, that logs' behavior and rate limits as well as the set of logs to monitor itself can change. Due to this, we cannot give a definitive assessment but only a speculation. The logs where catch-up was most difficult were those operated by *TrustAsia*. Starting with the logs for 2027, they are switching to provide Static CT API logs, which should improve the situation [49].

7. Discussion

7.1. Limitations of Certbat

Apart from the problems due to the CT Log APIs, Certbat in its current state has limitations.

7.1.1. SQLite

SQLite is a solid database. But in our software design, we did not consider that it can only have one open transaction at a time. This hinders parallelization in the crawler and the historization feature. We could possibly get more throughput by addressing this, though this might be at the expense of higher CPU load.

7.1.2. Telemetry

With the statistics database, we have some insights about Certbat at runtime. It would be even better if we had more such telemetry data, for example by using specialized libraries for this purpose. This could help optimize Certbat more effectively.

7.1.3. Domain discovery

Domain discovery is dependent on the chosen web server providers. These offer multiple ways to configure routing and virtual hosts including some more sophisticated ones like using regular expressions. Currently, Certbat cannot handle these because it cannot know which certificates to retrieve for such cases.

7.2. Limitations of current CT log APIs

Our main problem with the current CT Log system is that it is required to download whole logs. That is, we need to download all certificates for every domain even though we are only interested in all certificates for our own domains.

One certificate entry is between 1.5 (Static CT API) and 7 KB (RFC 6962 API). After downloading exactly $9'025'735'136$ certificates we had downloaded $36'534'793'041'026$ bytes, resulting in $\frac{36'534'793'041'026}{9'025'735'136} \approx 4'048$ bytes per certificate entry. At the time of writing, there are around $35'000'000'000$ certificate entries in all logs combined. This results in a total data volume from $(35 \cdot 10^9 \cdot 4 \cdot 10^3 = 35 \cdot 4 \cdot 10^{12} = 140 \cdot 10^{12})$ bytes (140 TB) of data to download. This is a lot of data to download, especially for an individual domain owner. But it is even worse for log operators, because they have to serve the data to *every* domain owner. Because there is currently no index of domain names, it is not possible for domain owners to only download the parts that contain certificates for their own domains.

7.3. Proposed changes to CT log APIs

Under the assumption that only domain owners can authoritatively decide which certificates are legitimate for their domains, they have to check the logs. But there should be a better way to do this than downloading whole logs.

We think that it is enough if they only download and check the certificates for their own domains. To do that, they need to know in which logs at which positions these certificates are.

Our solution to that is based on our historization strategy but with some important changes. First, with only one Bloom filter per block. Second, the Bloom filter itself needs to be signed and added to an append-only CT-like log, such that misbehaving log operators are detectable.

With this, the total data that needs to be downloaded can be reduced. We used the following formula to decide what the total size to download needs to be. The total number of entries ($35 \cdot 10^9$) is explained under Section 7.2. The size of one Bloom filter is explained under Section 5.7.

Total number of blocks (number of certificate entries / entries per block):

$$n = \frac{35 \cdot 10^9}{2^{16}} \approx 534'058 \quad (8)$$

Storage required per block (with p for the false positive rate of the Bloom filter):

$$\frac{-120'000 \cdot \ln(p)}{\ln(2)^2} \quad (9)$$

The total number of data to download is given by the size of all Bloom filters plus the number of blocks that need to be downloaded, either because an interesting certificate is inside the block, or because it is a false positive. To calculate the minimum number of data to download we only consider the false positives.

$$D = 534'058 \cdot \frac{-120 \cdot 10^3 \cdot \ln(p)}{\ln(2)^2 \cdot 8} + p \cdot 140 \cdot 10^{12} \quad (10)$$

Where D is the total number of bytes to download. We can then choose p to minimize this expression. This gives us $p \approx 0.00012$, resulting in a total download size D (in bytes):

$$D = 534'058 \cdot \frac{-120 \cdot 10^3 \cdot \ln(0.00012)}{\ln(2)^2 \cdot 8} + 0.00012 \cdot 140 \cdot 10^{12} \approx 167 \cdot 10^9. \quad (11)$$

That is about 170 GB. This is approximately $\frac{1}{800}$ of the total size of 140'000 GB.

This method works as long as we know exactly which domains we have to search for but fails if we want to perform more complex queries. For example, it allows us to find all certificates that are valid exactly for our domain. But it is not possible to find all certificates for our domain, including all (known and unknown) subdomains. To work around this, we consider the following two approaches.

7.3.1. Approach 1: Domain + eTLD

Instead of storing the full domains, we only store the effective top level domain (eTLD) plus the registered domain. This is effectively what internet users can register (referred to as eTLD + 1). For example for `www.dom1.work` we store `dom1.work` since that is what can be registered.

This decreases the block size and allows to find all subdomains of a registered domain, but might lead to more false positives, depending on the query. We measured the number of distinct eTLD + 1 domains for 100 blocks, shown in Figure 24.

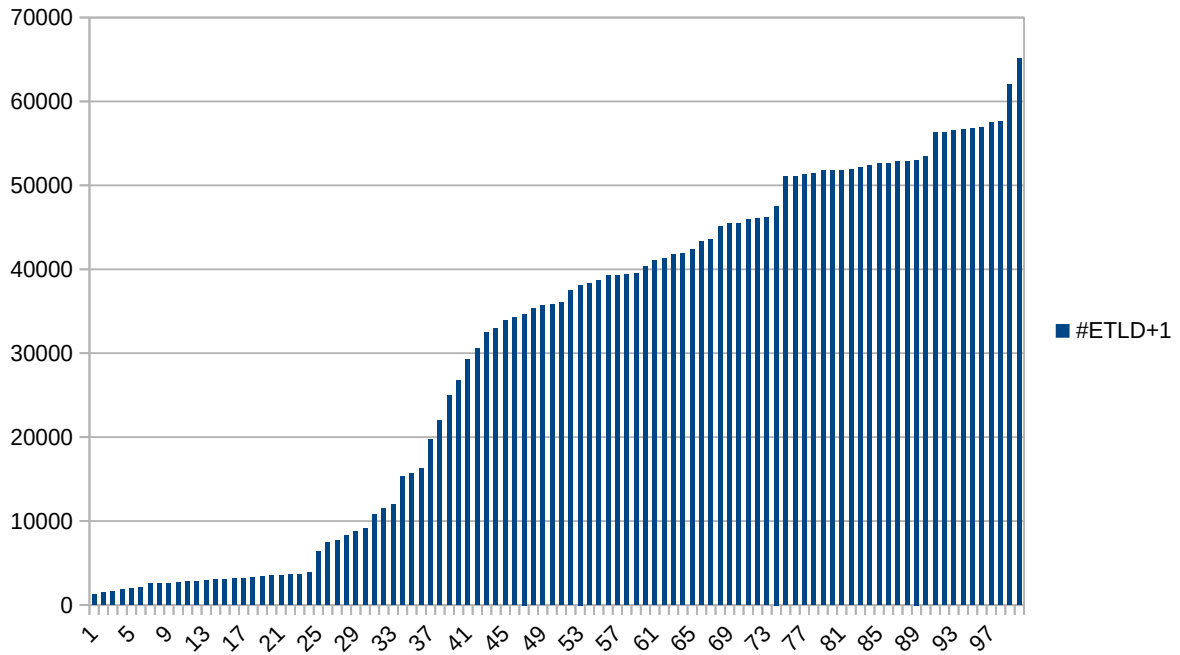


Figure 24: Number of distinct eTLD + 1 domains for 100 blocks

From this we conclude that we can use Bloom filters that are good for around 60K distinct entries. Using the same calculations as above, we get $p \approx 0.000029$ and $D \approx 46 \cdot 10^9$. That is around 46 GB to download. But we can only query for eTLD + 1, which might be sufficient. This is approximately $\frac{1}{3000}$ of the total download size, a big improvement.

7.3.2. Approach 2: Name blocks

Instead of storing the domain names in Bloom filters, store them in plain text (similar to the names tiles described in Section 2.2.6.1). This allows arbitrary queries and results in no false positives, but requires more downloaded data per block.

We measured the size and the compressed size of all distinct domain names for 100 blocks, shown in Figure 25.

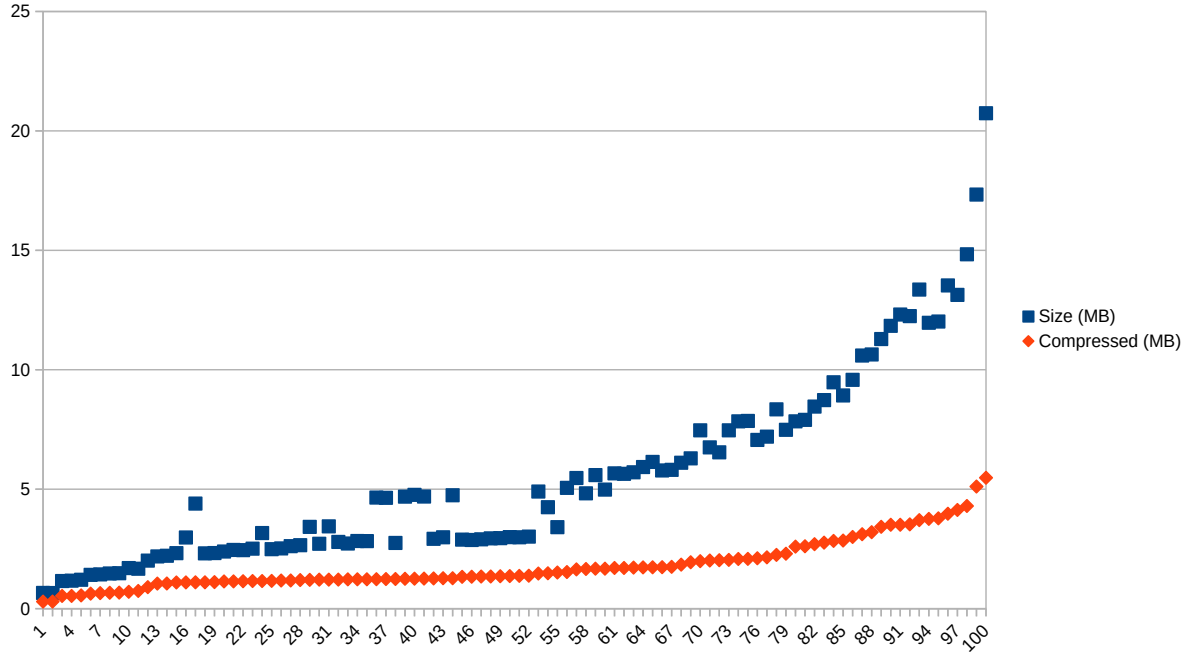


Figure 25: Size of the distinct domain names per block for 100 blocks

With this information, we assume 3 MB per block on average. For 534'058 blocks, this gives us:

$$D = 534'058 \cdot 3 = 1'602'174 \quad (12)$$

This is around 1.6 TB, approximately $\frac{1}{87}$ of the full download.

7.3.3. Comparison

VARIANT	SIZE	SIZE FACTOR	COMMENT
Download the whole logs	140'000 GB	1	Currently the only reliable option
Download names only	1.6 TB	$\frac{1}{87}$	Allows arbitrary queries but is still a lot to download
Store all domains in Bloom filters	167 GB	$\frac{1}{800}$	Only allows exact queries
Only store eTLD + 1	46 GB	$\frac{1}{3000}$	Only allows to query for eTLD + 1

Table 6: Download data size comparison

Table 6 shows the results in a table. We can see that the approach using eTLD + 1 results in the biggest improvement. For domain owners, this approach would still allow them to find certificates for their domains.

8. Conclusion

One of the fundamental issues of the PKI system is that any CA can issue certificates for any domain. Certificate Transparency helps mitigate this by introducing transparency and observability. Even if logged, illegitimately or incorrectly issued certificates can still go unnoticed. In order to detect such certificates for their domains, domain owners must monitor the logs. This is currently a challenging and inaccessible task, especially for independent domain owners with limited resources.

This report presented Certbat, a free software CT log monitor aimed at improving this situation. Certbat can automatically discover domains from web servers and automatically retrieve deployed certificates. It is able to crawl all logs with relatively low resource consumption. Detected certificates don't immediately result in alerts but are checked against the deployed certificates, minimizing false-positives and overwhelming alerts. It is available as easy-to-install packages including a sane default configuration. These features lower the barrier of entry for domain owners wanting to start monitoring and thus address some of the accessibility issues. Certbat's implementation was evaluated using manual tests and measurements.

Even though Certbat might be an improvement over alternatives like Cert Spotter, it still can't solve those issues of CT which are caused by the protocol. Namely, high volumes of data needed to be downloaded. To address this, we proposed a solution which would allow downloading only a small fraction of the total data using Bloom filters.

Acknowledgements

This report is based on the authors' Bachelor thesis in Computer Science at Bern University of Applied Sciences, School of Engineering and Computer Science, submitted in June 2026. We thank our advisor, Prof. Dr. Christian Grothoff, for proposing the project, inspiring the design goals and guiding us throughout the work. We also thank Han van der Kleij, who acted as external expert for the thesis.

Bibliography

- [1] R.-G. Holz, “Empirical Analysis of Public Key Infrastructures and Investigation of Improvements,” Doctoral dissertation, 2014. Accessed: Apr. 27, 2026. [Online]. Available: <https://mediatum.ub.tum.de/node?id=1182735>^o
- [2] “Certificate Transparency (CT) Logs.” Accessed: May 14, 2026. [Online]. Available: <https://letsencrypt.org/docs/ct-logs/>^o
- [3] “How CT Works : Certificate Transparency.” Accessed: Mar. 16, 2026. [Online]. Available: <https://certificate.transparency.dev/howctworks/>^o
- [4] “Certspotter/README.Md at 68a01db23fa33c1423441db7f8215b7436ac90c4 · SSLMate/Certspotter”. Accessed: May 30, 2026. [Online]. Available: <https://github.com/SSLMate/certspotter/blob/68a01db23fa33c1423441db7f8215b7436ac90c4/README.md>^o
- [5] DEFCONConference, “DEF CON 19 - Moxie Marlinspike - SSL And The Future Of Authenticity.” Accessed: Mar. 18, 2026. [Online]. Available: https://www.youtube.com/watch?v=UawS3_iuHoA^o
- [6] M. Marlinspike, “SSL And The Future Of Authenticity (RSA Conference 2012).” Accessed: Mar. 18, 2026. [Online]. Available: <https://docs.huihoo.com/rsaconference/usa-2012/SSL-and-the-Future-of-Authenticity.pdf>^o
- [7] M. Azer, A. Shaker, S. Nagy, A. Raafat, and M. Abuzaid, “Organizational Risk Assessment Based on Attacks Repetition,” in *2012 Seventh International Conference on Availability, Reliability and Security*, Aug. 2012, pp. 286–290. doi: 10.1109/ARES.2012.90^o
- [8] “The Comodo Fall-out; Can We Trust Websites Again? | Simply Security.” Accessed: Mar. 18, 2026. [Online]. Available: <https://web.archive.org/web/20120622034623/https://www.simplysecurity.com/2011/05/16/the-comodo-fall-out-can-we-trust-websites-again/>^o
- [9] “Comodo Report of Incident - Comodo Detected and Thwarted an Intrusion on 26-MAR-2011.” Accessed: Mar. 18, 2026. [Online]. Available: <https://www.comodo.com/Comodo-Fraud-Incident-2011-03-23.html>^o
- [10] N. van der Meulen, “DigiNotar: Dissecting the First Dutch Digital Disaster,” *Journal of Strategic Security*, vol. 6, no. 2, pp. 46–58, 2013, Accessed: Mar. 18, 2026. [Online]. Available: <https://www.jstor.org/stable/26466760>^o
- [11] F. Kovacs, “Full Disclosure: Is This MITM Attack to Gmail's SSL ?.” Accessed: Mar. 18, 2026. [Online]. Available: <https://seclists.org/fulldisclosure/2011/Aug/348>^o
- [12] B. Laurie, “Certificate Transparency: Public, Verifiable, Append-Only Logs,” *ACM Queue*, vol. 12, no. 8, pp. 10–19, Aug. 2014, doi: 10.1145/2668152.2668154^o
- [13] “(PDF) Black Tulip Report of the Investigation into the DigiNotar Certificate Authority Breach.” Accessed: Mar. 18, 2026. [Online]. Available: https://www.researchgate.net/publication/269333601_Black_Tulip_Report_of_the_investigation_into_the_DigiNotar_Certificate_Authority_breach^o
- [14] C. Ellison and B. Schneier, “Ten Risks of PKI: What You're Not Being Told about Public Key Infrastructure,” 2000, doi: 10.1201/9780203498156.ch23^o
- [15] P. Gutmann, “Everything You Never Wanted to Know about PKI but Were Forced to Find Out,” Nov. 2002, Accessed: Jun. 10, 2026. [Online]. Available: <https://www.cs.auckland.ac.nz/~pgut001/pubs/pkitutorial.pdf>^o
- [16] B. Laurie and G. S. Team, “Improving SSL Certificate Security.” Accessed: Mar. 18, 2026. [Online]. Available: <https://security.googleblog.com/2011/04/improving-ssl-certificate-security.html>^o
- [17] B. Laurie, A. Langley, and E. Kasper, “Certificate Transparency,” Request for Comments RFC 6962, Jun. 2013. doi: 10.17487/RFC6962^o

- [18] “CertificateTransparency/Changes.Md at Main · GoogleChrome/CertificateTransparency”. Accessed: Mar. 18, 2026. [Online]. Available: <https://github.com/GoogleChrome/CertificateTransparency/blob/main/changes.md>
- [19] “User Agents : Certificate Transparency.” Accessed: Mar. 18, 2026. [Online]. Available: <https://certificate.transparency.dev/useragents/>
- [20] “Certificate Transparency : Certificate Transparency.” Accessed: Mar. 16, 2026. [Online]. Available: <https://certificate.transparency.dev/>
- [21] “SecurityEngineering/Certificate Transparency - MozillaWiki.” Accessed: Mar. 19, 2026. [Online]. Available: https://wiki.mozilla.org/SecurityEngineering/Certificate_Transparency
- [22] “Certificate Transparency Log Policy.” Accessed: Mar. 19, 2026. [Online]. Available: https://googlechrome.github.io/CertificateTransparency/log_policy.html
- [23] “Apple’s Certificate Transparency Log Program.” Accessed: Mar. 19, 2026. [Online]. Available: <https://support.apple.com/en-us/103703>
- [24] “Chrome Certificate Transparency Policy.” Accessed: Mar. 19, 2026. [Online]. Available: https://googlechrome.github.io/CertificateTransparency/ct_policy.html
- [25] B. Laurie, E. Messeri, and R. Stradling, “Certificate Transparency Version 2.0,” technical report RFC9162, Dec. 2021. doi: 10.17487/RFC9162
- [26] “Certificate Transparency Versions and Status of CTv2 - Issuance Policy.” Accessed: Mar. 19, 2026. [Online]. Available: <https://community.letsencrypt.org/t/certificate-transparency-versions-and-status-of-ctv2/218492/4>
- [27] “C2SP/Static-Ct-API.Md at Main · C2SP/C2SP”. Accessed: Mar. 19, 2026. [Online]. Available: <https://github.com/C2SP/C2SP/blob/main/static-ct-api.md>
- [28] “The Sunlight CT Log.” Accessed: Mar. 19, 2026. [Online]. Available: <https://sunlight.dev/>
- [29] “A different kind of CT log.” Accessed: Mar. 19, 2026. [Online]. Available: https://docs.google.com/document/d/1YsxLGZxYE1KTCTjDK2OI-bcTzbrI313SZ1QWgqmnRDc/edit?usp=embed_facebook
- [30] “Sunlight/Names-Tiles.Md at 39d20c4a0729016b8573e8ac5be09cee6377b524 · FiloSottile/Sunlight”. Accessed: Jun. 03, 2026. [Online]. Available: <https://github.com/FiloSottile/sunlight/blob/39d20c4a0729016b8573e8ac5be09cee6377b524/names-tiles.md>
- [31] “The Names Tiles Static CT API Extension.” Accessed: Apr. 30, 2026. [Online]. Available: <https://groups.google.com/a/chromium.org/g/ct-policy/c/5T0VUHVghuU/m/hRDHM-39AgAJ>
- [32] “Certspotter/LICENSE at 68a01db23fa33c1423441db7f8215b7436ac90c4 · SSLMate/Certspotter”. Accessed: May 30, 2026. [Online]. Available: <https://github.com/SSLMate/certspotter/blob/68a01db23fa33c1423441db7f8215b7436ac90c4/LICENSE>
- [33] “The Mozilla Public License Version 2.0 Is out—and GPL-compatible! — Free Software Foundation — Working Together for Free Software.” Accessed: May 30, 2026. [Online]. Available: <https://www.fsf.org/blogs/licensing/mpl-2.0-release>
- [34] “Andrew Ayer.” Accessed: May 30, 2026. [Online]. Available: <https://www.agwa.name/>
- [35] “Debian – Details of Package Certspotter in Trixie.” Accessed: May 30, 2026. [Online]. Available: <https://packages.debian.org/trixie/certspotter>
- [36] “AUR (En) - Certspotter.” Accessed: May 30, 2026. [Online]. Available: <https://aur.archlinux.org/packages/certspotter>
- [37] “Certspotter << Security - Ports - FreeBSD Ports Tree.” Accessed: May 30, 2026. [Online]. Available: <https://cgkit.freebsd.org/ports/tree/security/certspotter>

- [38] "Release v0.24.1 c · SSLMate/Certspotter@68a01db". Accessed: May 30, 2026. [Online]. Available: <https://github.com/SSLMate/certspotter/commit/68a01db23fa33c1423441db7f8215b7436ac90c4>^o
- [39] "Indeed, a CT Monitor Which Sends Alerts about Legitimate Certificates Is Almost ... | Hacker News." Accessed: Jun. 03, 2026. [Online]. Available: <https://news.ycombinator.com/item?id=37958471>^o
- [40] "Monitors : Certificate Transparency." Accessed: Mar. 16, 2026. [Online]. Available: <https://certificate.transparency.dev/monitors/>^o
- [41] "Git.Glasklar.Is/Rgdd/Silentct/-/Raw/E3a631e7c86562553598d7fb2e8cf27df100e2fb/AUTHORS." Accessed: Jun. 06, 2026. [Online]. Available: <https://git.glasklar.is/rgdd/silentct/-/raw/e3a631e7c86562553598d7fb2e8cf27df100e2fb/AUTHORS>^o
- [42] B. Li *et al.*, "Certificate Transparency in the Wild: Exploring the Reliability of Monitors," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, in CCS '19. New York, NY, USA: Association for Computing Machinery, Nov. 2019, pp. 2505–2520. doi: 10.1145/3319535.3345653^o.
- [43] B. Li *et al.*, "The Invisible Side of Certificate Transparency: Exploring the Reliability of Monitors in the Wild," *IEEE/ACM Transactions on Networking*, vol. 30, no. 2, pp. 749–765, Apr. 2022, doi: 10.1109/TNET.2021.3123507^o.
- [44] "Verifiable Indexes for a More Usable Transparency Ecosystem | Trillian." Accessed: Jun. 04, 2026. [Online]. Available: <https://transparency.dev/summit2025/talks/verifiable-indexes.html>^o
- [45] "Verifiable Indexes for a More Usable Transparency Ecosystem - Martin Hutchinson." Accessed: Jun. 04, 2026. [Online]. Available: <https://www.youtube.com/watch?v=gfrXTgmbS1s>^o
- [46] R. Dahlberg and T. Pulls, "Verifiable Light-Weight Monitoring for Certificate Transparency Logs." Accessed: Jun. 04, 2026. [Online]. Available: <https://arxiv.org/abs/1711.03952v2>^o
- [47] "Introducing TesseractCT." Accessed: Jun. 09, 2026. [Online]. Available: <https://blog.transparency.dev/introducing-tesseract>^o
- [48] S. Floyd and E. Kohler, "Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 2: TCP-like Congestion Control." [Online]. Available: <https://www.rfc-editor.org/info/rfc4341>^o
- [49] "CT Log Operator: TrustAsia [40917853] - Chromium." Accessed: Jun. 11, 2026. [Online]. Available: <https://issues.chromium.org/issues/40917853#comment26>^o