Certificate Transparency Deployment Study

Theodor Nolte

«Hauptprojekt»
12. Dezember 2017

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Inhaltsverzeichnis

1 Einleitung

Die durch Transport Layer Security (TLS) geschützte Kommunikation basiert auf digitalen Zertifikaten, wleche von Zertifizierungsstellen (Certificate Authorities, kurz CAs) ausgestellt werden. In der Vergangenheit ist es mehrfach vorgekommen, daß Zertifikate an Dritte ausgestellt worden sind, die für diese Zertifikate nicht hätten autorisiert werden dürfen.

Problematisch an solch fälschlich ausgestellten Zertifikaten im gegenwärtigen TLS ist zudem, daß diese zu lange unentdeckt bleiben und CAs Vorfälle verschleiern können. Weiter kann der Mißbrauch von fälschlich ausgestellten Zertifikaten unentdeckt bleiben.

Der wohl spektakulärste Vorfall dieser Art war der Einbruch, der im zweiten Quartal 2011 stattfand, bei der niederländischen DigiNotar CA. DigiNotar CA war eine Zertifizierungsstelle für Webserver-Zertifikate, für sogenannte qualifizierte Zertifikate, die als digitaler Unterschriftersatz dienen, sowie für digitale Zertifikate, die in niederländischen Behörden verwendet worden sind [28]. Dem Angreifer ist es gelungen, sich administrativen Zugriff auf alle Zertifikatsserver der Firma zu verschaffen. Dabei ist es ihnen gelungen, mindestens 531 Webserverzertifikate unter anderem für populäre Domains wie `google.com`, `microsoft.com` oder `skype.com` auszustellen [14]. Davon wurde ein Wildcardzertifikat für `*.google.com` großflächig hauptsächlich im Iran für Man-in-the-Middle-Angriffe verwendet. Es wurde von der Bürgerrechtsorganisation Electronic Frontier Foundation der Verdacht geäußert, daß diese Angriffe auf die iranische Regierung zurückzuführen sind [32]. Zwischen dem ersten von dem Angreifer ausgestellten Zertifikat (am 10. Juli 2011), dem Bekanntwerden des Einbruchs [15] und dem Entziehen des Vertrauens in den Browsern Ende August bzw. Anfang September 2011 [4, 16] vergingen ca. fünf Wochen. Da nun DigiNotar keine produktiv nutzbaren Zertifikate mehr ausstellen konnte, weil der Firma das Vertrauen entzogen worden ist und ihr seitens der niederländischen Aufsichtsbehörde für Telekommunikation das Ausstellen qualifizierter Zertifikate untersagt wurde, ging DigiNotar schließlich in die Insolvenz [5].

Es handelt sich in diesen Fällen von falsch ausgestellten Zertifikaten um fehlerhaftes Verhalten von Zertifizierungsstellen. Damit ist der Kern von TLS betroffen. Denn TLS basiert auf dem Vertrauen darin, daß die Zertifizierungsstellen korrekt arbeiten, sie also in den von ihnen in Form von ausgestellten Zertifikaten getätigten Identitätsbehauptungen zuvor die Identitäten korrekt festgestellt haben.

Die Grundidee von Certificate Transparency (CT) ist es, dem Vertrauensmodell von TLS ein weiteres Vertrauens-Element hinzuzufügen, dem Vertrauen in die Öffentlichkeit. Alle nach CT ausgestellten Zertifikate werden öffentlich gemacht und können von jedem, also den Zertifikatsinhabern, Zertifizierungsstellen, Anwendern und Dritten eingesehen werden und somit nachvollzogen werden.

Der diesjährige Transparenzbericht von Google [11] führt auf, daß rund drei Viertel aller Webseiten-Aufrufe HTTPS verschlüsselt erfolgen. Ab April 2018 wird CT für alle mit HTTPS verschlüsselte Webseiten erforderlich sein, um als sicher angezeigt zu werden [30]. Diese Arbeit untersucht die derzeitige Unterstützung von CT bei den Webservern. Es wird also das

Deployment von CT untersucht.

Im Grundlagen-Kapitel 2 wird die Funktionsweise der Web-Public-Key-Infrastruktur (Web-PKI) und von Certificate Transparency erläutert. In Kapitel 3, wird zunächst die Methodologie der Deployment-Untersuchung von Certificate Transparency erläutert. Anschließend werden in diesem Kapitel die Implementierung, die Durchführung und die Ergebnisse der Deployment-Untersuchung besprochen. In Kapitel 4 wird diese Ausarbeitung mit einer Zusammenfassung und einem Ausblick abgeschlossen.

2 Grundlagen

2.1 Die Public-Key-Infrastruktur für TLS-Zertifikate

2.1.1 Transport Layer Security (TLS)

Der Standard Transport Layer Security (TLS), vormals Secure Socket Layer (SSL) benannt, ist im RFC-5246 definiert [6]. Es handelt sich um ein hybrides Verschlüsselungsprotokoll und wird ursprünglich von HTTPS [27] zur sicheren, verschlüsselten Übertragung von Webseiten verwendet. Mit hybrid ist gemeint, daß am Anfang einer TLS-Session, nach dem SSL-Handshake-Protokoll, mittels asymmetrischer Verschlüsselung ein beiden Kommunikationspartnern gemeinsamer Sitzungsschlüssel ausgetauscht wird. Die weitere verschlüsselte Kommunikation ist symmetrisch und basiert auf dem gemeinsamen, ausgehandelten Schlüssel. (Vergleiche auch Schritte i bis vi in Abbildung 1.)

Der für die asymmetrische Verschlüsselung verwendete öffentliche Schlüssel ist im Serverzertifikat enthalten. Der dazugehörige private, nicht-öffentliche Schlüssel bildet den Gegenpart.

Einen guten Überblick über die Prozesse, Komponenten und Bestandteile von TLS-Zertifikaten gibt Schmeh [31]. Im Fachbuch ‘Cryptography Decrypted’ [20] wird besonders gut der Zweck von TLS-Zertifikaten erläutert.

2.1.2 Das TLS Certificate Authority System

In Abbildung 1 sind die Komponenten und Schritte dargestellt, die für die grundsätzliche Einrichtung und das Herstellen einer gesicherten TLS-Verbindung benötigt werden.

Zunächst benötigt der Webserver ein von einer Zertifizierungsstelle ausgestelltes Zertifikat. Dies geschieht durch folgende Schritte:

1. Der Domaininhaber des Webservers (hier verinfacht der Webserver selber) erstellt einen Zertifikatsantrag (`openssl req -new -out csr.pem`) und übermittelt ihn der Registrierungsstelle (Registration Authority, kurz: RA). Dabei wird ein Schlüsselpaar mit öffentlichen und privaten Schlüssel erstellt. Der Antrag enthält den öffentlichen zu signierenden Schlüssel.

2. Der Teilnahmeservicemitarbeiter der Zertifizierungsstelle überprüft, ob der Antragsteller authorisiert ist sowie die Korrektheit der angegebenen Daten im Antrag. Im einfachsten Fall wird authorisiert, indem sie auf eine Bestätigungs-E-Mail, die an die Domain, für die das Webserver-Zertifikat zu erstellen ist, antworten muss (Domain-Validierung, DV). Aufwendigere Prüfungen sehen eine Identifikation mittels Ausweises vor.

3. Nach erfolgreicher Prüfung, d.h. der Antragsteller von der RA autorisiert worden ist, ein Zertifikat zu erhalten, übermittelt die RA an die Zertifizierungsstelle (Certificate Authority, kurz: CA) den Auftrag, ein entsprechendes Zertifikat zu erstellen (Erstellorder).

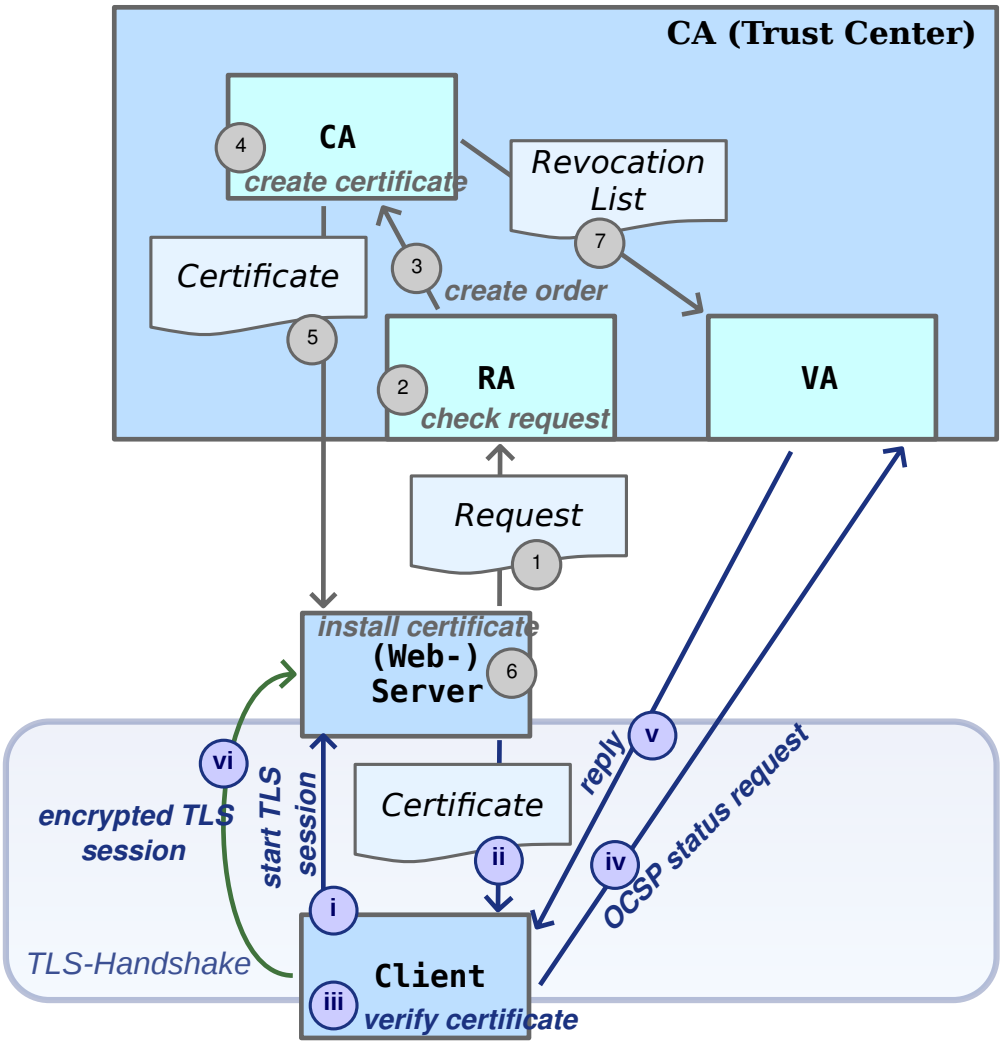

Abbildung 1: Komponenten einer Certificate Authority (CA) sowie benötigte Schritte für eine verschlüsselte TLS-Sitzung zwischen zwei Kommunikationspartnern (hier einem Webserver und einem Client, etwa einem Webbrowser). Schritte 1 bis 6: Erstellung eines Zertifikats für den Webserver. Schritte i bis vi: Der Webserver authentifiziert den Client

4. Das Zertifikat wird erzeugt. Hierbei wird der öffentliche Schlüssel des Antrages mit dem privaten Schlüssel der CA signiert.
5. Das erzeugte Zertifikat wird zum Webserver übertragen (über den Antragsteller, oder beispielweise bei Letsencrypt automatisiert).
6. Das Zertifikat wird nun auf dem Webserver installiert.
7. Die CA erstellt Sperrlisten von Zertifikaten die für ungültig erklärt worden sind. Diese Liste wird regelmäßig aktualisert und der Validation Authority (VA) übergeben, welche Dritten (z.B. einem Webbrowser) Auskunft erteilt, ob ein bestimmtes Zertifkat gültig ist oder nicht. Solch eine Abfrage geschieht üblicherweise mittels OCSP.

Nun hat der Webserver ein gültiges Zertifikat, daß ihre Identität ausweist und bei HTTPS-Seitenabrufen ausliefert wird. Die folgenden Schritte laufen bei jeder erfolgreichen TLS-Sitzung von neuem ab:

i. Der Client (Webbrowser) initiiert eine TLS-Session
ii. Neben der Cipher-Suite, d.h. den Informationen, welche kryptografischen Verfahren und Versionen vom Webserver unterstützt werden, übermittelt der Webserver das Zertifikat samt Zertifikatkette.
iii. Der Client überprüft das Zertifikat, indem es die Zertifikatkette bis zur Root-CA, die auch in der Liste der Root-CAs seiner Browsersoftware enthalten ist, abgleicht.
iv. Als nächstes stellt der Client eine Zertifikatsperrabfrage an die VA der PKI, die das Zertifikat ausgestellt hat
v. Die VA antwortet, daß das Zertifikat nicht gesperrt ist. Diese Antwort ist mit dem privaten Schlüssel der CA signiert um ihre Echtheit nachzuweisen.
vi. Der Client hat nun den Webserver erfolgreich authentifiziert, beide können nun einen gmeinsamen Sitzungsschlüssel vereinbaren und mit diesem gesichert kommunizieren; der Browser ruft die Webseite also verschlüsselt auf.

Im Englischen wird die CA-Komponente sowie die Zertifizierungsstelle mit Certificate Authority (CA) benannt. Nachfolgend ist mit CA die Zertifizierungsstelle in ihrer Gesamtheit gemeint, wenn nicht explizit anders beschrieben.

2.2 Certificate Transparency

2.2.1 Fachliche Motivation für Certificate Transparency

Die TLS-PKI in ihrer gegenwärtigen Struktur basiert auf dem Vertrauen in die CAs, daß beim Ausstellen von Zertifikaten keine Fehler gemacht werden. Es gibt jedoch keinen organisatorischen oder technischen Mechanismus, für jedes Zertifikat zu überprüfen, ob es korrekt oder aber unzulässig von einer CA ausgestellt worden ist.

Genau hier setzt Certificate Transparency (CT) an. Das Prinzip von CT ist es, jedes ausgestellte Zertifikat zu veröffentlichen und somit zu ermöglichen, das Ausstellen aller Zertifikate für jeden nachzuvollziehbar und überprüfbar zu machen. Insbesondere kann ein Domain-Inhaber alle ausgestellten Zertifikate über seine Domain ermitteln und mögliche nicht durch ihn initiierte Zertifikate entdecken. Ebenfalls können CAs alle in ihrem Namen ausgestellten Zertifikate überwachen.

Um das Veröffentlichen zu ermöglichen wird die TLS-PKI erweitert durch sogenannte Logs. Jedes (dem CT-Standard folgend) ausgestellte Zertifikat wird in einem oder mehreren solcher Logs veröffentlicht. Die Veröffentlichung eines Zertifikats in einem Log ist öffentlich nachprüfbar. Zugangsbeschränkungen auf Logs sind explizit nicht vorgesehen; jedem TLS-Kommunikationspartner sowie Dritten ist es möglich, Anfragen an Logs zu stellen.

Die technische Beschaffenheit eines CT-Logs verhindert, bereits veröffentlichte Zertfikate aus dem Log zu entfernen oder durch ein anderes Zertifikat auszustauschen. Der Sicherheitsgewinn dieser sogenannten *append only* Eigenschaft für die TLS-PKI erfolgt mittelbar, unzulässig ausgestellte Zertifikate werden durch die Veröffentlichung nicht direkt verhindert. Doch es ist davon auszugehen, daß ein unzulässig ausgestelltes Zertifikat beim Überprüfen der Logs durch den Domaininhaber, der jeweiligen CA oder durch Dritte entdeckt wird.

CT ist hierbei eine technische Erweiterung der von TLS verwendeten Public Key Infrastruktur (TLS-PKI). Dabei werden keine Bestandteile der gegenwärtigen TLS-PKI ersetzt, noch stellt es eine weitere CA dar (die etwa jedes neu auszustellende Zertifikat wiederum signiert).

Die Zielsetzung von CT ist es also, die Verwendung von irrtümlich oder böswillig ausgestellten, fehlerhaften Zertifikaten maßgeblich zu erschweren. Durch die Veröffentlichung von Zertifikaten in Logs wird erreicht:

1. Eine CA kann kein Zertifikat für eine Domain ausstellen, ohne daß der Domaininhaber davon erfährt bzw. ohne großen Aufwand davon erfahren kann.

2. Sowohl die CAs als auch Domaininhaber und Dritte können feststellen, ob Zertifikate irrtümlich oder böswillig ausgestellt worden sind.

3. Anwender können überprüfen, ob das Zertifikat (einer Domain) veröffentlicht worden ist. Dabei kann angenommen werden, daß aufgrund der Veröffentlichung das Zertifikat auf seine Korrektheit hin sowohl von der ausstellenden CA als auch vom Domaininhaber überprüft worden ist.

2.2.2 Funktionsweise von Certificate Transparency

Die TLS-PKI (ohne CT) setzt sich zusammen, aus: * CA * dem Subjekt (z.B. der Webserver) * dem Anwender (Client, etwa der Webbrowser)

CT bringt das *Log* als eine weitere Komponente hinzu. Und es gibt es in CT die zwei Rollen *Monitor* und *Auditor*. Das Log und die beiden Rollen sind in Abbildung 2 aufgezeigt.

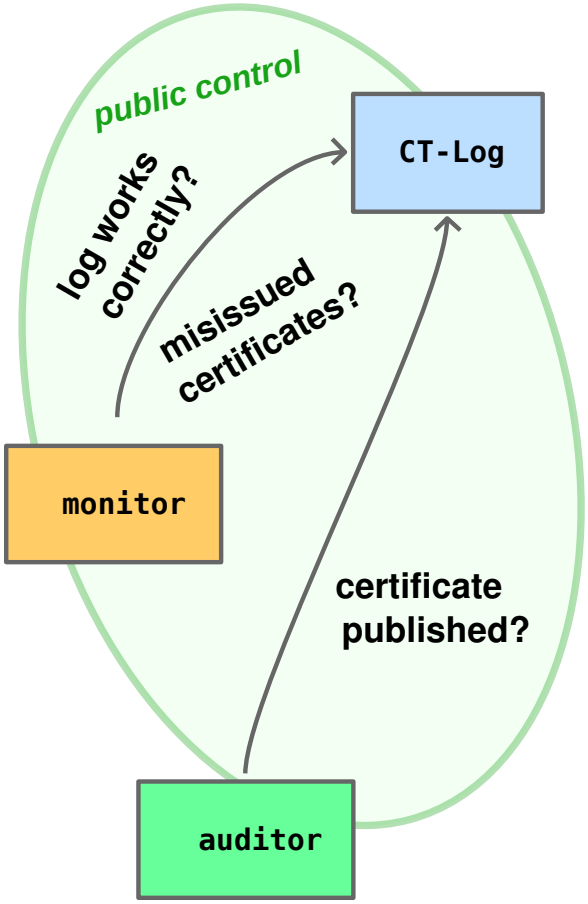

Abbildung 2: Komponenten und Rollen von Certificate Transparency

Im Log werden Zertifikate veröffentlicht. Die Veröffentlichungen sind zeitlich geordnet und neue Zertifikate werden ausschließlich hinten angefügt. Vorhandene Einträge sind nicht mehr änderbar (*append only* Eigenschaft). In regelmäßigen Zeitabständen, etwa einmal pro Stunde, wird eine neue Version vom Log veröffentlicht, d.h. es werden neu angenommene Zertifikate öffentlich einsehbar gemacht. Wird ein Zertifikat zur Veröffentlichung von einem Log angenom-

men, so wird als Bestätigung ein sogenannter *signed certificate timestamp* (SCT) zurückgegeben. Der SCT ist sozusagen eine vom Log signierte Annahmequittung zu einem Zertifikat. Über diesen SCT kann das Zertifikat im Log nachgeschlagen werden. Das Veröffentlichen selber erfolgt anonym; jeder kann beliebige gültige Zertifikate an ein Log zur Veröffentlichung senden. Als Schutz vor Zertifikats-Spam werden aber ausschließlich Zertifikate angenommen, die in der Chain-of-Trust der TLS-PKI eingebunden sind.

Ein Monitor überwacht die Integrität von einem Log. Insbesondere wird die *append only* Eigenschaft von einem Log überprüft. Hierzu vergleicht der Monitor die aktuelle Log-Version mit der vorhergehenden Version. Da ein Monitor hierzu eine komplette Sicht auf das Log hat, kann er auch von CAs und Domain-Inhabern dazu verwendet werden, um veröffentliche Zertifikate zu erkennen, die von einer anderen CA ausgestellt worden sind, aber nicht hätten ausgestellt werden dürfen.

Der SCT ist eindeutig dem Zertifikat und dem Log zuordenbar. Der Auditor prüft nun, ob ein konkretes Zertifikat anhand des dazugehörigen SCTs im Log veröffentlicht worden ist, indem es Konsistenzüberprüfungen über die Enthaltenheit des Zertifikats im Log durchführt.

Während ein CT-Log eine eigenständige Komponente darstellt und als ein eigenständiger Server betrieben wird, ist dies beim Monitor und Auditor nicht zwingend notwendig. So ist im von CT-Projekt vorgeschlagenen Setup vorgesehen, daß der Monitor von der CA betrieben wird, also in die dort bestehende Server-Infrastruktur eingegliedert ist. Der Auditor, so wird vorgeschlagen, soll im Browser des TLS-Clients integriert sein. Dies stellt jedoch ein Datenschuztproblem dar, da das Log anhand der Anfragen des Clients Einsicht über den Verlauf des Webbrowsers erhält.

2.2.3 Das Log

Proofs – Das Log aus Anwendersicht In Abbildung 3 ist eine Gesamtübersicht aufgezeigt. Die grünen Verbindungen des Monitors mit der CA und dem Webserver sowie der grüne Doppelpfeil zwischen dem Auditor und dem Browser spiegelt das Interesse, welche Instanz (CA, Webserver oder Webbrowser) welche Rolle (Monitor oder Auditor) in CT einnimmt.

Ein Auditor überprüft, ob ein Zertifikat korrekt Veröffentlicht worden ist. Es wird sozusagen die Frage gestellt: “certificate published?”. Als Antwort erhält es vom Log ein sogenanntes *merkle audit proof*, einen kryptografischen Beweis, daß das Zertifikat in diesem Log veröffentlicht ist. Dies spiegelt sich beim Webseitenabruf wieder, ob dem verwendeten Zertifikat das Vertrauen in die Öffentlichkeit gegeben werden kann.

Der Monitor stellt zunächst die Frage: “log works correctly?”, ob das Log integer ist und ob insbesondere die Append-Only-Eigenschaft erfüllt ist. Dies kann der Monitor prüfen, indem er das Log mittels sogenannter *merkle consistency proofs* über unterschiedliche Log-Versionen diesen über die Zeit überwacht. Zugleich bekommt ein Monitor eine Gesamtsicht vom Log und kennt somit alle im Log veröffentlichten Zertifikate und in den Zertifikaten eingearbeitete Domains, was dem Monitor ermöglicht, falsch ausgestellte Zertifikate zu erkennen (“misissued

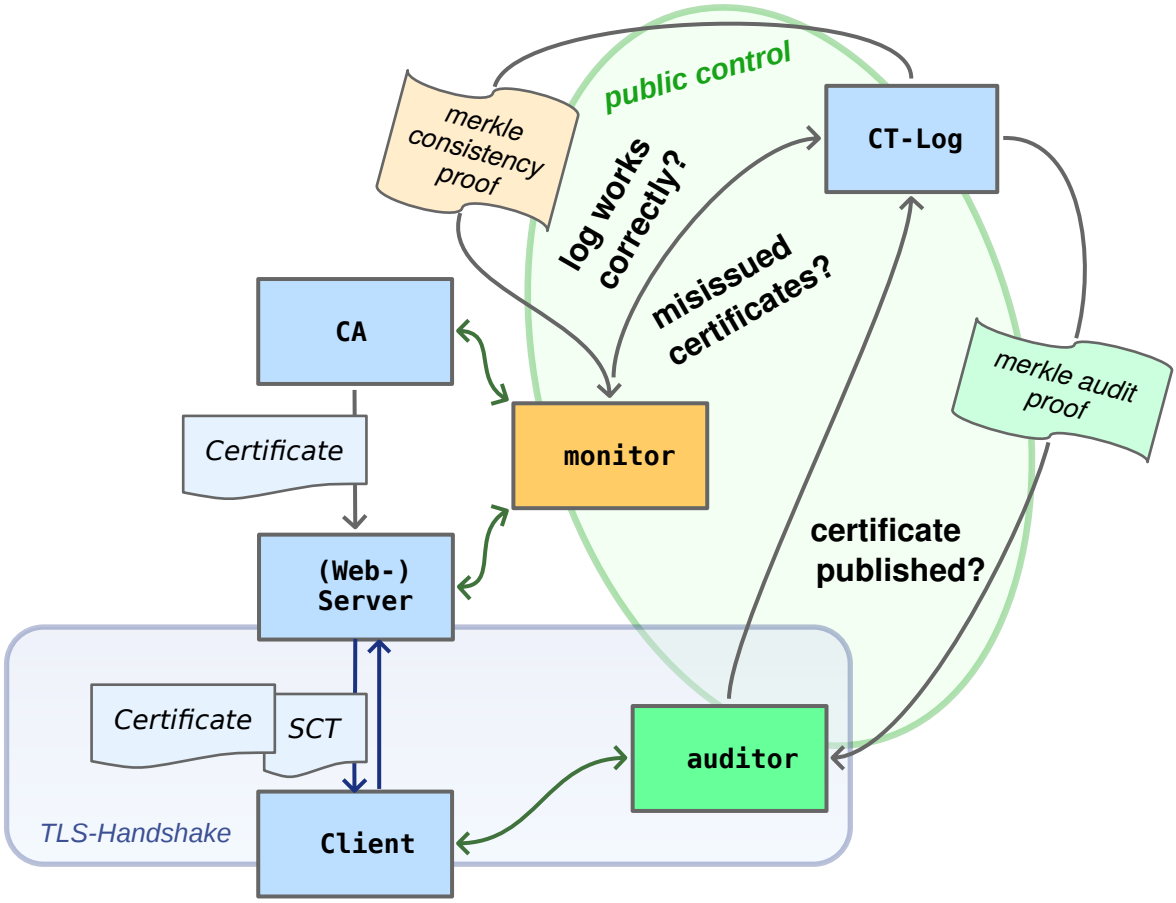

Abbildung 3: Komponenten von Certificate Transparency

certificates?”).

Zu jedem von einem Log aufgenommenen Zertifikat gibt es einen entsprechenden Eintrag. Dieser Eintrag umfasst neben dem Zertifikat selber bzw. dem Pre-Certificate auch die Zertifikatskette, die notwendig ist, um das Zertifikat gegenüber der TLS-Chain-of-Trust zu verifizieren. Nicht enthalten in diesem Eintrag ist jedoch der Zeitstempel der Annahme bzw. der komplette SCT. Diese Informationen werden separat gespeichert.

Intern ist das CT-Log als ein Binärbaum von aufeinander aufbauenden Hash-Knoten dem Merkle Hash Tree (auch kurz Merkle Tree benannt) aufgebaut, welcher Hashes dieser Zertifikat-Einträge als Blätter hat.

Merkle Hash Tree Der einfachste Ansatz Zertifikate in einem Log zu veröffentlichen wäre es, einen Hash über eine Liste von Zertifikaten zu bilden und anschließend diesen Hash zu signieren. Um die Veröffentlichung eines Zertifikats durch einen Client zu überprüfen, müssten vom Log zunächst sämtliche anderen Zertifikate and den Client übermittelt werden, damit dieser den Hash selber berechnen kann. Anschließend kann der Client diesen mit dem veröffentlichten Hash vergleichen. Würde das Zertifikat vom veröffentlichten abweichen hätte dies einen anderen Hash zur Folge. Der Rechenaufwand (und Übertragungsumfang) steigt linear zum Umfang der Liste und gehört der Komplexitätsklasse `O(n)` an.

Um diesen Rechenaufwand zu mindern wenden die CT-Logs einen “Trick” an. Und zwar wird zunächst von allen Einträgen jeweils ein Hash gebildete. Je zwei dieser Hashes bilden einen weiteren Hash. Hiervon werden wiederum je zwei gehasht usw. bis am Ende ein Hash von Hashes gebildet wird, der den Hash aller Einträge darstellt. Um diesen Gesamt-Hash zu einem Eintrag berechnen zu können, müssen nicht sämtliche andere Einträge vorliegen. Eine Teilmenge der Hash-Hashes reicht aus, wie wir im folgenden Kapitel noch sehen werden. Dies entspricht der Komplexitätsklasse `O(log_2(n))`.

Der Merkle Hash Tree ist ein binärer Baum, welcher gemäß RFC6962 [17] nach folgenden Regeln zusammengesetzt ist:

Gegeben sei eine Liste von `n` Einträgen:

```
C[n] := {c0, c1, ..., cn-1}.
```

Der Hash einer leeren Liste ist der Hash von einem leeren Byte-String:

```
MTH({}) := HASH()
```

Der Hash einer Liste, die genau einen Eintrag enthält, d.h. ein Blatt vom Binärbaum, ist folgendermaßen definiert:

```
MTH({c0}) := HASH(0x00 || c0)
```

Die internen Knoten des Binärbaums sind rekursiv definiert. Für `m > 1` sei `k` die größte Potenz von `2`, die kleiner als `m` ist (es gilt somit: `k < m <= 2k`):

```
MTH(C[m]) := HASH(0x01 || MTH(C[0:k]) || MTH(C[k:m]))
```

Hierbei sei `C[i:j]` eine Liste von `j-i` Einträgen `ci` bis `cj-1`:

```
C[i:j] := {ci, ci+1, ..., cj-1}
```

`HASH` ist ein Hashing-Algorithmus dessen Länge der Ausgabe in Bytes immer `HASH_SIZE` groß ist. Derzeit wird für CT ausschließlich der Hashing-Algorithmus SHA-256 [1] verwendet, somit hat `HASH_SIZE` den Wert `256/8 = 32`. Der Hash der leeren Liste ist demnach: `MTH({}) = HASH() =` `e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855`.

Der Knoten `MTH(C[n])` über alle Einträge, also die Wurzel des Binärbaums, wird als Tree Head bezeichnet.

In periodischen Zeitabständen, dem Maximum-Merge-Delay (MMD), wird der Merkle Tree basierend auf allen bisherigen und neu aufgenommenen Zertifikat-Einträgen neu gebildet. Der neue Tree Head wird nun mit dem Zertifikat des Logs signiert und veröffentlicht. Dieser signierte Tree Head wird als Signed Tree Head (STH) bezeichnet.

Merkle Audit Proof Beim *merkle audit proof* wird geprüft, ob ein bestimmtes Zertifikat in dem Merkle-Baum korrekt enthalten ist. Hierzu antwortet das Log dem Auditor auf die Enthaltenheitsanfrage (*"certificate published?"*) mit einem *merkle audit proof* (siehe ebenfalls Abbildung 3).

Technisch wird über den Merkle Hash Tree ein Enthaltenheitsbeweis (ein *inclusion proof*) zu einem Eintrag erbracht. Das *inclusion proof* `PATH(m, C[n])` ist eine Liste von Knoten des Merkle Trees, die benötigt werden, um zu dem Eintrag `m` aus der Menge aller Einträge `C[n]` eines Logs den *tree hash*, der vom Log als STH veröffentlicht ist, berechnen zu können.

Das *inclusion proof* ist folgendermaßen definiert: Gibt es genau einen Eintrag (`C[1] = {c0}`), so ist das *inclusion proof* über den Binär-Baum, der nur das einzelne Blatt enthält, leer:

```
PATH(0, {c0}) = ()
```

Bei mehr als einem Eintrag (`n > 1`) ist das *inclusion proof* rekursiv definiert: Sei `k` die größte Potenz von `2`, die kleiner als `n` ist (es gilt: `k < n <= 2k`). Das *inclusion proof* für das `m+1`-te Element `cm` einer Liste von `n` Elementen (`m < n`) lautet:

```
m < k:

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

m => k:

    PATH(m, C[n]) := PATH(m - k, C[k:n]) : MTH(C[0:k])
```

Der Doppelpunkt-Operator : Bedeutet das Anfügen des Elements `element` an die Liste `list`: `appended_list = list : element`.

Beim in Abbildung 4 dargestellten Merkle Tree lässt sich exemplarisch folgendes *inclusion proof* für den Eintrag `c2` nachvollziehen:

```
PATH(2, C[8])
= PATH(2, C[0:4]) : MTH(C[4:8])
= PATH(0, C[2:4]) : MTH(C[0:2]) : MTH(C[4:8])
= PATH(0, C[2:3]) : MTH(C[3:4]) : MTH(C[0:2]) : MTH(C[4:8])
= () : MTH(C[3:4]) : MTH(C[0:2]) : MTH(C[4:8])
= [MTH(C[0:2]), MTH(C[3:4]), MTH(C[4:8])]
= [i, d, n]
```

Konsistenzüberprüfung eines Logs Beim *merkle consistency proof* wird die *append only* Eigenschaft des Logs nachvollzogen. Die *append only* Eigenschaft besagt ja, daß neue Einträge im Log nur hinten angefügt werden können. Im Umkehrschluss bedeutet dies, daß alle Einträge einer vorigen Log-Version in einer späteren Log-Version an vorderster Stelle enthalten sind. Hierzu werden zwei Versionen vom Log miteinander verglichen. Alle Einträge des älteren Logs müssen in der neuen Version vom Log in genau gleicher Reihenfolge und vor den neu hinzugekommenen Einträgen enthalten sein.

Version `p` sei eine ältere, bereits überprüfte Version über `m` Einträge (Tree Head: `MTH(C[0:m])`). Version `q` ist die neue Version über `n` Einträge (`m <= n`; Tree Head: `MTH(C[n])`). Nun soll geprüft werden, ob Version `q` gegenüber Version `p` konsistent ist. Beide Versionen müssen in ihren ersten `m` Einträgen identisch sein, genau dies ist nun zu prüfen. Hierzu benötigen wir eine (möglichst minimale) Menge an Knoten in der neuen Version, um sowohl die Wurzel der Version `q` als auch der Version `p` berechnen zu können. Stimmen die Berechnungen mit den veröffentlichten STHs beider Versionen jeweils überein, so ist die neue Log-Version `q` konsistent gegenüber der älteren Version `p`.

Das *consistency proof* ist für eine Liste von `n` Einträgen `C[n] = {c0, c1, ..., cn-1}` eines Logs der Version `q` gegenüber einer vorigen Version `p` über `m` Einträge mit dem Tree Head `MTH(C[0:m])` folgendermaßen definiert:

```
PROOF(m, C[n]) := SUBPROOF(m, C[n], true)
```

Für `SUBPROOF` gilt:

```
m = n,  m ist der Wert, mit dem PROOF aufgerufen wurde:

    SUBPROOF(m, D[m], true) := ()

m = n, sonst:
```

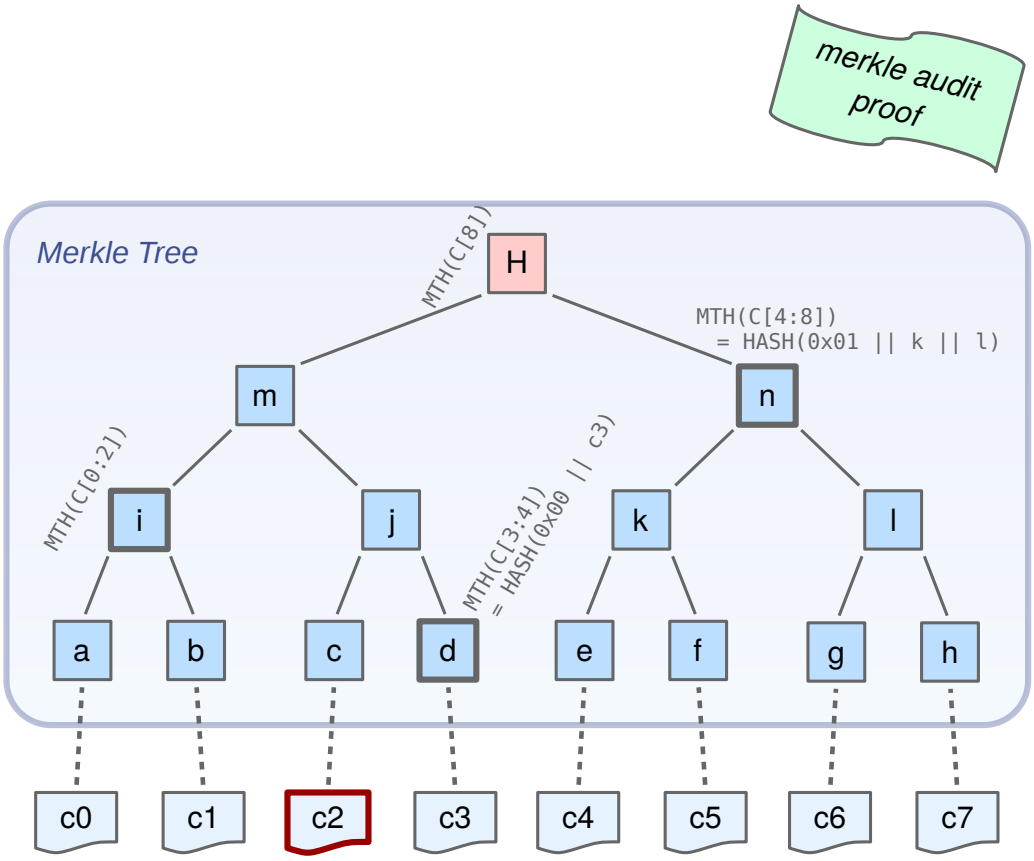

Abbildung 4: merkle audit proof: `PATH(c2, C[8]) = [i, d, n]`. D.h. mit den Knoten `i, d` und `n` kann zusammen mit `c2` der Tree Head `H` berechnet werden und mit dem vom Log veröffentlichten STH verglichen werden

```
    SUBPROOF(m, C[m], false) := {MTH(C[m])}
```

Für `m < n` gilt für `SUBPROOF` eine rekursive Definition:
Sei `k` die größte Potenz von `2`, die kleiner als `n` ist (es gilt: `k < n <= 2k`).

```
m < n  and  m <= k:

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

m < n  and  m > k:

    SUBPROOF(m, C[n], b) := SUBPROOF(m - k, C[k:n], false) : MTH(C[0:k])
```

Mit den in Abbildung 5 dargestellten Merkle Trees der Log-Versionen p und q lässt sich exemplarisch folgendes *consistency proof* nachvollziehen:

```
PROOF(6, C[8]) = SUBPROOF(6, C[0:8], True)
= SUBPROOF(2, C[4:8], False) : MTH(C[0:4])
= SUBPROOF(2, C[4:6], False) : MTH(C[6:8]) : MTH(C[0:4])
= {MTH(C[4:6])} : MTH(C[6:8]) : MTH(C[0:4])
= [MTH(C[0:4]) : MTH(C[4:6]) : MTH(C[6:8])]
= [m, k, l]
```

Log-API Für die Kommunikation mit einem Log stehen für Log-Clients wie einem Monitor, einem Auditor oder um Zertifikate zu registrieren als Schnittstelle *HTTPS POST* und *GET requests* zur Verfügung.

Zum Veröffentlichen eines Zertifikats wird ein JSON-Objekt übergeben, in welchem das zu veröffentlichende Webserver-Zertifikat samt seiner Zertifikatskette bis zum Wurzelzertifikat enthalten sein muss. (Das Wurzelzertifikat selber kann auch weggelassen werden.) Zurückgegeben wird in einem JSON-Objekt neben Metadaten der SCT, der die Aufnahme bestätigt und zugleich die Zusage darstellt, daß das Zertifikat innerhalb des MMD veröffentlicht sein wird:

```
POST https://<log server>/ct/v1/add-chain
POST https://<log server>/ct/v1/add-pre-chain  # for pre-certificates

Input:

{
    "chain": "<base64-encoded certificates>"
}
```

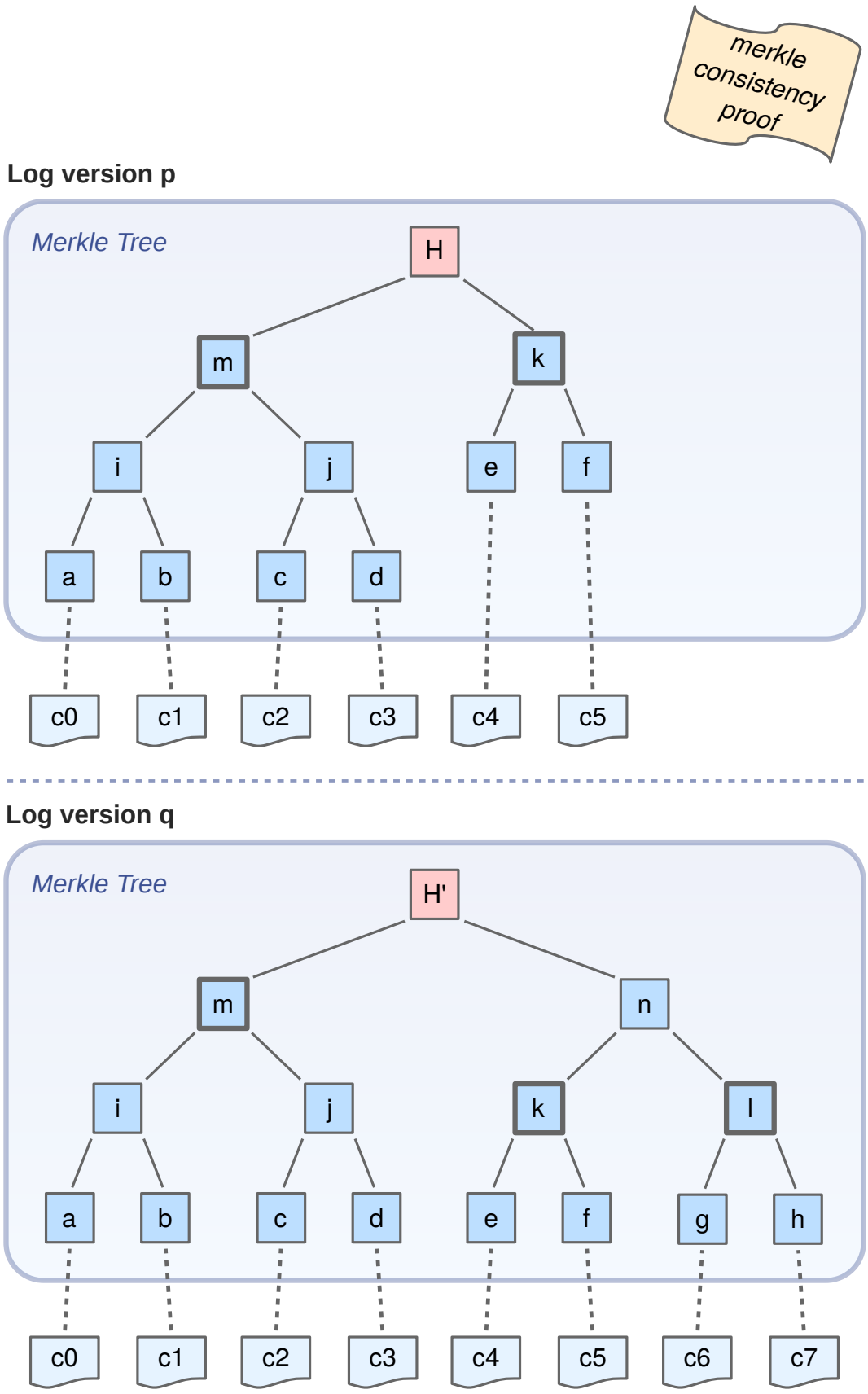

Abbildung 5: merkle consistency proof: `PROOF(6, C[8]) = [m, k, l]`. D.h. mit den Knoten `m`, `k` und `l` kann sowohl der Tree Head `H` der Log-Version `p` als auch der Tree Head `H'` der neueren Log-Version `q` berechnet werden

```
Output:

{
    "sct_version": "v1",
    "id": "<id of the log>",
    "timestamp": "<the SCT>",
    "extensions": "<for future use>",
    "signature": "<base64-encoded signature of the SCT>"
}
```

Der STH geht wird folgendem *request* abgerufen:

```
GET https://<log server>/ct/v1/get-sth
```

Beispielsweise wurde für den Kommandozeilen-Aufruf
`curl https://ct.googleapis.com/aviator/ct/v1/get-sth` am 23.12.2015 um 14:58 Uhr folgendes JSON-Objekt zurückgegeben:

```
{
    "tree_size": 10725363,
    "timestamp": 1450875348467,
    "sha256_root_hash": "ilUk72kUo3gwv3K0Vq8oxDiCiRPuLuarigZRmy2pmBA=",
    "tree_head_signature": "BAMARjBEAiAJ6a9TaVO1PGEdy8pRkr1JGypNY3GidLDb
              NcT9ApNWkAIgaHTSf0kZhtWzhyUpZ3sZVVnfmB+4EElEAv/oO/DW+3U="
}
```

Ein *consistency proof* wird durch folgenden Aufruf zurückgegeben:

```
GET https://<log server>/ct/v1/get-sth-consistency

Input:

{
    "first": <tree size in decimal>,
    "second": <tree size in decimal>
}

Output:

{
    "consistency": [<array of merkle tree nodes, base64-encoded>]
}
```

Ein *audit proof* liefert folgender *request*:

```
GET https://<log server>/ct/v1/get-proof-by-hash

Input:

{
    "hash": "<base-64 encoded leaf hash of the certificate to audit>",
    "tree_size": <tree_size in decimal>
}

Output:

{
    "leaf_index": <0-based index of the hash entry>,
    "audit_path": [<array of base64-encoded merkle tree nodes>]
}
```

Weiter sind noch *requests* spezifiziert, um Einträge, akzeptierte Root-Zertifikate und zu Debugging-Zwecken einen Eintrag in Kombination mit dem dazugehörigen *audit proof* zurückzugeben.

2.2.4 Zertifikate Veröffentlichen

Nachfolgend werden die unterschiedlichen Vorgehensweisen betrachtet, um ein Zertifikat in einem Log zu veröffentlichen. Anschließend wird dargestellt, wie eine TLS-Verbindung mit Unterstützung von CT zustande kommt.

Wenn ein Zertifikat von einem Log zur Veröffentlichung angenommen wird, gibt es als Bestätigung den *signed certificate timestamp* (SCT) zurück. Dieser dient als Verweis auf den Eintrag zum Zertifikat im Log und wird beim TLS-Handshake vom Webserver neben dem Zertifikat (samt der Zertifikatskette) dem Client übergeben. So kann der Client die Veröffentlichung nachprüfen.

Beim Entwurf von CT wurden praktische Aspekte des Deployments berücksichtigt. So gibt es in CT unterschiedliche Vorgehensweisen, ein Zertifikat zu veröffentlichen. Die Veröffentlichung selber kann sowohl von der ausstellenden CA oder dem Webserver (oder Dritten) durchgeführt werden. Je nach Vorgehensweise sind die an der Austellung beteiligten Parteien (CA und Webserver) an der Veröffentlichung unterschiedlich beteiligt. Auch im nachfolgenden Gebrauch des Zertifikats bei TLS-Sessions wird der SCT auf unterschiedliche Weisen dem TLS-Client übermittelt.

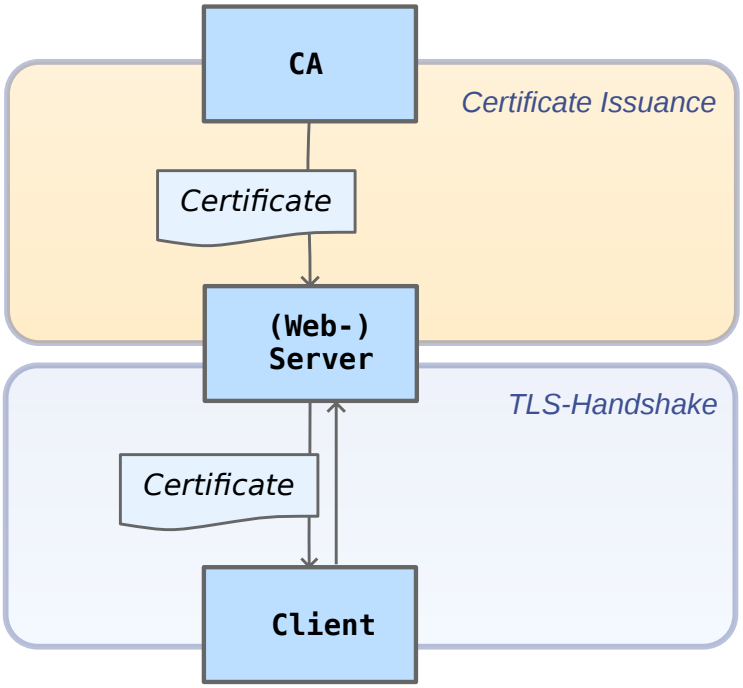

Abbildung 6: Server-Zertifikat ausstellen sowie verwenden (TLS-Handshake), derzeitiger TLS-Standard; ohne SCT

Zertifikatsauslieferung ohne Veröffentlichung In Abbildung 6 ist das bisherige Vorgehen zum Ausstellen eines Zertfikats innerhalb der Web-PKI dargestellt. Die CA erzeugt ein Zertifikat und liefert es an den Webserver aus. Dieser wiederum authentisiert sich im Zuge von TLS-Sessions beim Client mit diesem Zertifikat (samt Zertifikatskette).

Es gibt keinen CT-Log, Zertifikate werden nicht veröffentlicht. TLS-Server liefern keinen SCT aus und TLS-Clients überprüfen keine SCTs.

Verweis auf Logeintrag in das Zertifikat integrieren In Abbildung 7 ist die Variante aufgezeigt, bei der der SCT in das Zertifikat selber eingearbeitet wird. Die ausstellende CA muss ihre Abläufe beim Ausstellen des Zertifikats dahingegehend anpassen, daß sie, bevor das eigentliche Zertifikat erzeugt wird, ein sogenanntes *pre-certificate* an das Log zur Veröffentlichung übergibt. In dem *pre-certificate* sind bis auf dem SCT bereits alle Bestandteile des auszustellenden Zertifikats enthalten. Die Annahmebestätigung vom Log, den SCT, arbeitet sie anschließend in das Zertifikat selber sein, indem der SCT als X.509v3-Extension mit der OID 1.3.6.1.4.1.11129.2.4.2 eingetragen wird (Schritte 1 und 2 in Abbildung 7).

In dieser Variante unterstützt der Webserver CT automatisch: Der nun für einen CT-TLS-Handshake benötigte SCT ist untrennbar im Zertifikat eingearbeitet und wird so dem Client beim TLS-Handshake übergeben.

Interessant ist hierbei der Aspekt, daß für Serverzertifikate mit SCT auch gegenüber Clients, die CT nicht implementieren, die Vertrauenswürdigkeit um das Vertrauen in die Öffentlichkeit aufgrund der erfolgten Veröffentlichung bestärkt wird.

Webserver liefert Verweis auf Logeintrag per TLS-Erweiterung separat aus In Abbildung 8 ist dargestellt, wie es dem Domaininhaber ermöglicht ist, sein Webserverzertifikat zu veröffentlichen, wenn die CA ihre Prozesse für CT nicht anpasst und keine Veröffentlichung unternimmt. Nachdem das Zertifikat ausgestellt worden ist (Schritt 1), wird es vom Webserver (bzw. dem Domaininhaber) im CT-Log registriert (Schritt 2), welcher wiederum mit dem SCT die Aufnahme im Log bestätigt (Schritt 3). Nun wird der Webserver so konfiguriert, daß er beim TLS-Handshake neben dem Zertifikat (samt Kette) per TLS-Extension auch den SCT dem Client übergibt.

Dieses Verfahren ist für eine Übergangsphase nützlich, wenn es noch CAs gibt, die ihre Prozesse noch nicht angepasst haben, der Kunde jedoch veröffentlichte Webserver-Zertifikate verwenden will. Auch bereits ausgestellte Zertifikate können nachträglich veröffentlicht werden.

Bemerkenswert ist hierbei, daß die Registrierung von (gültigen) Zertifikaten ohne Authentifizierung erfolgt. Damit ein Zertifikat vom Log angenommen wird, muss es lediglich von einer CA ausgestellt worden sein, die in der Kette zu einem Root-Zertifikat enthalten ist. Wer das Zertifikat an das Log übermittelt, ist egal. So gibt es offenbar Crawler-Skripte, die Zertifikate von Webseiten in die bisher betriebenen (Test-) Logs eintragen.

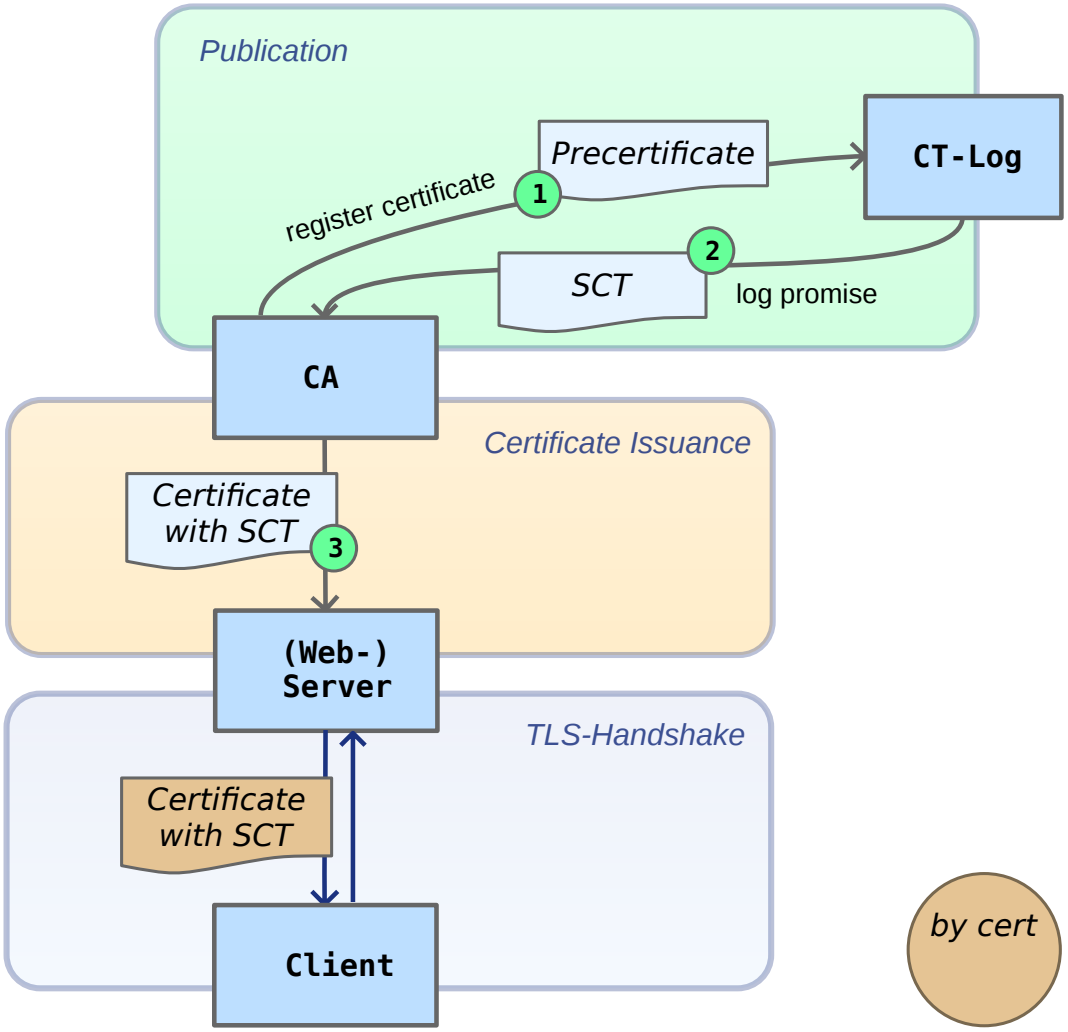

Abbildung 7: Server-Zertifikat veröffentlichen, ausstellen sowie verwenden; SCT im Zertifikat enthalten (per X.509v3-Erweiterung)

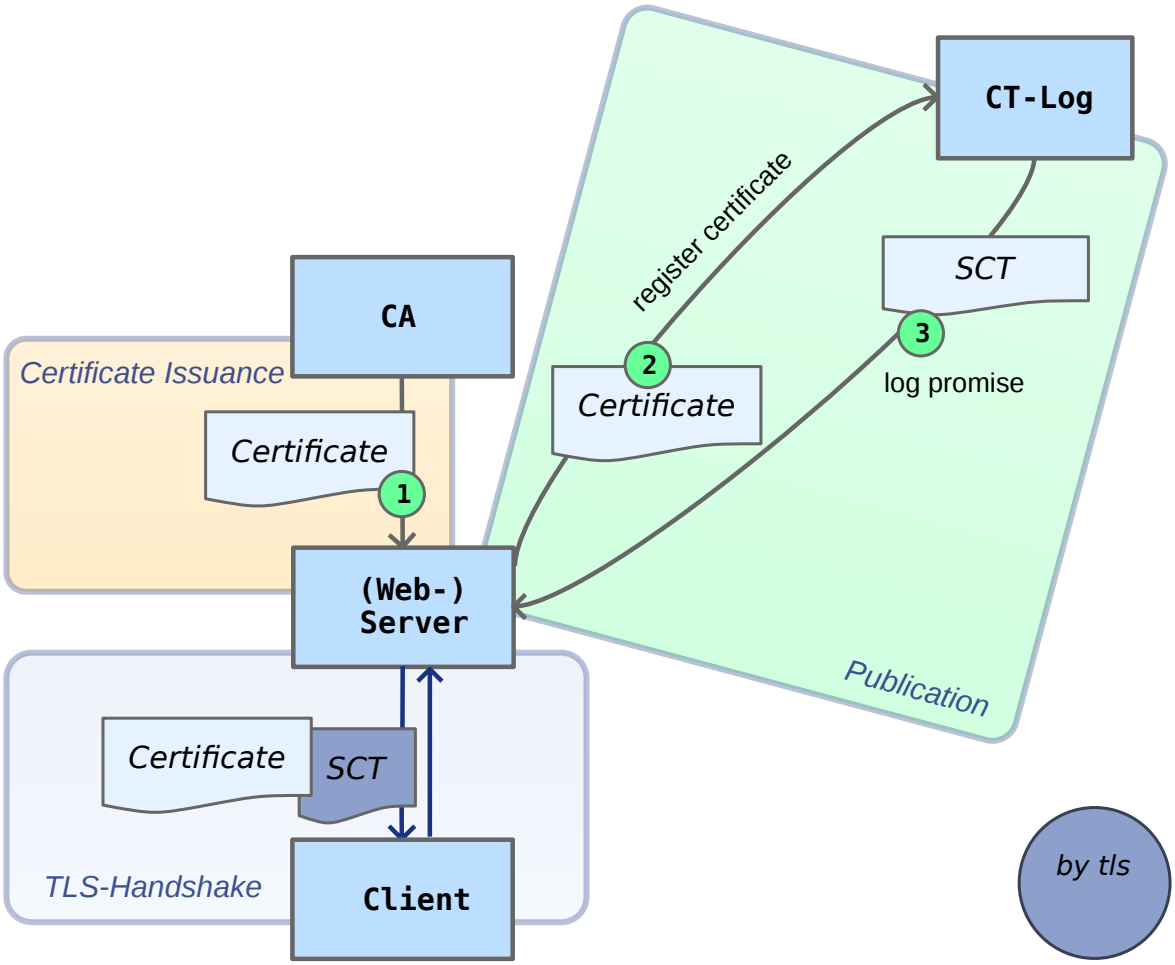

Abbildung 8: Server-Zertifikat ausstellen und veröffentlichen sowie verwenden; SCT wird dem Client (Webbrowser) per TLS-Erweiterung parallel zum Zertifikat übermittelt

Verweis auf Logeintrag mittels OCSP übergeben Auch in der in Abbildung 9 dargestellten Variante wird der SCT beim TLS-Handshake per TLS-Extension übergeben. Sowohl die CA als auch der Webserver muss seine bisherigen Abläufe erweitern. Das Zertifikat wird von der ausstellenden CA an die Domain ausgestellt und im gleichen Zuge beim CT-Log registriert. Das Log bestätigt die Aufnahme gegenüber der CA mit dem SCT. Nun erhält der Webserver den SCT, indem er an die CA eine entsprechende OCSP-Anfrage stellt und mit der OCSP-Response den SCT bekommt (Schritte 2 und 3 [orange]).

Diese Variante bietet sich besonders an, wenn die Zertifikataustellung automatisiert erfolgen soll. OCSP wurde aus dem pragmatischen Grund gewählt, daß CAs sowieso schon einen OCSP-Service für die Sperrabfragen betreiben (Quasi-Standard).

2.2.5 Zertifikate in Chrome

Derzeit unterstützen CT nur der Browser Chromium und der darauf basierende Google-Browser Chrome [33]. Hierbei nimmt der Browser weder die Rolle eines Monitors noch die eines Auditors ein, es findet keine Interaktionen mit CT-Logs statt. Stattdessen werden die beim TLS-Handshake übertragenen SCTs verifiziert. Hierzu hat der Browser eine Liste der öffentlichen Schlüssel von akzeptierten CT-Logs.

Die Chromium CT-Policy [8] legt fest, wie viele CT-Logs zu einem Webserver-Zertifikat benötigt werden und welche Anforderungen CT-Logs erfüllen müssen, um von Chromium akzeptiert zu werden. Beispielsweise müssen von Chromium akzeptierte CT-Logs hochverfügbar sein. Nach den Regeln muss ein Webserver zusammen mit dem HTTPS-Zertifikat mindestens zwei SCTs ausliefern, die von mindestens zwei unterschiedlichen CT-Log-Betreibern ausgestellt worden sind.

Seit Anfang 2015 müssen Extended-Validation-Zertifikate (EV-Zertifikate) CT unterstützen [12]. Ein EV-Zertifikat, welches CT nicht unterstützt, wird im Chromium-Browser als nicht sicher angezeigt. Zunächst war von Google angekündigt worden, daß für sämtliche Webserver-Zertifikate eine CT-Unterstützung ab Oktober 2017 erforderlich sein wird [29]. Die Einführung ist zwischenzeitlich aufgeschoben worden [30]. Nun wird CT für sämtliche Webserver-Zertifikate ab April unterstützt werden müssen, um von Chromium als vertrauenswürdig akzeptiert zu werden.

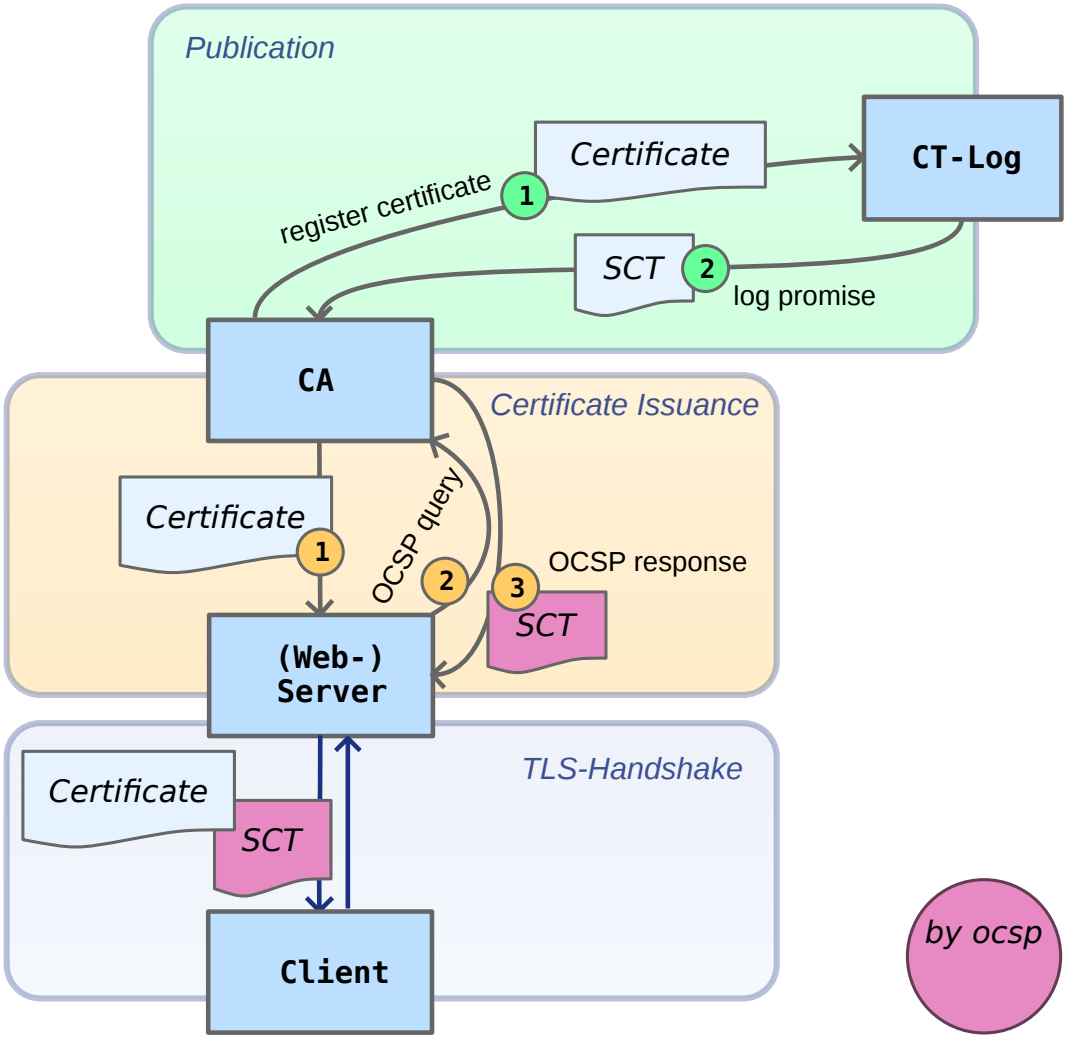

Abbildung 9: Server-Zertifikat mit SCT ausstellen und veröffentlichen sowie verwenden; SCT per OCSP-Stapling

3 Untersuchung der Verbreitung von Certificate Transparency

3.1 Methodologie

Es wird untersucht, ob bei mit TLS verschlüsselten Webseitenaufrufen Certificate Transparency unterstützt wird.

Bei der Erhebung mit den Webseitenaufrufen werden drei Schritte vorgenommen: (1) Auswahl der Domain-Namen der Webseiten vornehmen; (2) Speichern der beim Handshake verwendeten TLS-Webserver-Zertifikate; (3) Ermitteln und Speichern der beim Handshake verwendeten SCTs.

(1) Domain-Namen auswählen. Um die Verbreitung von CT bei verschlüsselten Webseiten zu ermitteln, bedarf es zunächst einer Beispiel-Menge von Domain-Namen von Webseiten. Hierfür wird die Alexa-Liste verwendet. Sie enthält, absteigend sortiert nach ihrer Popularität, die 1 Millionen Domain-Namen der meistbesuchten Websites. Es ist somit möglich, die Verbreitung von CT mit der Popularität von Webseiten in Zusammenhang zu stellen. Auf die Alexa-Liste wird auch in anderen Web-Erhebungen zurückgegriffen, beispielsweise in [44]. Dort wird auch auf weitere Arbeiten verwiesen, die die Alexa-Liste verwenden.

(2) TLS-Zertifikate abrufen. Es werden die Domain-Namen auf TLS-Zertifikate abgebildet. Als Zwischenschritt werden zunächst die Domains auf IP-Adressen per DNS-Resolver-Abfrage abgebildet. Es wird hier nicht weiter berücksichtigt, daß die IP-Adresse zu einer Domain je nach gefragtem DNS-Server sich unterscheiden kann. Auch wird nicht darauf eingegangen, ob in in Content Delivery Networks (CDNs) gehosteten Webseiten unterschiedliche Zertifikate verwendet werden. Beides bleibt offen für spätere Analysen.

Je Eintrag aus der Alexa-Liste wird als Domain-Name der Eintrag selber verwendet sowie der Eintrag versehen mit dem Präfix *'www.'*. Um die von den Webservern verwendeten TLS-Zertifikate zu erhalten, werden die Webseiten per HTTPS abgerufen. Sofern die aufgerufene Webseite Verschlüsselung unterstützt, wird das Zertifikat beim TLS-Handshake übermittelt.

(3) SCTs abrufen. SCTs können beim TLS-Handshake über drei Mechanismen übermittelt werden (“by-Cert”, “by-TLS-Extension”, “by-OCSP-Response”). Für jeden TLS-Handshake werden alle drei Mechanismen angewendet. So werden alle zu einem Zertifikat bereitgestellten SCTs erfasst.

3.2 Implementierung

Für die Analyse wurden vom Autor zwei Softwarekomponenten entwickelt: ctutlz [36] und ctstats. ctstats führt die Erhebungen aus und erstellt die Auswertung. ctutlz wird von ctstats als Modul verwendet und ist für den TLS-Handshake und das Erfassen der dabei übermittelten SCTs zuständig.

Beide Komponenten sind in Python geschrieben. Bei Python handelt es sich um eine Skript-Sprache, in der Programmieraufgaben relativ einfach umgesetzt werden können. Python unterstützt das Schreiben von wartbarem Code. So wird durch die syntaktische Besonderheit,

daß die Programmstruktur durch Einrückungen festgelegt wird statt durch Klammern, ein aufgeräumtes Quelltext-Bild erzwungen. Leitmotive der Sprache sind geprägt von Lesbarkeit, Einfachheit, dem expliziten Ausschreiben von Zusammenhängen im Quelltext, und durch Konventionen, welche den vorigen Punkten genügen [43]. Für die Umsetzung der Aufgabenstellung steht mit pyOpenSSL ein umfangreicher, gut gewarteter OpenSSL-Wrapper zur Verfügung [26]. Für die Darstellung der Auswertungen bietet sich matplotlib an [34]. Nicht zuletzt wurde Python gewählt, da der Autor mit Python gut vertraut ist.

Die Implementierung des Programmcodes zum Erfassen der SCTs hat sich als ein schwieriges und komplexes Unterfangen mit vielen Fehlversuchen herausgestellt. So haben folgende Ansätze nicht geklappt:

- Das Modul ssl aus der Standard-Python Bibliothek [35] unterstützt keine OCSP-Abfragen, noch CT.
- m2crypto, eine umfangreiche Python-Crypto-Bibliothek [19], unterstützt kein CT.
- pyOpenSSL [26], noch vor m2crypto der umfangreichste OpenSSL-Wrapper in Python, unterstützt CT nur in Ansätzen. Das gilt auch für cryptography [25], dem Python-Crypto-Modul, auf das pyOpenSSL für die OpenSSL-Einbindung zurückgreift. Das Erfassen der SCTs beim TLS-Handshake wird von beiden Bibliotheken nicht unterstützt. Auch das Verifizieren der SCTs wird in pyOpenSSL nicht unterstützt.
- OpenSSL [21] (Github-Repository: [22]) unterstützt CT und hierbei das Ermitteln der SCTs auf allen drei möglichen Wegen (by-Cert, by-TLS, by-OCSP). Jedoch hat sich eine Implementierung, die OpenSSL in einem eigenen Prozess per externen Kommandoaufruf aus dem Python-Code heraus aufruft, als viel zu unperformant herausgestellt. Hinzu kommt, daß mehrere Kommando-Aufrufe von `openssl s_client` pro URL notwendig sind, um die SCTs auf allen drei möglichen Wegen zu erhalten.

3.2.1 ctutlz: SCTs von Webservern holen

Das Python-Package `ctutlz` [36] stellt das Skript `verify-sct` bereit, um die von einem Webserver zu einem Zertifikat bereitgestellten SCTs auf allen drei möglichen Wegen zu holen und diese anschließend zu verifizieren. Diese Funktionalität kann auch als Biliotheksfunktion in einem Python Programm verwendet werden. Für den TLS-Handshake wird die C-API von OpenSSL direkt verwendet. Es handelt sich um die erste Implementierung dieser Art, und es werden keine Subprozesse gestartet, die OpenSSL als Kommandozeilen-Tool ausführen. `ctutlz` ist OpenSource, gehostet auf github [36] und umfasst inklusive Tests rund 4300 Zeilen Code.

`ctutlz` verwendet OpenSSL, indem es den Wrapper pyOpenSSL sowie cryptography einbindet. Das C-Binding in Python, also das Zugreifen auf die C-Schnittstelle von OpenSSL aus

Python-Code heraus, bewerkstelligt cryptography mit dem Python-Package cffi [24]. Beim TLS-Handshake mit pyOpenSSL müssen im OpenSSL-Context-Objekt, das die TLS-Verbindung repräsentiert, Callback-Funktionen registriert werden, um die SCTs zu holen. Genau diese Funktionalität wird von pyOpenSSL jedoch nicht unterstützt. In ctultz sind nun entsprechende Callback-Funktionen definiert, die mit cffi nun "selber" über die C-API von OpenSSL im mit pyOpenSSL erzeugten OpenSSL-Context-Objekt registriert werden [38].

Beim Verifizieren der SCTs wird ähnlich vorgegangen. Auch hier fehlt es an Funktionalität in pyOpenSSL. So unterstützt das beim Verifizieren benötigte `pkey`-Objekt keine elliptischen Kurven. Diese sind jedoch bei den Zertifikaten der CT-Logs üblich. `ctutlz` löst dies, indem die Funktion `pkey_from_cryptography_key()` von OpenSSL durch eine Version ersetzt wird, die elliptische Kurven unterstützt [37]. Dieses Vorgehen wird Duck-Typing genannt [45]. Für diese Funktionserweiterung hat der Autor einen Pull-Request für pyOpenSSL gestellt [41].

Für das Verifizieren der SCTs muss eine Liste von CT-Logs vorhanden sein. Mit Hilfe des Skripts `ctloglist` in `ctutlz` hat der Autor auf der veröffentlichten CT-Log-Liste [10] Fehler entdeckt und über das CT-Forum reportet [13].

Beim Herunterladen und Verifizieren der SCTs muss mit unterschiedlichen Encodings und Datenstrukturen umgegangen werden: ASN.1, DER, B64, PEM und "TLS Data Format" (TDF) [6].

Für das Verstehen der Verifikation von SCTs hat der Blog-Eintrag von Pierky [23] dem Autor sehr geholfen. Das Einarbeiten und Verständnis für pyOpenSSL, cryptography und cffi wurde vom Autor durch die Software-Projekte pyopenssl-examples [42] festgehalten. Für die Einarbeitung in die C-API von OpenSSL wurde das Software-Projekt openssl-examples [39] erstellt. Dort ist auch ein Aufruf enthalten in OpenSSL verständlichen Beispiel-Code unter Test einzuführen [40].

3.2.2 ctstats: Auswertung

Das Python-Package `ctstats` führt die Erhebung aus, indem es über die Alexa-Top1M-Liste iteriert und für jeden TLS-Handshake das Ergebnis in eine Datenbank schreibt. Nachdem die Erhebung abgeschlossen ist, wird durch ein Skript (enthalten in `ctstats`) eine Analyse der Daten durchgeführt; es werden Tabellen, Graphen und Diagramme erstellt. `ctstats` umfasst über 7800 Codezeilen und muss als Prototyp bezeichnet werden. Vor einer Veröffentlichung bedarf es einer umfassenden Überarbeitung.

Der Ablauf der Erhebung beginnt mit dem Holen der aktuellen Alexa-Liste. Die Funktionalität hierfür wird durch die API von `ctutlz` bereit gestellt. Zu jedem Domain-Eintrag wird zusätzlich ein Eintrag mit `www.`-Präfix angelegt. Für jeden Eintrag wird versucht ein TLS-Handshake durchzuführen. Wurden dabei SCTs geholt, so werden diese umgehend verifiziert, d.h. es wird geprüft, ob die Signatur des SCTs zum Zertifikat des entsprechenden CT-Logs passt. Auch die Funktionalität der Durchführung des TLS-Handshakes und der Verifikation der SCTs wird von

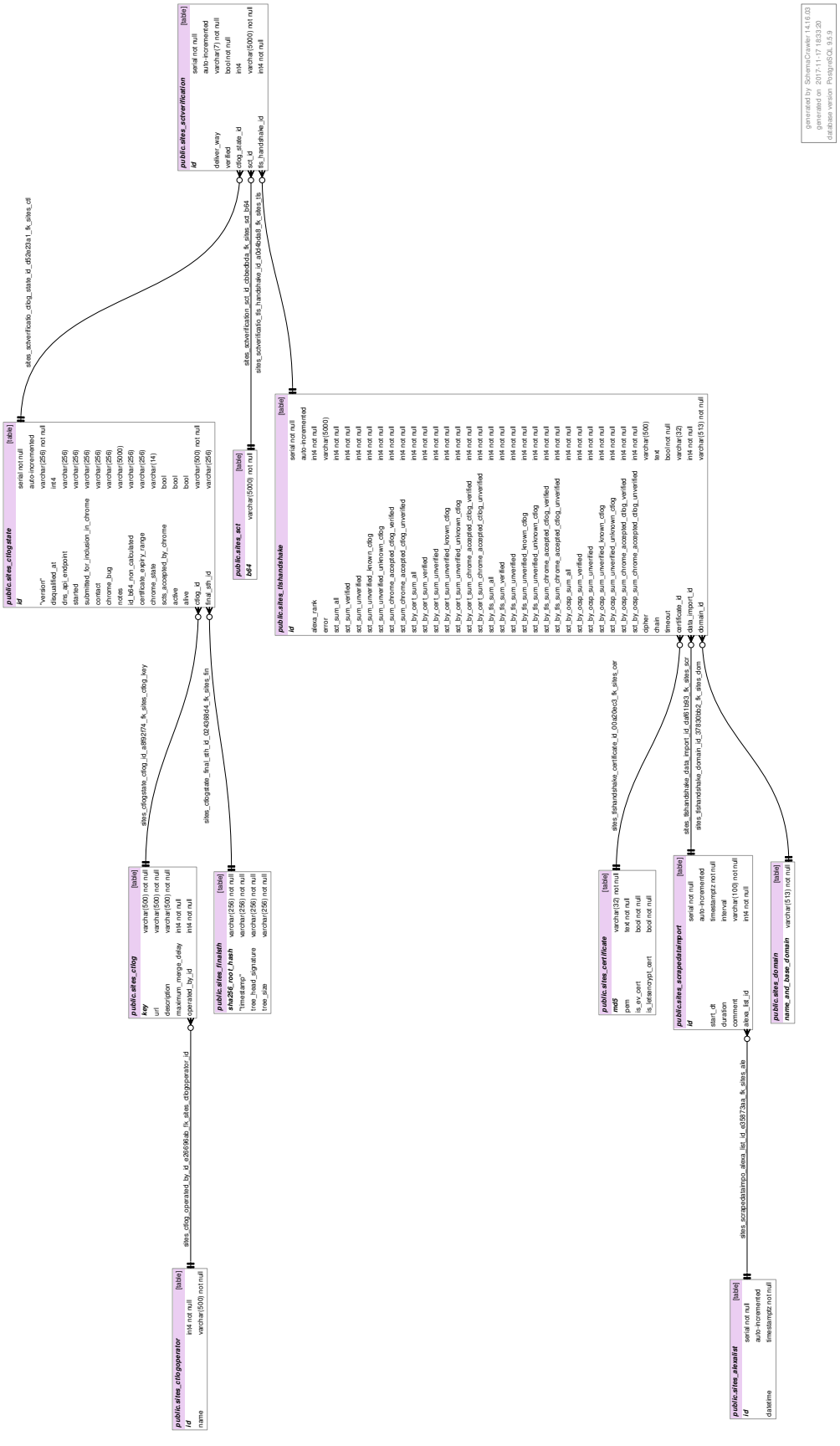

Abbildung 10: Datenbank-Schema der Tabelle ‘ctstats’

public.sites_view_tlshandshake	[view]
tlshandshake_id	int4
domain_name	text
base_domain	text
alexa_rank	int4
data_import_id	int4
data_import_start_dt	timestamptz
data_import_duration	interval
data_import_comment	varchar(100)
alexalist_id	int4
alexalist_datetime	timestamptz
error	varchar(5000)
certificate_md5	varchar(32)
certificate_pem	text
is_ev_cert	bool
is_letsencrypt_cert	bool
sct_sum_all	int4
sct_sum_verified	int4
sct_sum_unverified	int4
sct_sum_unverified_known_ctlog	int4
sct_sum_unverified_unknown_ctlog	int4
sct_sum_chrome_accepted_ctlog_verified	int4
sct_sum_chrome_accepted_ctlog_unverified	int4
sct_by_cert_sum_all	int4
sct_by_cert_sum_verified	int4
sct_by_cert_sum_unverified	int4
sct_by_cert_sum_unverified_known_ctlog	int4
sct_by_cert_sum_unverified_unknown_ctlog	int4
sct_by_cert_sum_chrome_accepted_ctlog_verified	int4
sct_by_cert_sum_chrome_accepted_ctlog_unverified	int4
sct_by_tls_sum_all	int4
sct_by_tls_sum_verified	int4
sct_by_tls_sum_unverified	int4
sct_by_tls_sum_unverified_known_ctlog	int4
sct_by_tls_sum_unverified_unknown_ctlog	int4
sct_by_tls_sum_chrome_accepted_ctlog_verified	int4
sct_by_tls_sum_chrome_accepted_ctlog_unverified	int4
sct_by_ocsp_sum_all	int4
sct_by_ocsp_sum_verified	int4
sct_by_ocsp_sum_unverified	int4
sct_by_ocsp_sum_unverified_known_ctlog	int4
sct_by_ocsp_sum_unverified_unknown_ctlog	int4
sct_by_ocsp_sum_chrome_accepted_ctlog_verified	int4
sct_by_ocsp_sum_chrome_accepted_ctlog_unverified	int4
cipher	varchar(500)
chain	text
timeout	bool

generated by	SchemaCrawler 14.16.03
generated on	2017-11-17 19:14:22
database version	PostgreSQL 9.5.9

Abbildung 11: Datenbank-View ‘tlshandshake’

public.sites_view_ctlogstate	[view]
ctlogstate_id	int4
"version"	varchar(256)
key	varchar(500)
url	varchar(500)
description	varchar(500)
maximum_merge_delay	int4
operator_id	int4
operator_name	varchar(500)
final_sth_sha256_root_hash	varchar(256)
final_sth_timestamp	varchar(256)
final_sth_tree_head_signature	varchar(256)
final_sth_tree_size	varchar(256)
disqualified_at	int4
dns_api_endpoint	varchar(256)
started_at	varchar(256)
submitted_for_inclusion_in_chrome_at	varchar(256)
contact	varchar(256)
chrome_bug	varchar(256)
notes	varchar(5000)
id_b64_non_calculated	varchar(256)
certificate_expiry_range	varchar(256)
chrome_state	varchar(14)
scts_accepted_by_chrome	bool
active	bool
alive	bool

generated by	SchemaCrawler 14.16.03
generated on	2017-11-17 19:15:59
database version	PostgreSQL 9.5.9

Abbildung 12: Datenbank-View ‘ctlogstate’

public.sites_view_sctverification	[view]
sctverification_id	int4
tlshandshake_id	int4
domain_name	text
base_domain	text
alexa_rank	int4
data_import_id	int4
data_import_start_dt	timestamptz
data_import_duration	interval
data_import_comment	varchar(100)
alexalist_id	int4
alexalist_datetime	timestamptz
tlshandshake_error	varchar(5000)
certificate_md5	varchar(32)
certificate_pem	text
is_ev_cert	bool
is_letsencrypt_cert	bool
sct_sum_all	int4
sct_sum_verified	int4
sct_sum_unverified	int4
sct_sum_unverified_known_ctlog	int4
sct_sum_unverified_unknown_ctlog	int4
sct_sum_chrome_accepted_ctlog_verified	int4
sct_sum_chrome_accepted_ctlog_unverified	int4
sct_by_cert_sum_all	int4
sct_by_cert_sum_verified	int4
sct_by_cert_sum_unverified	int4
sct_by_cert_sum_unverified_known_ctlog	int4
sct_by_cert_sum_unverified_unknown_ctlog	int4
sct_by_cert_sum_chrome_accepted_ctlog_verified	int4
sct_by_cert_sum_chrome_accepted_ctlog_unverified	int4
sct_by_tls_sum_all	int4
sct_by_tls_sum_verified	int4
sct_by_tls_sum_unverified	int4
sct_by_tls_sum_unverified_known_ctlog	int4
sct_by_tls_sum_unverified_unknown_ctlog	int4
sct_by_tls_sum_chrome_accepted_ctlog_verified	int4
sct_by_tls_sum_chrome_accepted_ctlog_unverified	int4
sct_by_ocsp_sum_all	int4
sct_by_ocsp_sum_verified	int4
sct_by_ocsp_sum_unverified	int4
sct_by_ocsp_sum_unverified_known_ctlog	int4
sct_by_ocsp_sum_unverified_unknown_ctlog	int4
sct_by_ocsp_sum_chrome_accepted_ctlog_verified	int4
sct_by_ocsp_sum_chrome_accepted_ctlog_unverified	int4
cipher	varchar(500)
chain	text
timeout	bool
deliver_way	varchar(7)
sct_b64	varchar(5000)
ctlogstate_id	int4
ctlog_version	varchar(256)
ctlog_key	varchar(500)
ctlog_url	varchar(500)
ctlog_description	varchar(500)
ctlog_maximum_merge_delay	int4
ctlog_operator_id	int4
ctlog_operator_name	varchar(500)
ctlog_final_sth_sha256_root_hash	varchar(256)
ctlog_final_sth_timestamp	varchar(256)
ctlog_final_sth_tree_head_signature	varchar(256)
ctlog_final_sth_tree_size	varchar(256)
ctlog_disqualified_at	int4
ctlog_dns_api_endpoint	varchar(256)
ctlog_started_at	varchar(256)
ctlog_submitted_for_inclusion_in_chrome_at	varchar(256)
ctlog_contact	varchar(256)
ctlog_chrome_bug	varchar(256)
ctlog_notes	varchar(5000)
ctlog_id_b64_non_calculated	varchar(256)
ctlog_certificate_expiry_range	varchar(256)
ctlog_chrome_state	varchar(14)
accepted_by_chrome	bool
ctlog_active	bool
ctlog_alive	bool
verified	bool

generated by SchemaCrawler 14.16.03
generated on 2017-11-17 19:15:25
database version PostgreSQL 9.5.9

Abbildung 13: Datenbank-View ‘sctverification’

`ctultz` bereit gestellt. Die Erhobenen Daten vom Handshake und das Ergebnis der Verifikation werden in eine Postgres-Datenbank zur späteren Analyse geschrieben. Abbildung 10 stellt das Schema der Datenbank dar.

Für einen kompletten Durchlauf für eine Erhebung über zwei Millionen TLS-Handshake-Versuche hat `ctstats` auf einem Dual-Core Rechner mit drei Gigabyte RAM und "normaler" Festplatte (keine SSD) rund 33 Stunden und 12 Minuten benötigt. Dabei sind 9273 MB an Daten in die Postgres-Datenbank geschrieben worden.

Die Erhebung wird in einer Pipeline ausgeführt. Für jeweils 10000 Domains werden insgesamt 2000 Thread-Pool-Prozesse ausgeführt, die jeweils 5 Threads für 5 TLS-Handshakes starten. Von diesen 2000 Thread-Pool-Prozessen werden maximal 150 gleichzeitig ausgeführt. sind Thread-Pool-Prozesse fertig und haben geendet, werden entsprechend neue Thread-Pool-Prozesse gestartet, bis alle 10000 TLS-Handshakes ausgeführt worden sind. Hierbei wird neben der maximalen Prozess-Anzahl das Starten weiterer Prozesse zurückgehalten, wenn einen maximale RAM-Ausnutzung von 60.0 Prozent überschritten wird. Das Resultat eines Handshakes wird in eine Queue geschrieben. Sind alle 10000 Domaineinträge abgearbeitet worden, so starten 20 Worker-Threads, die die Daten aus der Queue in die Postgres-Datenbank schreiben. Anschließend startet die nächste Iteration über 10000 Domains, bis alle Domains abgearbeitet worden sind. Die hier aufgeführten Maximalwerte wurden mühsam per Try-and-Error ermittelt und sollten bei einem Rechner mit besserer Austattung vermutlich optimiert werden, um die Dauer der Erhebung zu minimieren.

Bei der Durchführung hat sich der Netzwerkanschluss mit 100 Mbit nicht als limiternder Faktor herausgestellt. Der Flaschenhals lag vielmehr an drei anderen Stellen. Der Arbeitsspeicher limitiert die Anzahl der gleichzeitig ausführbaren Thread-Pool-Prozesse. Wird das RAM-Limit zu hoch angesetzt so kommt es zu Swapping und Fehlerzuständen, die einen Abbruch der Erhebung, Unbedienbarkeit des Systems und einen Neustart des Rechners bedingen. Die Schreibgeschwindigkeit der Festplatte limitiert maßgeblich die Zeit, die für das Wegschreiben der erhobenen Daten in die Datenbank benötigt wird. Werden versucht, mehr Threads zu starten, als durch das Linux-Betriebssystem maximal zulässig (Maximalwert wurde hochgesetzt mit `ulimit -n 4001`), so stürzt der Hauptprozess der Erhebung ab und sie muss neu gestartet werden.

Für das Erstellen der Auswertung werden viele, komplexe SQL-Queries ausgeführt. Um die jeweiligen Queries in ihrer Komplexität zu verringern wurden die Datenbank-Views `tlshandshake` (siehe Abbildung 11), `ctlogstate` (Abbildung 12) und die auf die beiden genannten Views zurückgreifende View `sctverification` (Abbildung 13) definiert.

Viele dieser Queries, etwa zum Normieren von Graphen, müssten für unterschiedliche Analysen mehrfach ausgeführt werden, was viel Zeit kosten würde und insbesondere beim Entwerfen und Programmieren der Abfragen hinderlich wäre. Um die Mehrfachausführung zu verhindern wurde ein File-Caching-Mechanismus programmiert, der den Funktionen, welche die jeweiligen Queries ausführen, per Python-Decorator (ein funktionales Sprach-Feature von Python) zugewiesen wird. Mit einem Cache über die Ergebnisse aller Queries zu einer Daten-

Erhebung benötigt das Skript zum Erzeugen aller Tabllen, Graphen und Diagramme keine 20 Sekunden. Ein initialer Lauf nach einer Erhebung über zwei Millionen Domains benötigt rund 90 Minuten. Für das Erstellen der Graphen und Diagramme wird das Python-Package `matplotlib` [34] verwendet.

Ein weiteres funktionales Sprach-Feature, nämlich lambda-Funktionen, wird sowohl in `ctstats` als auch in `ctutlz` verwendet um Python-Named-Tuples mit `lazy-evaluated-vals` zu erweitern. Der Nutzen liegt darin, daß die Verwendung von Named-Tuples zu besonders übersichtlichem Code verhilft, ohne zusätzliche Performance-Einschränkungen zu erzeugen.

3.3 Ergebnisse

Für die Erhebung wurde die Alexa-Top1M-Liste vom 28.9.2017 verwendet, die Erhebung ist ebenfalls am 28.9.2017 gestartet und hat am 29.9.2017 geendet.

Tabelle 1: TLS Handshake Tries (without www. prefix)

	count	percent
all	1000000	100.00
timeout	118239	11.82
no certificate	220927	22.09
certificate (no ev)	639509	63.95
ev certificate	21325	2.13

Tabelle 2: TLS Handshake Tries (with www. prefix)

	count	percent
all	1000000	100.00
timeout	105629	10.56
no certificate	225386	22.54
certificate (no ev)	644417	64.44
ev certificate	24568	2.46

In den Tabellen 1 und 2 sind die Statistiken der durchgeführten TLS-Handshake-Versuche aufgelistet. Die Unterschiede zwischen Domains mit oder ohne ‘www.’-Präfix sind vernachlässigbar. Auch die nachvolgenden Graphen und Diagramme würden nur vernachlässigbare Unterschiede aufzeigen. Deshalb werden für alle weiteren Betrachtungen immer alle Datensätze, mit und ohne ‘www.’-Präfix, zusammen ausgewertet.

Tabelle 3: TLS Handshake Tries (w/wo www. prefix)

	count	percent
all	2000000	100.00
timeout	223868	11.19
no certificate	446313	22.32
certificate (no ev)	1283926	64.20
ev certificate	45893	2.29

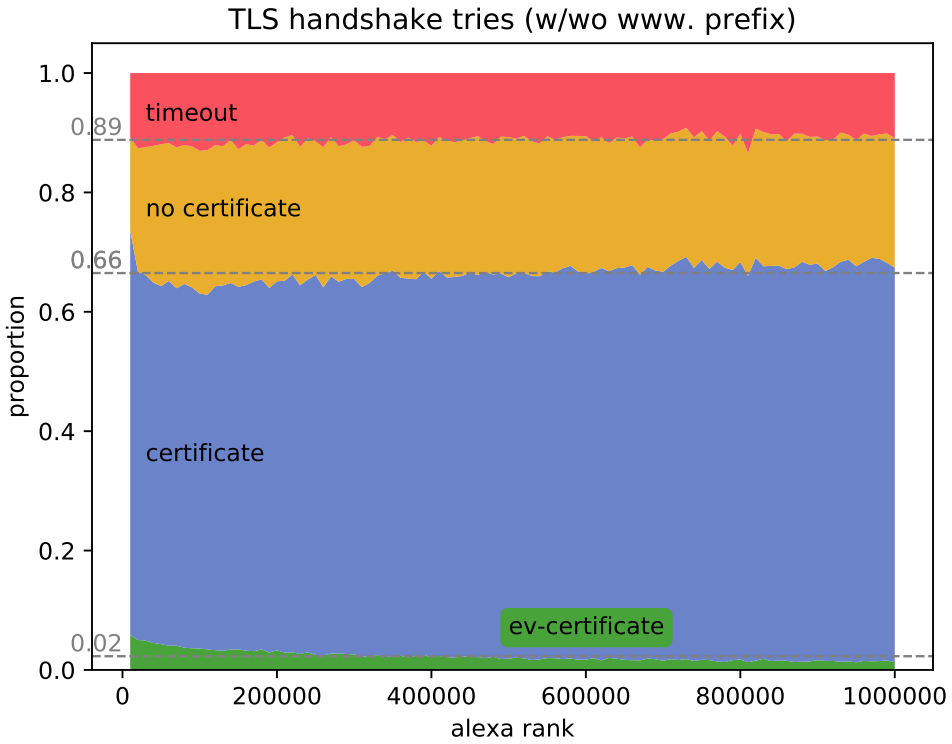

Abbildung 14: TLS-Handshake-Versuche (alexa ranked)

Tabelle 3 führt die Statistik der TLS-Handshake-Versuche über die Domains der Alexa-Top1m-Liste mit und ohne 'www.'-Präfix. Rund 11 Prozent der TLS-Handshake-Versuche wurden abgebrochen. Der Timeout wurde mit 5 Sekunden relativ kurz gewählt. Bei einem längeren Timeout hätte die Erhebung über 2 Millionen TLS-Handshake-Versuche zu lange gedauert. Ebenfalls aus Zeitgründen wurde bei einem Timeout kein zweiter Versuch unternommen. Ab-

bildung 14 zeigt die Verteilung der Ergebnisse der TLS-Handshake-Versuche über den Verlauf der Alexa-Top1m-Liste mit absteigender Popularität auf. Je 20000 Ergebnisse (w/wo-www) über jeweils 10000 Rankings sind hierbei zusammengefasst worden. Gut zu erkennen ist, daß die Timeouts über das ganze Ranking relativ konstant bei 9 Prozent liegen.

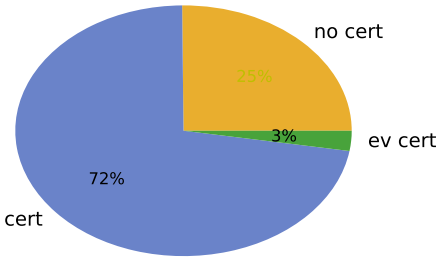

Abbildung 15: TLS Handshakes

Das in Abbildung 15 dargestellte Tortendiagramm zeigt die anteilige Verteilung der TLS-Handshake-Versuche auf, bei denen der Server geantwortet hat. Bei 75 Prozent wurde der TLS-Handshake erfolgreich durchgeführt, die Domains sind per HTTPS erreichbar. Rund 3 Prozent der erreichbaren Domains verfügen über ein EV-Zertifikat. 25 Prozent der erreichbaren Domains sind nur per HTTP zu erreichen. Der Graph in Abbildung 16 zeigt die Ergebnisse der TLS-Handshake-Versuche, bei denen der Server geantwortet hat. Bei den Top-Domains ist sind EV- und Nicht-EV-Zertifikate am stärksten verbreitet. Der Anteil der EV-Zertifikate flacht fortwährend über das Alexa-Ranking leicht ab. Die Anteile der “normalen” Nicht-EV-Zertifikate nehmen zunächst in einer starken Kurve ab um anschließend bis zum Ende des Alexa-Rankings leicht zu steigen.

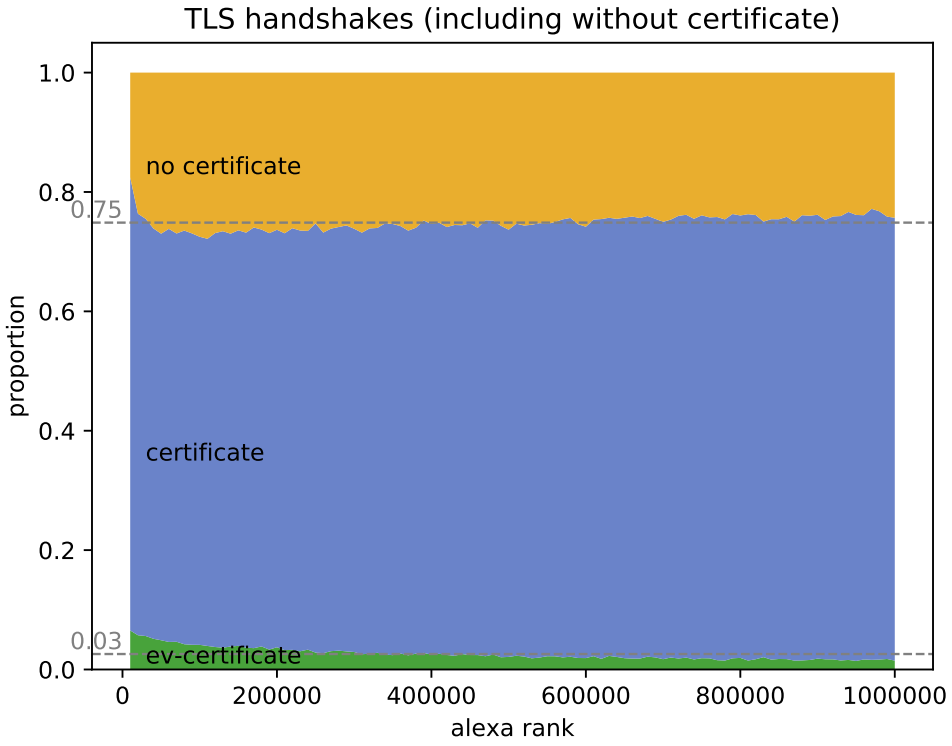

Abbildung 16: TLS Handshakes (alexa ranked; without timeouts)

Tabelle 4: Verifications of SCTs

	count	percent
all	1038227	100.00
verified	1015784	97.84
unverified; ctlog known	22401	2.16
unverified; ctlog unknown	42	0.00

Tabelle 5: Verifications of SCTs by-cert

	count	percent
all	865852	100.00
verified	845810	97.69
unverified; ctlog known	20000	2.31
unverified; ctlog unknown	42	0.00

Tabelle 6: Verifications of SCTs by-tls

	count	percent
all	172099	100.00
verified	169703	98.61
unverified; ctlog known	2396	1.39
unverified; ctlog unknown	0	0.00

Tabelle 7: Verifications of SCTs by-ocsp

	count	percent
all	276	100.00
verified	271	98.19
unverified; ctlog known	5	1.81
unverified; ctlog unknown	0	0.00

In Tabelle 4 sind die Anzahlen über die Verifikationen aller erhaltenen SCTs aufgelistet. In den Tabellen 5, 6 und 7 sind die Verifikationen der SCTs nach der jeweiligen Art der Übermittlung beim TLS-Handshake geführt.

Tabelle 8: SCTs by Deliver Way (w/wo www. prefix)

	count	percent
all	1038227	100.00
by-cert	865852	83.40
by-tls-extension	172099	16.58
by-ocsp-response	276	0.03

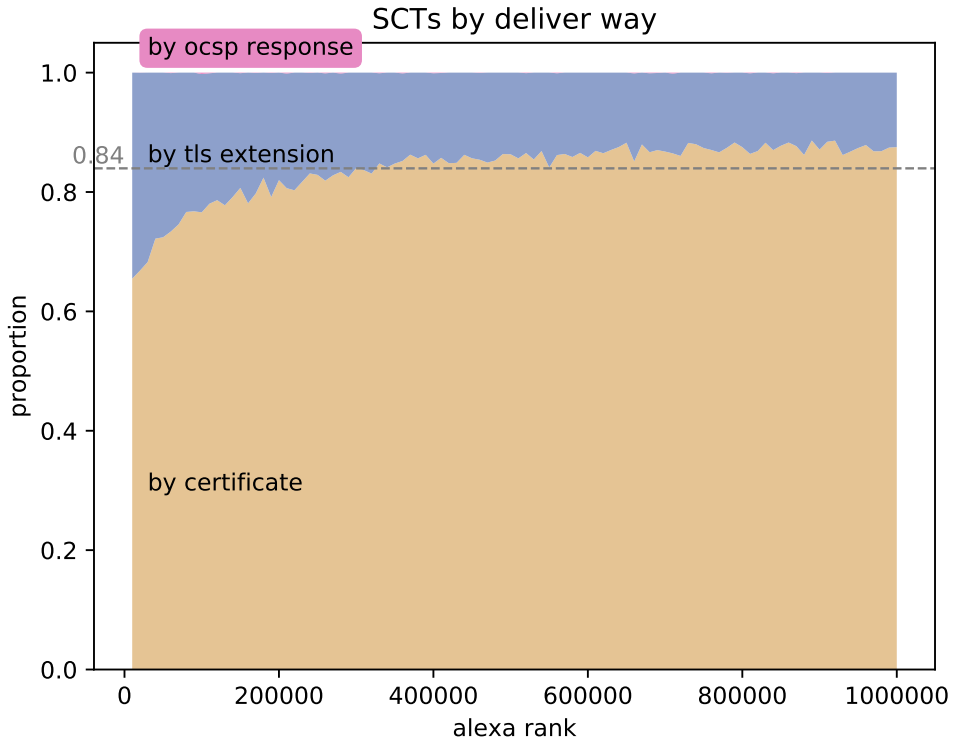

Abbildung 17: SCTs by Deliver Way (alexa ranked)

Die Anteile der SCTs nach der jeweiligen Art der Übermittlung sind in Tabelle 8 aufgelistet. Der Anteil der SCTs die im Zertifikat enthalten sind (by-cert) liegt bei rund 84 Prozent. Der Anteil der per TLS-Extension übertragenen SCTs (by-tls) liegt im Durchschnitt bei unter 17 Prozent. Der Anteil der mittels OCSP-Status-Response übertragenen SCTs (by-ocsp) ist mit drei Zehntel-Promille vernachlässigbar. Im Graphen der Abbildung 17 lässt sich ablesen, daß der überdurchschnittliche Anteil an per TLS-Extension übertragenen SCTs zunächst stark zu-

rückgeht um sich ab einem Alexa-Rank von rund 400000 bis zum Ende des Rankings relativ stabil bei rund 14 Prozent einzupendeln.

Tabelle 9: SCTs of EV-Certificates by Deliver Way (w/wo www. prefix)

	count	percent
all	134205	100.00
by-cert	133932	99.80
by-tls-extension	141	0.11
by-ocsp-response	132	0.10

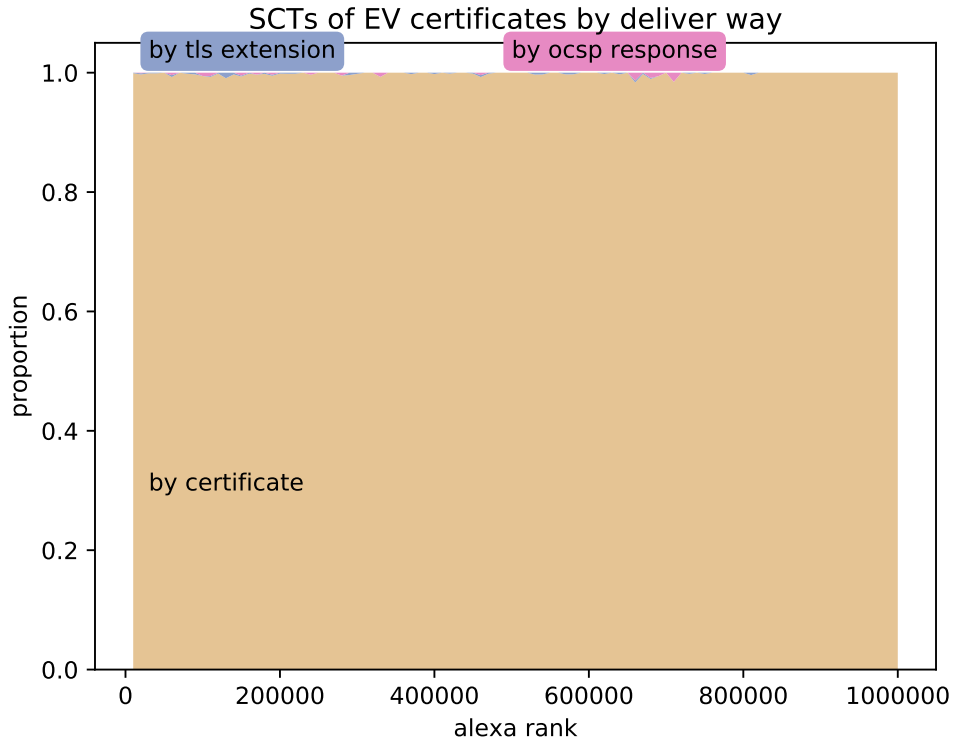

Abbildung 18: SCTs of EV Certificates by deliver way (alexa ranked)

Tabelle 9 listet die Anteile der SCTs nach der jeweiligen Art der Übermittlung nur für EV-Zertifikate auf. Jeweils rund ein Tausensdel der SCTs wurden per TLS-Extension oder OCSP-Status-Response übermittelt. So gut wie alle SCTs der EV-Zertifikate, das sind 99,8 Prozent,

sind im übermittelten Zertifikat selber enthalten. Dies wird im Graphen der Abbildung 18 stark verdeutlicht.

Tabelle 10: Certificates with or without SCTs (w/wo www. prefix)

	count	percent
all	1329819	100.00
no SCTs	990221	74.46
1 or more SCTs	339598	25.54
1 SCT	53	0.00
2 SCTs	111717	8.40
3 SCTs	130358	9.80
4 SCTs	63970	4.81
5 or more SCTs	33500	2.52

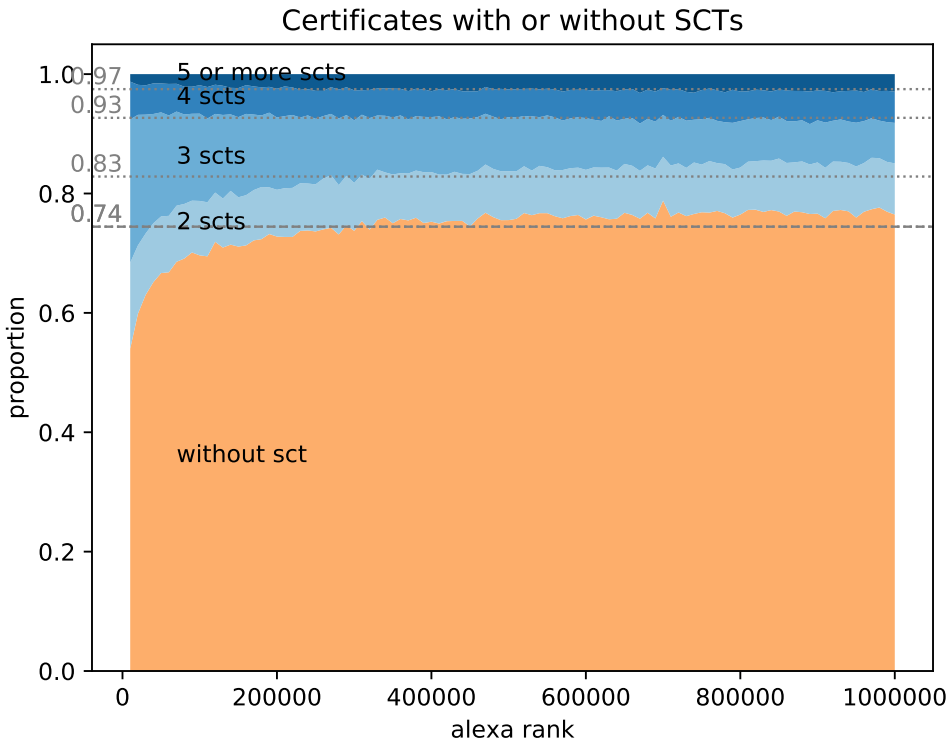

Abbildung 19: Certificates with or without SCTs (alexa ranked)

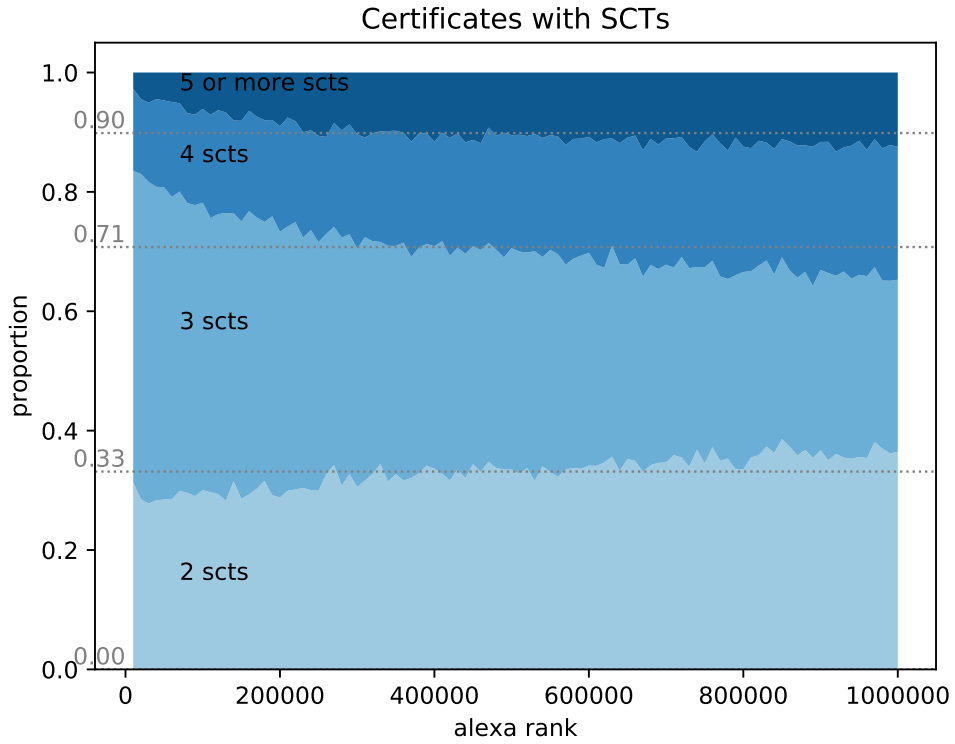

Abbildung 20: Certificates with SCTs (alexa ranked)

Tabelle 10 listet die absoluten Häufigkeiten und Anteile auf, über wie viele SCTs ein Zertifikat verfügt. Bei knapp mehr als 990000 TLS-Handshakes wurden keine SCTs übermittelt, und es wurde somit CT nicht unterstützt. Unterstützt wurde CT bei knapp 340000 TLS-Handshakes, indem mit dem Zertifikat auch SCTs übermittelt worden sind. Davon wurde in lediglich 53 Fällen mit dem Zertifkat genau ein SCT übermittelt. In allen anderen Fällen wurden zwei oder mehr SCTs übermittelt. Abbildung 19 stellt einen Graphen mit den anteiligen Häufigkeiten der SCTs je TLS-Handshake über das Alexa-Ranking dar. Rund 74 Prozent unterstützen kein CT, 26 Prozent unterstützen CT. Der Grahph in Abbildung 20 zeigt ausschließlich die relativen Häufigkeiten der Fälle, bei denen SCTs übermittelt worden sind. Bei den TLS-Handshakes wurden meistens drei, dann zwei, dann vier und zuletzt fünf oder mehr SCTs übermittelt. Am Anfang des Alexa-Rankings sind TLS-Handshakes, bei denen drei SCTs übermittelt worden sind, am stärksten vertreten. Mit absteigendem Ranking wird die Verteilung der SCT-Anzahlen zunehmend ausgeglichen.

Abbildung 21 zeigt wie Abbildung 19 die anteiligen Häufigkeiten der SCTs je Zertifikat auf, jedoch ausschließlich für EV-Zertifikate. In lediglich 2 Prozent wird CT nicht unterstützt. Der

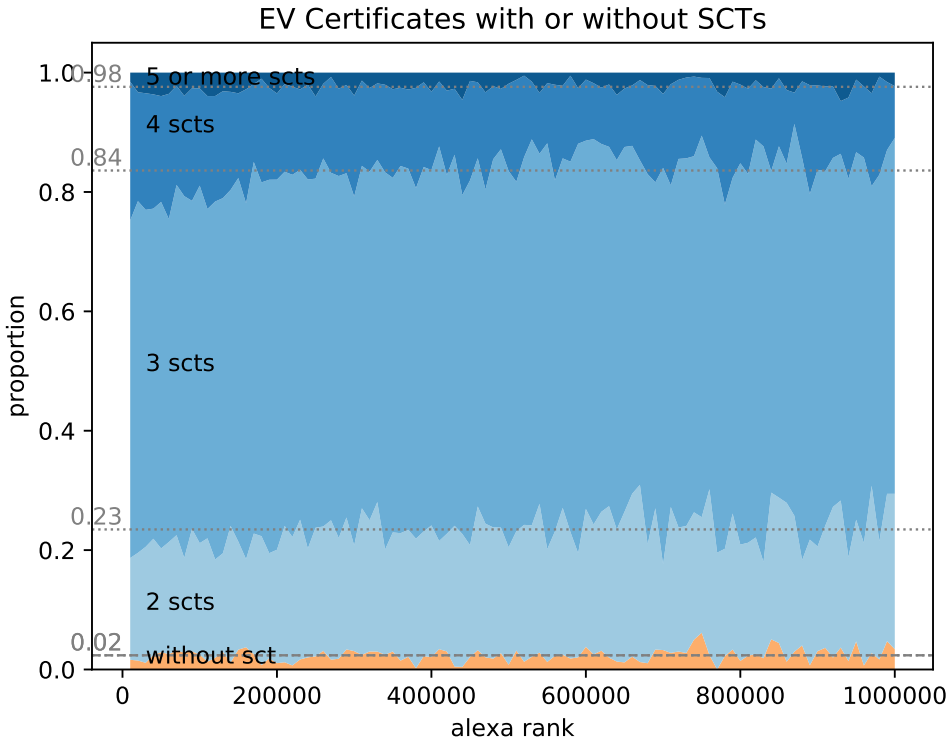

Abbildung 21: EV Certificates with or without SCTs (alexa ranked)

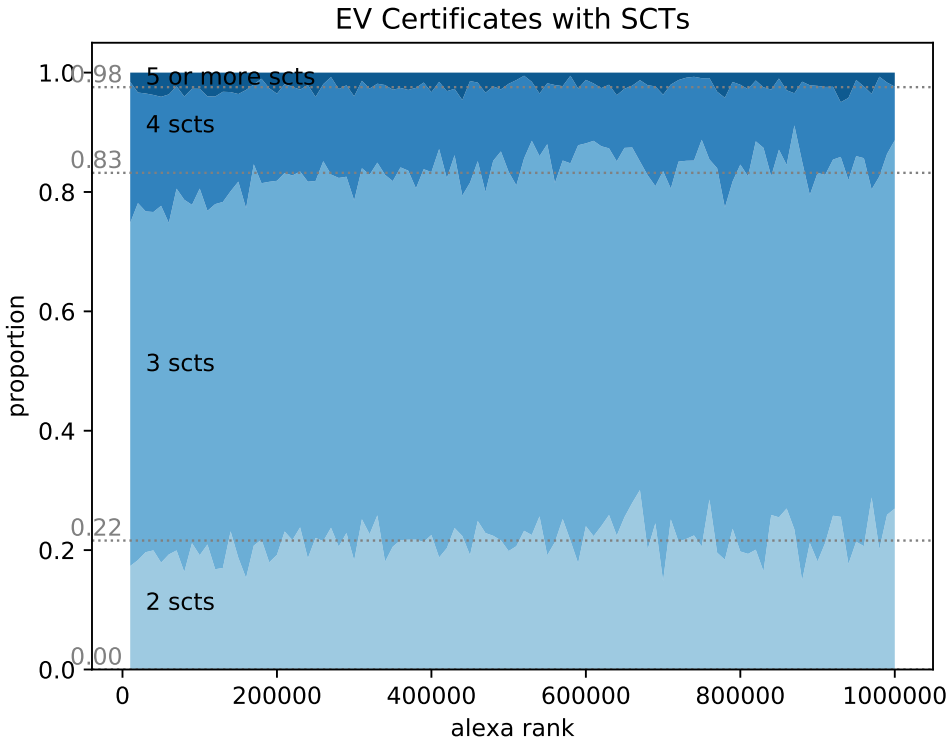

Abbildung 22: EV Certificates with SCTs (alexa ranked)

Anteil der Zertifikate, in denen SCTs mit übermittelt werden liegt bei 98 Prozent. Abbildung 22 stellt analog zur Abbildung 20 den Graphen über das Alexa-Ranking nur für SCTs von TLS-Handshakes mit EV-Zertifikaten dar. Der Anteil der TLS-Handshakes, bei denen drei SCTs übermittelt worden sind, also drei SCTs im Zertifikat enthalten sind, ist der stärkste mit deutlich mehr als 61 Prozent. Insgesamt verlaufen die Anteile über das ganze Ranking konstanter als es bei den Anteilen der TLS-Handshakes aller Zertifikate der Fall ist. Der Anteil über 4 SCTs ist bei hoher Popularität leicht erhöht.

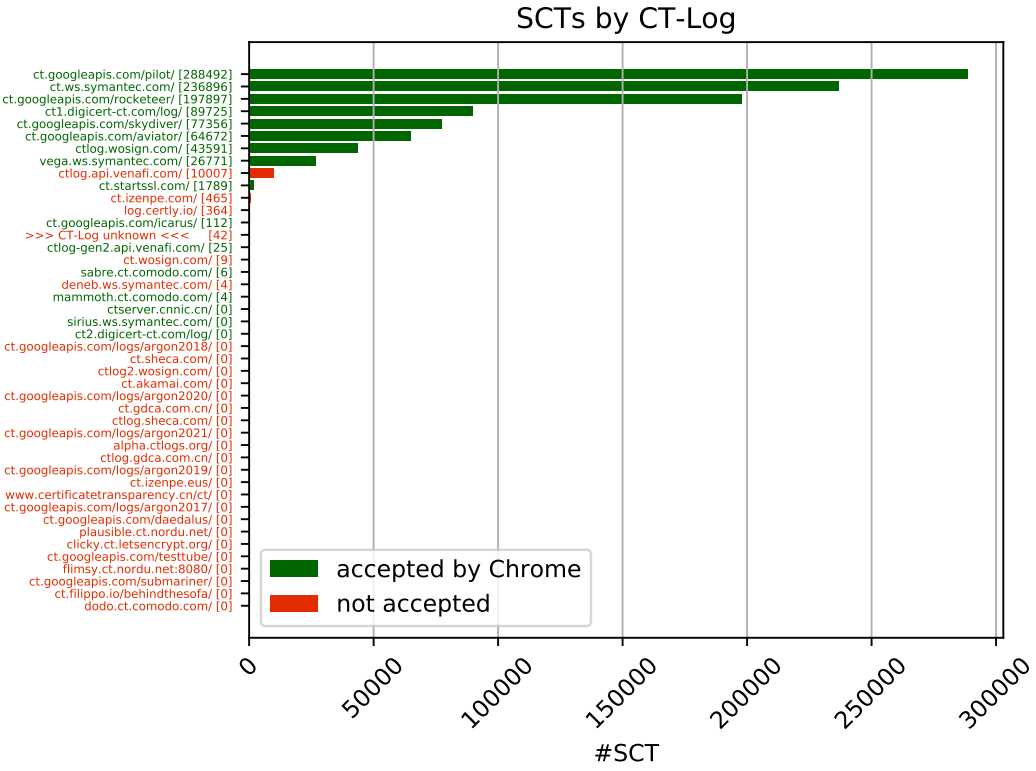

Abbildung 23: SCTs by CT-Log

Das Diagramm in Abbildung 23 zeigt auf, wie viele der bei den TLS-Handshakes übermittelten SCTs von den jeweiligen CT-Logs ausgestellt worden sind. Es sind alle auf der CT-Webseite [10] zum Zeitpunkt der Erhebung veröffentlichten CT-Logs aufgeführt. SCTs von CT-Logs, die von Chrome nicht als Veröffentlichungsnachweis akzeptiert werden, sind gar nicht oder nur sehr gering vorhanden. Von den drei CT-Logs, `ct.googleapis.com/pilot/`, `ct.ws.symantec.com/` und `ct.googleapis.com/rocketeer/` der zwei CT-Log-Operatoren `Google` und `Symantec`, sind 70 Prozent der SCTs ausgestellt worden. Abbildung 24 zeigt einen Graphen der CT-Log-Anteile über das Alexa-Ranking. Die Anteile der drei

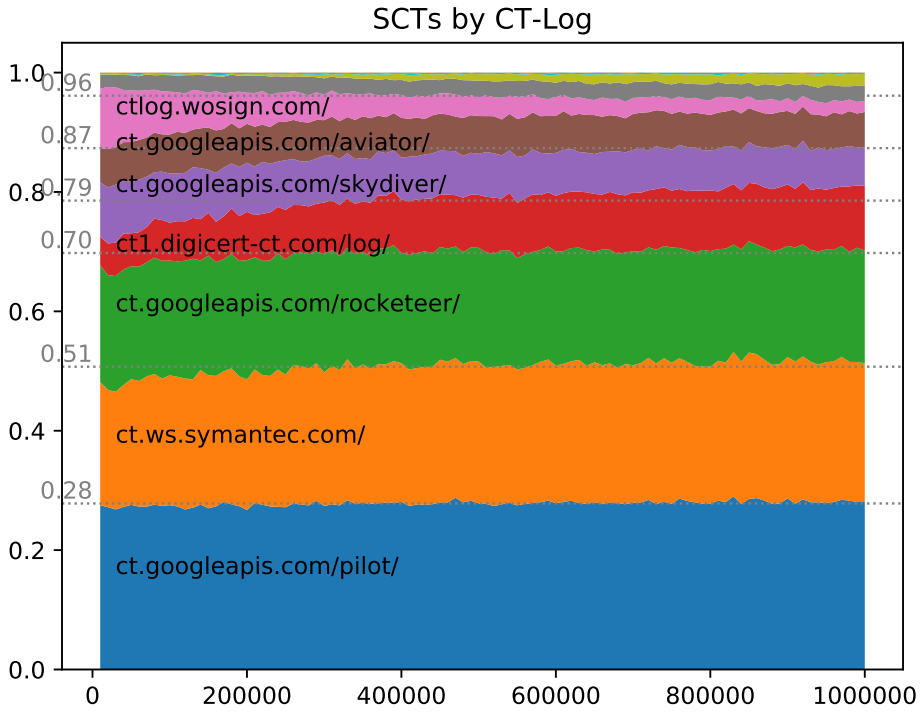

Abbildung 24: SCTs by CT-Log (alexa ranked)

meistvertretenden CT-Logs sind über das ganze Ranking relativ ausgeglichen. Eine leichte Delle der andere CT-Logs bei hohem Ranking wird durch einen stärkeren Anteil des CT-Logs `ctlog.wosign.com/` ausgeglichen.

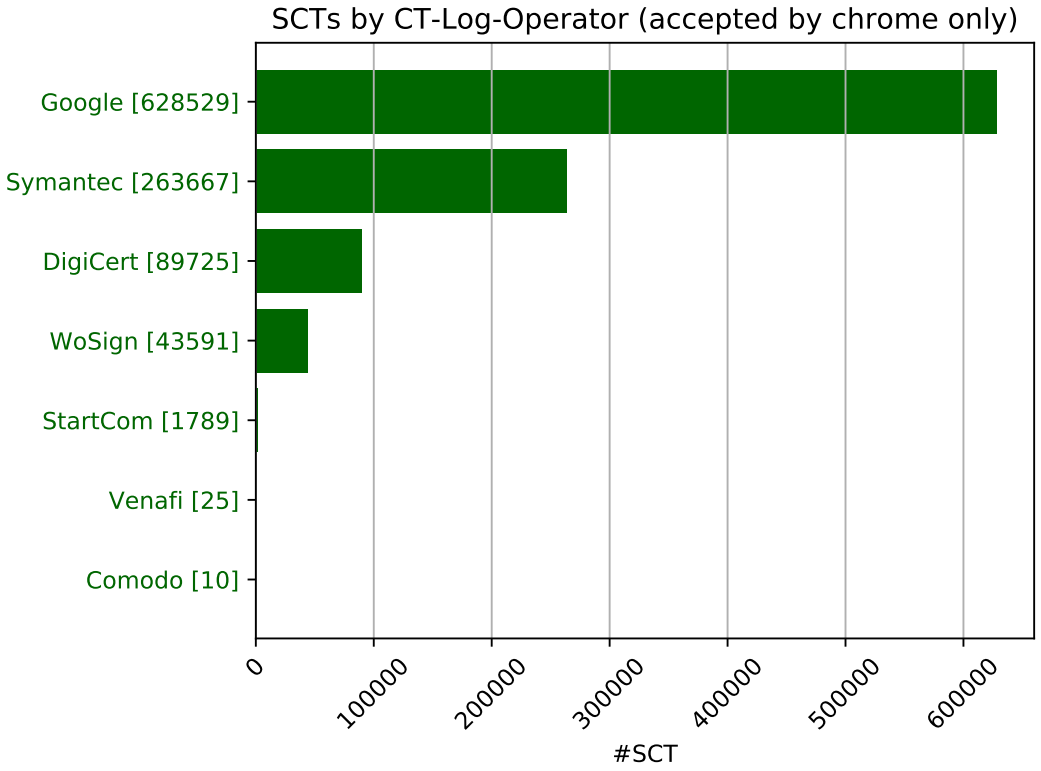

Abbildung 25: SCTs by CT-Log Operator

In dem Diagram der Abbildung 25 sind die absoluten Häufigkeiten der übermittelten SCTs nach den CT-Log-Operatoren aufgeführt, von denen die SCTs der TLS-Handshakes stammen. Es sind nur die CT-Logs-Operatoren gelistet mit von Chrome akzeptierten CT-Logs. Vier CT-Log-Operatoren haben relevante Anteile an eingesetzten SCTs, die anderen drei CT-Log-Operatoren spielen in der Praxis zur Zeit keine Rolle. Dies wird besonders deutlich im Graphen in Abbildung 26. Mit 61 Prozent der SCTs stammen mehr als die Hälfte aller SCTs von durch `Google` betriebene CT-Logs. Rund 87 Prozent der SCTs stammen von nur zwei unterschiedlichen CT-Log-Operatoren, von `Google` und von `Symantec`.

In der Abbildung 27 ist ein Graph dargestellt, der die Anteile der Zertifikate zeigt, deren SCTs ausschließlich von den CT-Logs mit den meisten in Produktion verwendeten SCTs stammen. Rund drei Prozent der TLS-Handshakes mit SCTs enthalten ausschließlich SCTs von CT-Logs, die nur von `Google` betrieben werden. Bei Rund 37 Prozent der TLS-Handshakes mit SCTs

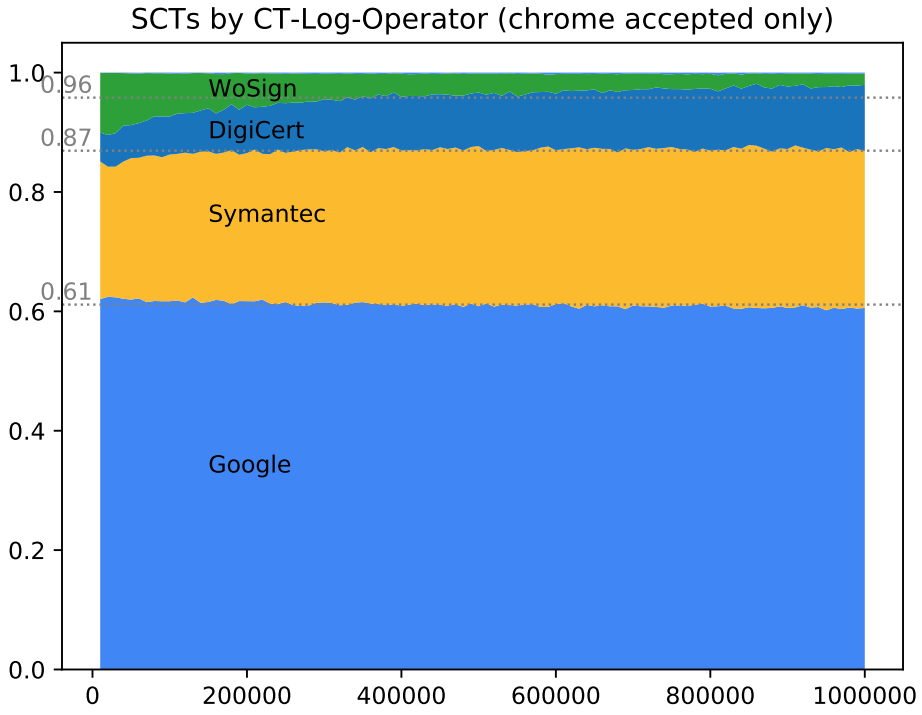

Abbildung 26: SCTs by CT-Log (alexa ranked)

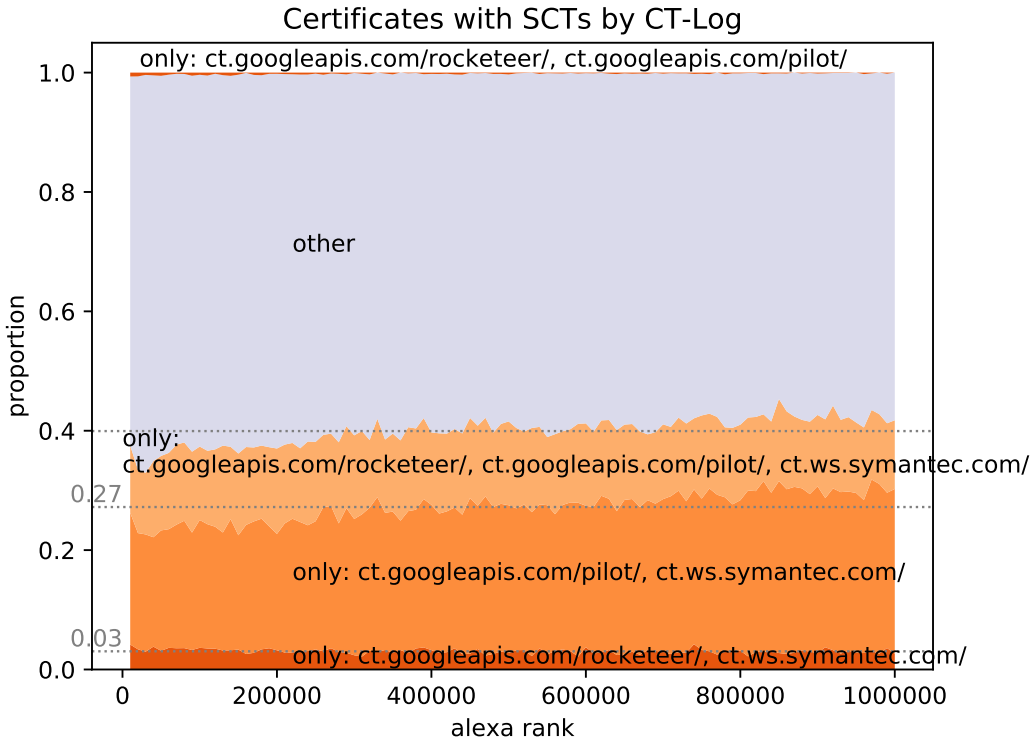

Abbildung 27: Certificates by CT-Log (alexa ranked)

stammen die SCTs ausschließlich von `Google` und von dem von `Symantec` betriebenen CT-Log `ct.ws.symantec.com/`.

4 Zusammenfassung und Ausblick

Im vorigen Kapitel haben wir die Ergebnisse der Erhebung von zwei Millionen TLS-Handshake-Versuchen über die Alexa-Top1M-Liste (w/wo. www-Präfix) untersucht. Von rund 1,3 Millionen durchgeführten TLS-Handshakes wurden in knapp 340000 Fällen SCTs mit ausgeliefert, was einem Anteil von ca. 25,5 Prozent entspricht. Knapp 75 Prozent der mit HTTPS verschlüsselten Webseiten müssen bis Anfang April 2018 eine Unterstützung von CT implementieren, um im Chrome Browser ohne Warnung angezeigt zu werden.

Es wurde aufgezeigt, daß die Übertragung von SCTs über den OCSP-Mechanismus praktisch nicht eingesetzt wird. Eine entsprechende OCSP-Status-Response ist lediglich bei 276 SCTs erfolgt (0,03 Prozent).

Die Diversität von CT im Deployment ist verbesserungswürdig. Bei den CT-Log-Anteilen der bei den TLS-Handshakes ausgelieferten SCTs dominieren drei CT-Logs. Zwei davon werden von Google betrieben, das Dritte ist das von Symantec betriebene CT-Log. Rund 87 Prozent aller SCTs stammen von diesen drei CT-Logs.

Viele Zertifikate haben SCTs von nur zwei unterschiedlichen Log-Operatoren. Auch hier ist mehr Diversität wünschenswert. Von den knapp 340000 TLS-Handshakes die CT unterstützt haben, wurden bei ca. 37 Prozent nur SCTs von Google und Symantec ausgeliefert. Würde das Symantec-Log nicht mehr von Chromium akzeptiert werden, wären die TLS-Handshakes mit nur einem akzeptierten CT-Log nicht mehr konform nach der CT-Chrome-Policy.

Die Entwicklung von CT wird vorangetrieben und dominiert von Google:

- RFC6962 wurde ausschließlich von Google-Mitarbeitern geschrieben [17].
- Der Nachfolge-Draft draft-ietf-trans-rfc6962-bis-08 entsteht unter Federführung von Google [18].
- Die offizielle Website (http://www.certificate-transparency.org/) zu CT wird von Google betrieben. Dort wird die Liste aller bekannten CT-Logs (“known logs”) und der von Chromium/Chrome akzeptierten CT-Logs veröffentlicht.
- Es gibt eine Open-Source-Implementierung von Google, die fortwährend weiterentwickelt wird [9].

Und das Deployment wird ebenfalls von Google kontrolliert und dominiert:

- Google setzt den Einsatz von CT durch, indem die Verwendung von mittels CT veröffentlichten Zertifikaten im Webbrowser Chrome von Google Schritt für Schritt strenger gefördert und schließlich erforderlich wird [12]. So wird seit Januar 2015 bereits gefordert, daß in Webseiten eingesetzte EV-Zertifikate in mindestens zwei Logs veröffentlicht werden. Ab April 2018 werden alle Webserver-Zertifikate CT unterstützen müssen, um von Chrome akzeptiert zu werden.

- 61 Prozent der bei den TLS-Handshakes übertragenen von Chrome akzeptierten SCTs stammen von CT-Logs, die von Google betrieben werden.
- Die Listen der bekannten ("known logs") und der von Chromium/Chrome akzeptierten CT-Logs werden von Google festgelegt und auf der von Google betriebenen Webseite zu CT veröffentlicht.

Der vergangene Zustand dieser Listen ist fehlerbehaftet gewesen, und auch eine Änderungsverfolgung ist derzeit noch im Aufbau [10]. Auch die möglichen Zustände (z.B. von Chrome akzeptiert, verworfen, usw.) und ihre möglichen Zustandsübergänge und Ursachen sind bisher nicht klar definiert worden [10]. Hier ist noch Verbesserung möglich und angekündigt worden [10]. Wünschenswert wäre es, wenn die CT-Log-Listen, zumindest die Liste der bekannten CT-Logs von einer unabhängigen Partei - etwa dem CA-Browser-Forum - und nicht von Google selber erstellt werden würden.

4.1 Ausblick

Der Autor erwartet, daß der Anteil der per TLS-Extension übertragenen SCTs zunehmen wird. Dieser Mechanismus liegt unter der Kontrolle der Webserver-Administratoren und kann nachträglich zu einem bereits existierenden Webserver-Zertifikat konfiguriert werden.

Um den Administrator hierbei zu unterstützen wird es in naher Zukunft ein ctutlz-Skript geben, mit dessen Hilfe ein Zertifikat bei einer Auswahl von CT-Logs veröffentlicht wird und die entsprechenden SCTs von den CT-Logserver-Antworten erstellt werden.

Mögliche weitere Erhebungen und Untersuchungen bieten sich an:

- Der Anteil der SCTs, bei denen die Verifikation fehlgeschlagen ist, ist bei den im Zertifikat enthaltenen SCTs höher als bei SCTs, die mittels TLS- oder OCSP-Extension übertragen worden sind. Hier lohnt es sich, genauer nach einer Ursache zu suchen.
- Ermitteln, in welchem Umfang Kombinationen der Auslieferungs-Arten (by-Cert, by-TLS, by-OCSP) der SCTs vorkommen.
- Erheben, ob Zertifikate auch in CT-Logs veröffentlicht sind zu denen keine SCTs beim TLS-Handshake ausgeliefert werden.

Bis zur verbindlichen Einführung von CT durch Chromium/Chrome im April 2018 und darüber hinaus wird sich das CT-Deployment wohl noch stark verändern. Diesen Verlauf zu erfassen und zu untersuchen wird ein spannendes, interessantes Arbeitsfeld darstellen. Hierfür kann ctutlz und ctstats erweitert werden. Der Grundstein der technischen Implementierung ist mit dieser Arbeit umgesetzt worden und kann entsprechend ausgebaut werden. Das in C geschriebenen Tool tls-scan [3] könnte um die benötigte CT-Funktionalität ergänzt werden, um eine performantere Umsetzung der von ctutlz bereitgestellten Funktionalität zu bekommen. tls-scan wiederum könnte dann die Daten für die Erhebung an ctstats liefern.

Es gibt die Möglichkeit, per DNS-Abfrage die SCTs zu Domains zu holen [2]. Dieser Mechanismus ist (noch) experimentell, könnte jedoch deutlich schneller für die Top-1M-Alexa-Liste durchgeführt werden, als zu jedem Webserver einzeln einen TLS-Handshake auszuführen.

Daß eine hohe Diversität der CT-Logs wünschenwert ist, wurde auch in der Google-Group zu CT besprochen [7]. Dort wurde auf unterschiedliche Arten von Diversität eingegangen. Hier könnte gut angeknüpft werden und die anstehende Umgestaltung der CT-Log-Listen konstruktiv begeleitet werden.

Literatur

[1] D. Eastlake 3rd and T. Hansen. US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF). RFC 6234, IETF, May 2011.

[2] Ben Laurie (Google). Draft: Certificate Transparency over DNS. https://github.com/google/certificate-transparency-rfcs/blob/master/dns/draft-ct-over-dns.md, Abrufdatum: 12. November 2017.

[3] Binu Ramakrishnan (prbinu). tls-scan. https://prbinu.github.io/tls-scan/, Abrufdatum: 12. November 2017.

[4] chromium.org. Chromium Code Reviews: Issue 7795014: Mark DigiNotar as untrusted. https://codereview.chromium.org/7795014, Abrufdatum: 30. August 2017.

[5] Daniel Bachfeld, heise online. DigiNotar wird liquidiert. http://heise.de/-1347001, Abrufdatum: 21. September 2017.

[6] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, IETF, August 2008.

[7] Eran Messeri (Google). Log diversity requirements group at the CT policy days - session notes. https://groups.google.com/a/chromium.org/forum/#!topic/ct-policy/rlp5zUTlVXw, 24. Februar 2017.

[8] Google. Certificate Transparency in Chrome. https://a77db9aa-a-7b23c8ea-s-sites.googlegroups.com/a/chromium.org/dev/Home/chromium-security/root-ca-policy/CTPolicyMay2016edition.pdf?attachauth=ANoY7cpq_bPjBPZshQLwFnBxXBbr4E-k21FkrL43tHlB75lH5l90qx5-dJLxMczfD-onJeuJ0Bf_kLRv_pjKdgeHLUZg8YHoBZk9O6eLcxZxzKeKGz1j9Lyc3VKNqdXCAX48YhHpHR79UWDNbpPTdYEIMwZ8mEqEU8YiV0Fhd2Tiq_BijSdF_JUjfyPZMBGt9NcC7--7z9NHMEvw2W6AcFjlmGt3yvE7CxZCzuhFA%3D%3D&attredirects=0, Mai 2016.

[9] Google. Certicate Transparency: Auditing for TLS Certificates. https://github.com/google/certificate-transparency, Abrufdatum: 12. Dezember 2017.

[10] Google. Certificate Transparency - Known Logs. https://www.certificate-transparency.org/known-logs, Abrufdatum: 12. November 2017.

[11] Google. Transparency Report. http://transparencyreport.google.com/https/overview, Abrufdatum: 11. November 2017.

[12] Google Inc. Extended Validation in Chrome. http://www.certificate-transparency.org/ev-ct-plan, Abrufdatum: 30. August 2015.

[13] Hanno Böck. Log lists outdated and inconsistent. https://groups.google.com/forum/?fromgroups#!topic/certificate-transparency/zBv7EK0522w, 18. Juli 2017.

[14] Hans Hoogstraaten and others, Fox-IT BV. Black Tulip: Report of the investigation into the DigiNotar Certificate Authority breach. http://heise.de/-1741726, Abrufdatum: 2. November 2017.

[15] Iranian Gmail-User 'alibo'. Report via Gmail: Is This MITM Attack to Gmail's SSL ? https://productforums.google.com/forum/?hl=en#!category-topic/gmail/share-and-discuss-with-others/3J3r2JqFNTw, Abrufdatum: 27. August 2017.

[16] Johnathan Nightingale, mozilla.org. Mozilla Security Blog: Fraudulent *.google.com Certificate. https://blog.mozilla.org/security/2011/08/29/fraudulent-google-com-certificate/, Abrufdatum: 6. Sept. 2017.

[17] B. Laurie, A. Langley, and E. Kasper. Certificate Transparency. RFC 6962, IETF, June 2013.

[18] Ben Laurie, Adam Langley, Emilia Kasper, Eran Messeri, and Rob Stradling. Certificate Transparency. Internet-Draft – work in progress 11, IETF, November 2015.

[19] Matěj Cepl. M2Crypto. https://gitlab.com/m2crypto/m2crypto/blob/master/README.rst, Abrufdatum: 12. November 2017.

[20] H.X. Mel and D.M. Baker. *Cryptography Decrypted*. Cryptography Decrypted. Addison-Wesley, 2001.

[21] OpenSSL Software Foundation. OpenSSL. https://www.openssl.org/, Abrufdatum: 12. November.

[22] OpenSSL Software Foundation. OpenSSL (github). https://github.com/openssl/openssl, Abrufdatum: 12. November.

[23] Pier Carlo Chiodi (Pierky). Manually Verify SCT with OpenSSL. https://blog.pierky.com/certificate-transparency-manually-verify-sct-with-openssl/, 29. April 2015.

[24] Python CFFI. CFFI - Foreign Function Interface for Python calling C code. https://bitbucket.org/cffi/cffi, Abrufdatum: 13. November 2017.

[25] Python Cryptographic Authority (pyca). pyca/cryptography. https://github.com/pyca/cryptography, Abrufdatum: 13. November 2017.

[26] Python Cryptographic Authority (pyca). pyOpenSSL - github-Repository. https://github.com/pyca/pyopenssl, Abrufdatum: 13. November 2017.

[27] E. Rescorla. HTTP Over TLS. RFC 2818, IETF, May 2000.

[28] Ronald Eikenberg, heise online. Protokoll eines Verbrechens: DigiNotar-Einbruch weitgehend aufgeklärt. http://heise.de/-1741726, Abrufdatum: 2. November 2017.

[29] Ryan Sleevi (Google). Announcement: Requiring Certificate Transparency in 2017. https://groups.google.com/a/chromium.org/forum/#!msg/ct-policy/78N3SMcqUGw/ykIwHXuqAQAJ, 25. Oktober 2016.

[30] Ryan Sleevi (Google). Certificate Transparency in Chrome - Change to Enforcement Date. https://groups.google.com/a/chromium.org/forum/#!msg/ct-policy/sz_3W_xKBNY/6jq2ghJXBAAJ, 21. April 2017.

[31] K. Schmeh. *Kryptografie: Verfahren, Protokolle, Infrastrukturen*. iX Edition. dpunkt.Verlag, 2013.

[32] Seth Schoen and Eva Galperin. Iranian Man-in-the-Middle Attack Against Google Demonstrates Dangerous Weakness of Certificate Authorities. https://www.eff.org/deeplinks/2011/08/iranian-man-middle-attack-against-google, Abrufdatum: 29. August 2017.

[33] The Chromium Projects. Certificate Transparency. https://www.chromium.org/Home/chromium-security/certificate-transparency, Abrufdatum: 12. November 2017.

[34] The Matplotlib development team. https://matplotlib.org/, Abrufdatum: 13. November 2017.

[35] The Python Standard Library. ssl — TLS/SSL wrapper for socket objects. https://docs.python.org/3/library/ssl.html, Abrufdatum: 12. November 2017.

[36] Theodor Nolte. ctutlz. https://github.com/theno/ctutlz, Abrufdatum: 12. November 2017.

[37] Theodor Nolte. ctutlz – Ducktyping for Elliptic-Curve-Support. https://github.com/theno/ctutlz/blob/b48594710aeb6732fd13b6bb8336c55da13a576f/ctutlz/sct/verification.py#L27, 11. Juni 2017.

[38] Theodor Nolte. ctutlz – Using Callbacks to Gather SCTs. https://github.com/theno/ctutlz/blob/fe7b4012bc361a0866fea918e40f957399e10e46/ctutlz/tls/handshake.py#L142, 31. August 2017.

[39] Theodor Nolte. openssl-examples. https://github.com/theno/openssl-examples, Abrufdatum: 12. November 2017.

[40] Theodor Nolte. openssl-examples - Aufruf für verständlichen Beispiel-Code unter Test. https://github.com/theno/openssl-examples#motivation-outdated-directory-demos-in-openssl, Abrufdatum: 12. November 2017.

[41] Theodor Nolte. Pull-Request in PyopenSSL: Allow elliptic curve keys in from_cryptography_key(). https://github.com/pyca/pyopenssl/pull/636, 10. Juni 2017.

[42] Theodor Nolte. pyopenssl-examples. https://github.com/theno/pyopenssl-examples, Abrufdatum: 12. November 2017.

[43] Tim Peters. PEP 20 – The Zen of Python. https://www.python.org/dev/peps/pep-0020/, Abrufdatum: 13. November 2017.

[44] Matthias Wählisch, Robert Schmidt, Thomas C. Schmidt, Olaf Maennel, Steve Uhlig, and Gareth Tyson. RiPKI: The Tragic Story of RPKI Deployment in the Web Ecosystem. In *Proc. of 14th ACM Workshop on Hot Topics in Networks (HotNets)*, pages 11:1–11:7, New York, Nov. 2015. ACM.

[45] Wikipedia. Duck typing. https://en.wikipedia.org/wiki/Duck_typing, Abrufdatum: 15. November 2017.