Radboud University Nijmegen

Master Thesis

Implementation and Analysis of Merkle Tree Certificates for Post-Quantum Secure Authentication in TLS

Author:
Maximilian Pohl
s1103073

Supervisor:
dr. ir. Bart Mennink
b.mennink@cs.ru.nl

Second Reader:
prof. dr. Peter Schwabe
p.schwabe@cs.ru.nl

Daily Supervisor:
Marlon Baeten M.Sc.
marlon@tweedegolf.com

January 30, 2025

Abstract

Constant improvements in quantum computers pose an increasing risk to the security of TLS, a vital part of the security in today's internet. The transition to post-quantum secure cryptography in TLS includes multiple challenges. This work focuses on the challenge of identifying the server to the client. Currently, X.509-based certificates allow browsers to check the server identity cryptographically. Naively replacing signatures in these certificates will significantly increase their size and thus slow down connections. Merkle Tree Certificates (MTCs) are a proposal to mitigate this increase as an optional optimization to the current Public Key Infrastructure (PKI). We analyzed this proposal regarding transmission sizes and CPU usage and concluded that these performance metrics outperform X.509 certificates with classical and post-quantum signatures. Moreover, we implemented a library to verify MTCs and integrated it with the TLS library *Rustls*. Using our implementation, we performed a successful TLS handshake, negotiating MTCs and verifying them. Overall, this work showed that MTCs are a valuable addition to the X.509 certificate infrastructure, and we are looking forward to large-scale experiments in real deployments.

Contents

1 Introduction

1.1 Motivation

Continuous improvements in quantum computing pose an incalculable risk to encryption algorithms used today. It is unclear when quantum computers will be capable of breaking today's encryption algorithms or even if they will ever be able to. Still, it is necessary to develop new ciphers and protocols already, as it takes much time and research to become confident in the security of a new cryptographic system.

One of the most important encryption protocols used today is TLS. Websites use it to protect the integrity, authenticity, and confidentiality of communication with the browser. According to Firefox telemetry data from October 2024, about 83 % of all page loads worldwide — and even 94 % in the USA — are secured by HTTPS, which is based on TLS [1]. Besides HTTPS, more protocols use TLS to secure the communication. For instance, IMAP and SMTP – used for e-mail exchange – and LDAP, which is often used as a central place to store credentials in a corporate network, have secured variants that build on TLS.

Securing TLS against attacks by quantum computers consists of three parts: protecting the confidentiality of the connection, ensuring server authentication, and validating the server identity. The first is the most critical, as messages stored today can be decrypted retroactively by a *harvest now, decrypt later* attack. The Internet Engineering Task Force (IETF) Internet-Draft "Hybrid key exchange in TLS 1.3" [2] provides a solution to that, and browsers are in the process of rolling it out already [3], [4]. Between 12 % and 20 % of the connections are already protected that way [5]. The second part of the Post-Quantum (PQ) transition – server authentication – is covered by KEMTLS. It mitigates that current PQ-secure signatures are much bigger than their classical counterparts. As the name suggests, KEMTLS ensures server authentication using PQ safe Key-Encapsulation Mechanisms (KEMs) instead of signatures. As current PQ KEMs encapsulations are about half as big as PQ signatures, KEMTLS reduces the message size compared to the naive replacement of classical signature with PQ signatures [6].

The last of the three parts concerns securing the certification process that binds domain names to a long-living private key. The current infrastructure uses certificates issued by trusted Certification Authorities (CAs), which attests to the link between the domain name and private key. These certificates comprise numerous signatures, which would significantly increase the size of the certificate if naively substituted with their

PQ counterparts. Big certificates increase the data transferred during TLS handshakes, resulting in a worse performance or even broken connections due to non-standard conforming implementations that worked fine so far. [7]

To avoid large certificates, the IETF Internet-Draft "Merkle Tree Certificates for TLS" [8] proposes a new architecture for certificate infrastructures. It uses Merkle Trees and Merkle Tree inclusion proofs to reduce the size of messages exchanged during TLS handshakes. The architecture is designed for most common use cases but has a reduced scope compared to the current certificate infrastructure. Thus, the proposed architecture is meant to enhance the current certificate infrastructure rather than replace it.

1.2 Our Contributions

This work analyzes the Internet-Draft for Merkle Tree Certificates (MTCs) regarding the computational effort and number of bytes transferred during the TLS handshake and implements the necessary changes in a TLS stack for the first time. First, we compare the size of TLS handshake messages in a classical, X.509-based Public Key Infrastructure (PKI) with the message size of the proposed MTC architecture. We do this for both classical, non-PQ secure signature schemes and with the PQ signature schemes that the National Institute of Standards and Technology (NIST) recently specified. We show that the MTC architecture is more size efficient in all cases and handles the big sizes of PQ signatures much better than an X.509-based setup. Furthermore, the MTC setup requires a new update channel, as Relying Parties (RPs) must regularly refresh their roots of trust. We estimate and interpret the size of these updates based on different assumptions derived from statistics in the current PKI. Further, we estimate the CPU cycles clients use to verify MTC and X.509 certificates and compare them with each other.

As a second contribution, we created the first TLS implementation compatible with the MTC architecture. We based our implementation on the popular *Rustls* library and modified it to deal with two new negotiation mechanisms. These negation mechanisms become necessary to allow the client and server to agree on the MTC certificate type and a specific trust anchor. In addition, we developed a library to verify MTCs and integrated it into the Rustls library. This demonstrates that the MTC Internet-Draft works in practice, and we confirmed that the negotiation mechanisms maintain interoperability with the existing certificate infrastructure.

During the implementation process, we encountered some difficulties with the specification. We contributed fixes and added some improvements for all the problems we found. For example, we encountered incorrect test vectors caused by using a 16-bit rather than the specified 8-bit length prefix. We corrected these in the Internet-Draft and the CA implementation that produced these test vectors. Besides the fixes, we incorporated a new trust anchor negotiation mechanism into the proposed standard and implemented the required changes in the provided CA implementation. Moreover, we proposed a length prefix for the MTC embedding in the TLS `Certificate` message to allow parsing the TLS certificate message without depending on an external state. This length prefix

will be incorporated in the next pre-release of the standard. Beyond that, we suggested a standard file structure of MTC-related files for TLS clients and servers, based on how certificate files are organized on modern Linux-based computer systems nowadays, incorporating the changed needs that arise with the use of MTC.

1.3 Outline

We start this work with an Introduction containing the Motivation (Section 1.1) and Our Contributions (Section 1.2). Afterward, Chapter 2 introduces the preliminaries required for this work. This includes Section 2.1, which explains the basics of Merkle Trees, followed by Section 2.2, which provides an overview of the current Public Key Infrastructure (PKI), including OCSP, ACME, and Certificate Transparency (CT). Moreover, in Section 2.3, we recap the relevant parts of a TLS handshake and briefly present an optimized version called KEMTLS. As a last preliminary, Section 2.4 lists some classical and Post-Quantum (PQ) secure signature schemes with their performance metrics. Next, Chapter 3 introduces the Merkle Tree Certificate (MTC) architecture and contains sections that provide details about the Certification Authority (CA) (Section 3.1), the Transparency Service (Section 3.2), and the negotiation of MTC in TLS (Section 3.3). After explaining the MTC architecture, Chapter 4 compares the MTC with the X.509 architecture. We split specific considerations into four sections. Section 4.1 compares the certificate sizes and Section 4.2 investigates the size requirements of the new update mechanism. Moreover, in Section 4.3, we propose a common file structure for devices that support MTC, which we base on the file structure commonly used for X.509-related files. The last comparison, in Section 4.4, focuses on the CPU associated with using MTC or X.509 certificates. Further, Chapter 5 summarizes the development process of the Rustls-based MTC server and client implementation and contributions to the standardization process. Lastly, Chapter 6 summarizes the findings and provides items that deserve further attention.

2 Preliminaries

This section provides information relevant to understanding the architecture of Merkle Tree Certificate (MTC) and its implications. It starts with a refresher of Merkle Trees and continues with an explanation of the present Public Key Infrastructure (PKI), including its building blocks, the Automatic Certificate Management Environment (ACME), the Online Certificate Status Protocol (OCSP), and the Certificate Transparency (CT) design. Afterward, this section summarizes the TLS protocol and the optimization KEMTLS, and ends with a list of relevant Post-Quantum (PQ) secure signature algorithms.

2.1 Merkle Trees

Merkle Trees, also known as Hash Trees, are binary trees with the property that the root and each inner node is the result of a cryptographic hash function on its child nodes. Merkle Trees are tamper-evident and enable efficient verification of whether an element is included in the tree. The term "tamper-evident" refers to the inability to add, delete, or modify information contained within the Merkle Tree without changing the root node. An efficient verification means that, given the information and proof, one can easily verify that the information is contained in the root hash.

As a reminder, a hash function takes an arbitrary length input and produces a fix-length output. In the following, we will use $h = H(x)$ to denote that h results from applying the hash function H in the input x. In addition, a cryptographic hash function typically has three properties: Collision resistance, first preimage resistance, and second preimage resistance. Collision resistance means that it is hard to find any two inputs $x \neq x'$ that result in the same hash output $H(x) = H(x')$. First preimage resistance means that it is hard to find an input x that produces a given hash output h. In other words, given a hash h, it is hard to find x such that $h = H(x)$. Second preimage resistance means that it is hard to find a second input x' that produces the same hash as a given input x. In other words, given an input x, it is hard to find x' such that $H(x) = H(x')$. Hard in this context means that something is computationally infeasible, i.e., cannot be calculated in polynomial time [9, Section 9.2.1].

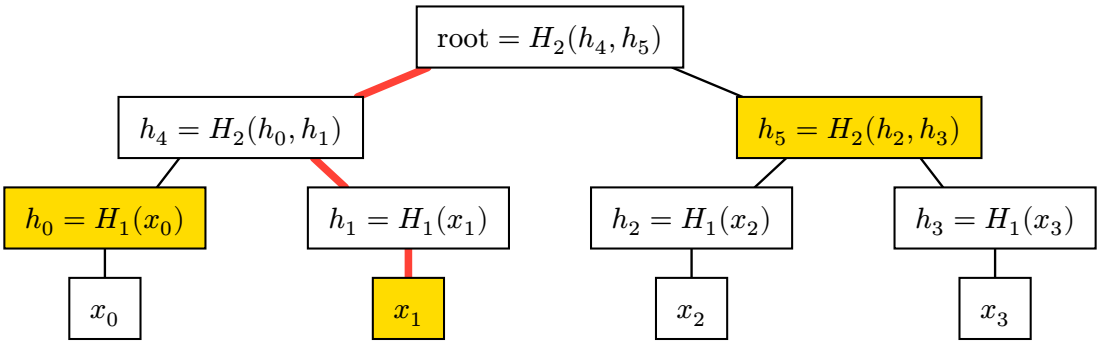

Figure 1: Visualization of a Merkle Tree with an inclusion proof for information x_1. The inclusion proof consists of the yellow marked h_0 and h_5 node, which allows a verifier to recalculate the red, thick path up to the root node.

Figure 1 shows an example tree for the information x_0, x_1, x_2 and x_3. The leaf nodes contain the hash of the information as $h_i = H_1(x_i)$. Each inner node is the result of the hash function $H_2(h_i, h_{i+1})$ with the content of the two child nodes. Therefore, one can build an inclusion proof to the root node without revealing any other information. The inclusion proof contains the sibling hash for each node traversing up to the root note, so h_0 and h_5 in the example in Figure 1.

Please note that the leaf and internal node use two different hash functions, H_1 and H_2. This ensures that an internal node can never be interpreted as a leaf node. This would allow the construction of multiple Merkle Trees with the same root hash [10]. In practice, it is enough to slightly alter the hash function, such as by prepending a single domain separator byte, which is different for the leaf and internal nodes [11]. The internal hash function H_2 takes two arguments, even though hash functions generally only take a single input. However, there are multiple ways to circumvent this restriction, such as concatenating the two inputs into one.

As long as the used hash function is collision-resistant, it is infeasible to alter, add, or delete any information included in the Merkle Tree without changing the hash of the root node [9, Section 13.4.1]. Instead, it is possible to add information to the tree and create a consistency proof showing that only specific data was added, but nothing else has changed in the tree. This property can be used to build logs that are verifiably append-only [11].

2.2 Public Key Infrastructure

A Public Key Infrastructure (PKI) is crucial to ensuring security in various digital systems. Its core functionality is to bind a cryptographic public key to a set of verified information [12]. Typically, this information represents an identity such as a domain name, company name, or the name of a natural person. However, in some cases, the verified information instead contains permissions or ownership without an identity. An

example is the Resource Public Key Infrastructure (RPKI), which is used to secure Border Gateway Protocol (BGP) announcements and, therefore, harden the security of routing on the internet [13].

The verified information and public key are combined with a cryptographic signature of the CA to form a *certificate*. The common format to encode the verified information and signature is X.509. Later, a Relying Party (RP) can parse the certificate, verify the signature, and trust the signed information given that it trusts the CA.

Such certificates are widely used in many applications. To name a few examples: It is used to encrypt and sign emails, to sign documents or software, to authenticate with a smart card, to restrict access to a Virtual Private Network (VPN), and to build a secure web browsing infrastructure [14].

This work focuses on this last use case, the web browsing. The corresponding PKI infrastructure is often referred to as *WebPKI*. It combines the domain name and possibly a company name with a public key, so an RP can verify that it is connected to the intended website. This verification is built into the HTTPS protocol, which combines the Hypertext Transfer Protocol (HTTP) with the TLS security layer. Typically, the CA does not issue a certificate from its root certificate that is stored as trusted in the RPs. Instead, an CA uses an intermediate certificate, signed with the root certificate, to sign the so-called End-Entity (EE) certificate, i.e., the one attesting the link between the domain name and public key. Together, the EE and corresponding intermediate and root certificates are referred to as *certificate chain*.

For security reasons, certificates have a limited lifetime, especially the EE in the web ecosystem. Since 2020, Chrome and Apple have enforced new EE certificates to be valid for at most 398 days [15], [16]. Let's Encrypt, a CA responsible for 57 % of all currently valid certificates, issues certificates for just 90 days [17], [18]. Let's Encrypt provides two reasons for their comparably short certificate lifetimes: They want to limit the damage a miss-issuance or key compromise can do and encourage an automated issuance process, which they see as a crucial part of widespread HTTPS adoption.

Nevertheless, it is important to have a mechanism in place to revoke certificates. The CA/Browser Form requires revocation mechanisms for all EE certificates that are valid for more than seven days [19]. Revocation becomes necessary if a private key leaks or the assured information is not accurate (anymore). There are two mechanisms in place for that: The CAs regularly publish Certificate Revocation Lists (CRLs), listing all revoked certificates and the Online Certificate Status Protocol (OCSP) allows RPs to query the CA if a certificate was revoked in real-time.

2.2.1 OCSP

OCSP is meant as an improvement over the classical CRL, as it avoids downloading a list of all blocked certificates occasionally but instead allows querying a CA about the status of one specific certificate whenever it is needed. The CA includes an HTTP endpoint to an OCSP responder in the certificates it issues, which RPs such as browsers can query for recent information about whether a certificate is valid [20].

In practice, this comes with a couple of issues: speed, high load on CA servers, availability, and privacy. Every time a RP checks a certificate, an additional round trip to the OCSP responder is required, which slows down the connection by about 30 % [21]. Moreover, the CAs have to answer these status requests, which results in a high server load and, therefore, costs. If an OCSP responder is not reachable, the RP either cannot connect to the server or has to ignore the failure, which is called fail-close and fail-open, respectively. The German healthcare system showcases that a fail-close approach can be very fragile in practice [22]. Browsers opted for a fail-open approach in favor of service availability. This decision, however, limits the benefit of recent information, as an attacker can block access to the OCSP endpoint and thereby block the information that a certificate was revoked [23]. Furthermore, OCSP raises privacy concerns, as CAs can build profiles of users based on which certificates they query.

OCSP stapling mitigates these issues. Instead of the user querying the OCSP responder, the server regularly does so and embeds the response in the certificate message of the TLS handshake [24, Section 8], [25, Section 4.4.2]. This reduces the load on the CA, eliminates the need for an additional round trip, fixes the privacy issues, and helps with service availability, as the OCSP responses are cached by the website server for a limited time.

2.2.2 ACME

As mentioned earlier, present certificate lifetimes are often 90 days, but not longer than 398 days. Short certificate lifetimes require automation to not overload humans with constant certificate renewals. Additionally, automation facilitates widespread adoption of HTTPS by lowering the (human) effort – and consequently costs – associated. Therefore, Let's Encrypt initiated the development of the Automatic Certificate Management Environment (ACME) in 2015 and started issuing certificates in that highly automated way in the same year [26]. The ACME protocol finally became an IETF standard in 2019 [27].

Please note that the fully automated ACME mechanism allows for *Domain Validation* (DV) certificates only. This means that the CA verifies that the requestor has effective control over the domain, as opposed to *Organization Validation* and *Extended Validation*, which require human interaction to verify the authenticity of the requesting organization. MTC requires a high degree of automation so DV certificates are the only practical certificate type for MTC. However, this is only a limited drawback for the applicable scope of MTC as 93 % of all valid certificates are DV certificates as of 2024-09-21 [17].

Three standardized ACME methods exist to verify that a user requesting a certificate has effective control over the domain: the `HTTP-01`, `DNS-01`, and `TLS-ALPN` challenges. Each generally works by placing a specific challenge value provided by the CA in a place that only the domain owner can do. As the names suggest, this is either a web page at a specific path, a TXT DNS entry, or a specific ALPN protocol in the TLS stack, respectively.

2.2.3 Certificate Transparency

Browser vendors ship a list of CAs which are trusted to issue genuine certificates only. As of November 6, 2024, there are 176 trusted root CAs built into Firefox and 134 in Chrome [28], [29]. If only a single CA misbehaves, this can tremendously impact the security of the whole system. One infamous example is DigiNotar's 2011 security breach, which allowed the attacker to listen into the connections of about 300,000 Iranian citizens with Google [30]. This was possible because the domain owner, i.e., Google, could not know that a certificate was issued in their name. In such a case, even the best certificate revocation mechanism is meaningless, as nobody could initiate it. As a direct consequence, Google initiated a program to ensure that all issued certificates are logged publicly so that a domain owner can recognize maliciously issued certificates and take action retroactively. This is referred to as Certificate Transparency (CT).

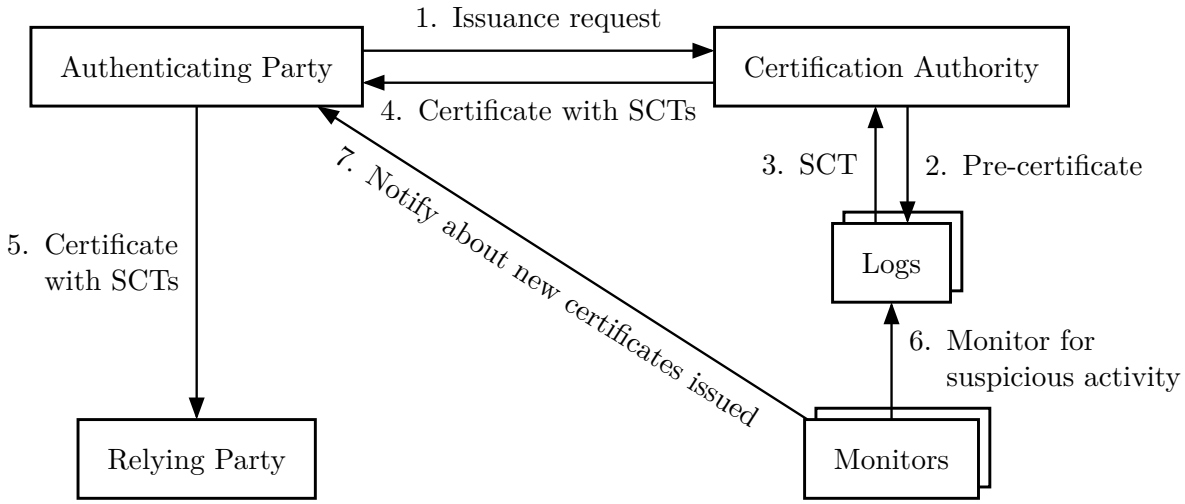

Figure 2: Certificate issuance with CT [31]

Figure 2 illustrates the certificate issuance flow with CT. Whenever a domain owner – subsequently called Authenticating Party (AP) – requests a new certificate from a CA and proves ownership of the domain, the CA creates a pre-certificate that is mostly identical to the final certificate except that it contains a poison extension and has no Signed Certificate Timestamps (SCTs) embedded yet. The poison extension ensures that this certificate is never accepted. The pre-certificate breaks the cyclic dependency, that a CT-log needs the certificate to create the SCT, but the CA needs the SCT to create the certificate. To issue the final certificate, the CA must send this pre-certificate to at least two independent CT-logs, which will ensure the certificate is logged publicly to an append-only log. In return, each log provides a SCT to the CA for including it in the final certificate. Subsequently, the CA returns the certificate with the embedded SCTs to the AP, which can use it whenever a RP connects thereafter. At the same time,

Monitors constantly watch the logs and possibly notify an AP for every certificate issued under their name. In addition to Monitors, there are also Auditors – not shown in the figure – that check the consistency of the log. This includes the append-only property, that all certificates are actually logged as promised, and that the log provides the same answers to all clients, independent of the location or other properties [31]. If the answers provided differ, this is called a *split-view* attack.

The functionality of an auditor may be spread over multiple entities. For example, the consistency checks could be performed by the monitors, while they are receiving the added certificates anyway. Additionally, there exists some specific and hardened hardware across the world that checks the log consistency over time and additionally tries to notice any split-view [32], [33]. It is more complex to check that a certificate actually gets included in the log after the log operator sends an SCT. To detect if a certificate was included in the log, browsers can request inclusion proof of a specific certificate for a signed tree head they trust. Unfortunately, this is difficult to do in practice, as browsers would leak which websites they visited to the log operators.

Nevertheless, CT is a success and practically unavoidable in the current WebPKI. This is primarily due to the requirements imposed by major web browsers: Chrome and Apple require their trusted root CAs to include at least two independent SCTs since 2018 and 2021, respectively [34], [35]. That way, effectively every certificate must be logged publicly to be of any value. This solves the problem of certificates that are unknown to an AP, and CT allows Monitors to analyze certificates and CAs for misbehavior.

2.3 TLS

Transport Layer Security (TLS) is building on a history back to the 1990s, when Netscape Communications published the first usable Secure Sockets Layer (SSL) version, SSL 2.0 in 1995 and shortly after SSL 3.0 in 1996. Afterward, the IETF took over the development, renamed the protocol to TLS, and published TLS 1.0 in 1999. The development went further, and the IETF published the TLS versions 1.1 and 1.2 in 2006 and 2008, respectively. Ten years later, a major overhaul was published as TLS 1.3 in 2018. It improved security and incorporated protocol simplifications and speed improvements [36].

The following will focus on TLS 1.3 as it is used for 94 % of all HTTPS requests according to Cloudflare Radar on 2024-09-23 and support for TLS 1.1 and older has been dropped by all relevant browsers in 2020 [5], [37], [38], [39], [40].

TLS consists of two sub-protocols: the handshake protocol for authentication and negotiation of cryptographic ciphers, followed by the record protocol for transmitting application data. The following concentrates on the handshake protocol and skips some functionality such as client certificates, the usage of previously or out-of-band negotiated keys, and 0-RTT data that can be used to send application data before the handshake is finished. This allows focus on the parts relevant to this thesis.

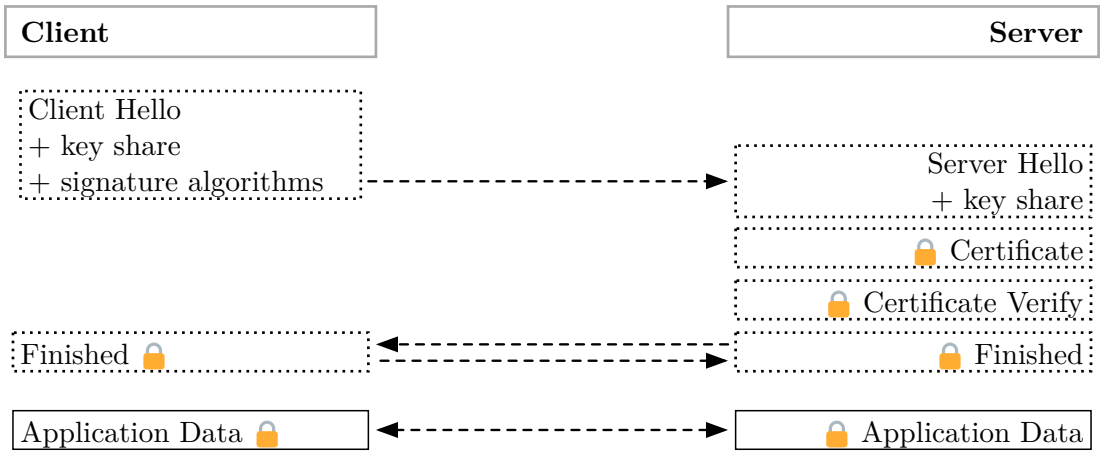

Figure 3: Overview of the simplified TLS 1.3 handshake [25]

Figure 3 illustrates the messages and extensions sent during the TLS handshake and whether they are encrypted. A TLS connection is always initiated by the client through a `ClientHello` message. This message contains extensions with a key share and signature algorithms the client supports. The server responds with a `ServerHello` message, which also includes a key share as an extension. Knowing both key shares, the server and client derive a shared symmetric secret and use it to protect all subsequent messages.

The following messages authenticate the server to the client by sending its certificate chain and a `CertificateVerify` message. The `CertificateVerify` message contains the signature over the handshake transcript up to that point. This proves that the server is in possession of the private key corresponding to the certificate and that messages have not been tampered with. The handshake ends with a `Finished` message each side sends and verifies. It contains a Message Authentication Code (MAC) over the transcript and thus assures the integrity and authenticity of the handshake. This MAC is not strictly necessary for security when performing a full handshake, as described in this work, but it is essential if protocol optimizations are used that allow using out-of-band or previously negotiated keys [41].

After the successful handshake, TLS continues to the record layer to exchange the application data.

2.3.1 KEMTLS

KEMTLS aims to improve the communication cost of TLS handshakes using the fact that the best currently available PQ KEMs are smaller than the PQ signatures. Instead of explicitly authenticating the server in the `CertificateVerify` using a signature, KEMTLS uses KEMs to authenticate the key exchange. The exact differences with the standard TLS handshake are not relevant to this work, so we refrain from explaining them here. However, it is relevant to note that a KEMTLS handshake requires a KEM

public key in the EE certificate instead of a signature key [6]. Hence, the utilization of KEMTLS impacts the size of the certificate, as shown in Section 4.1.

2.4 Post-Quantum Signatures

Table 1 compare the Elliptic Curve Digital Signature Algorithm (ECDSA) and RSA-2048 as classical signature schemes and the PQ signature schemes selected by the NIST for standardization. Module-Lattice-Based Digital Signature Algorithm (ML-DSA) was known as CRYSTALS-Dilithium and NIST standardized it as FIPS 204 in 2024, together with the Stateless Hash-Based Digital Signature Algorithm (SLH-DSA) as FIPS 205 [42], [43]. A NIST draft for the FFT (Fast-Fourier transform) over NTRU-Lattice-Based Digital Signature Algorithm (FN-DSA) was expected in late 2024, but has hot been published as of January 2025.

The NIST decided to specify three signature algorithms, as each of them has its benefits and drawbacks. ML-DSA is the recommended algorithm for most applications, as it has reasonable values in all categories. SLH-DSA is currently the most trusted algorithm, as it relies on the security of the well-established SHA-2 or SHA-3 hashing algorithms, that would need to be dramatically broken to harm the security of SLH-DSA [44]. This makes SLH-DSA a suitable candidate for long-term keys or situations where an upgrade is hard. FN-DSA might seem to have the best statistics, but it has the big drawback of relying on fast floating-point operations for signature generation. Without that, signing is about 20 times slower. The challenge with floating-point arithmetic is to implement it in constant time, and resistant against power analysis, as demonstrated by side-channel attacks against existing FN-DSA implementations [45], [46]. This is mainly a concern for signatures produced on-the-fly. If the signature is computed ahead-of-time, there is no timing leak to be observed. Moreover, verifying does not rely on floating-point arithmetic, and even if it did, there would not be a private key that could be leaked. [47]

Name	PQ	Sizes (bytes)		CPU cycles			
		Public Key	Signature	Signing		Verification	
				cycles	relative	cycles	relative
Ed25519	✗	32	64	46,314	1.00	158,754	1.00
ECDSA P-256	✗	32	64	108,545	2.34	255,095	1.61
RSA-2048	✗	256	256	1,850,021	39.95	44,881	0.28
ML-DSA-44	✓	1,312	2,420	152,827	3.30	68,674	0.43
SLH-DSA-128s	✓	32	7,856	260,458,611	5,623.76	341,835	2.15
SLH-DSA-128f	✓	32	17,088	15,385,413	332.20	802,701	5.06
FN-DSA-512	✓	897	666	327,217	⚠ 7.07	62,934	0.40

bad performance

good performance

Table 1: Comparison of selected classical signature schemes with algorithms (to be) standardized by the NIST. The ⚠ indicates floating-point arithmetic, which is often susceptible to side-channel attacks [47]. The CPU cycles were taken from the SUPERCOP database and measured on an AMD Ryzen 7 7700; 8 x 3800MHz machine [48]. The PQ algorithms were not listed in their final versions yet. Therefore, we used benchmarks of preliminary versions that are closest to the final standard.

3 Merkle Tree Certificates for TLS

The motivation to create a new certificate architecture is mainly driven by the large size of PQ signatures. Unfortunately, today's WebPKI relies on signatures in various places not just limited to the CA signature in the certificate, but also for the embedded SCT for CT and possibly OCSP staples for certificate revocation. Therefore, replacing all these signatures naively results in a significant increase in bytes transferred during a TLS handshake, as Section 4.1 will show in detail. To prevent this, the Internet-Draft proposes an architecture that reduces the number of signatures where possible and instead greatly relies on hash functions. Hash functions have the advantage of being computationally lightweight, small, and PQ secure.

On a high level, the idea is to certify a batch of assertions simultaneously by building a Merkle Tree. Instead of signing each assertion individually, the CA signs only the tree head. These tree heads are distributed to Transparency Services, which serve a similar goal as the logs in CT, but additionally provide a channel for RPs to regularly update to the most recent batch tree heads.

Note that the MTC proposal does not aim to replace the certificate infrastructure as we know it today; instead, it functions as an optional optimization. Compared to today's WebPKI, it has a reduced scope and assumes more prerequisites for functioning properly. Chapter 4 elaborates on these differences; the following focuses more on the technical details of MTC.

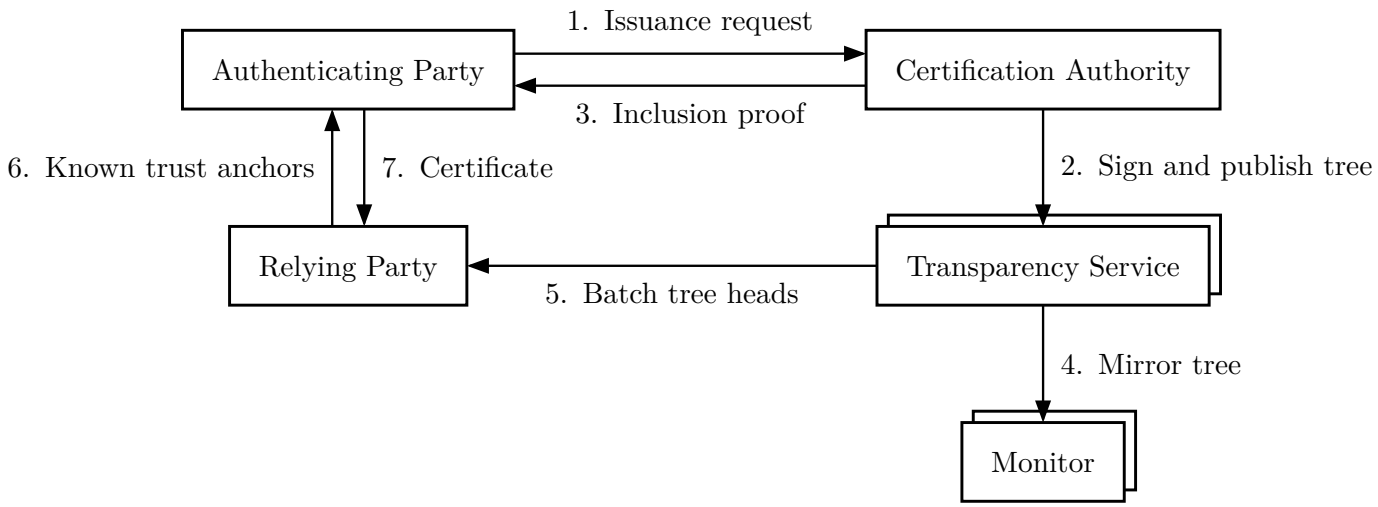

Figure 4: Overview of certificate issuance for Merkle Tree Certificates [8]

Figure 4 provides an overview of the MTC architecture. The following introduces the roles and terminology used in the figure and later sections:

- An *Authenticating Party (AP)* is an entity to be authenticated, such as a web server.
- A *Relying Party (RP)* authenticates an AP by verifying the certificate. This could be a browser, for example.
- A *Certification Authority (CA)* collects assertions from APs, validates them, and issues certificates.
- A *Transparency Service* mirrors the CAs, validates the batches, and forwards them to the RPs.
- A *Monitor* monitors the transparency services for suspicious or unauthorized certificates.
- An *assertion* is information that an AP gets certified by a CA, i.e., a public key and one or multiple domain name(s) or IP address(es). An *Abridged assertion* hashes the public key stored in an assertion to reduce the size, especially for potentially large PQ keys.
- A *batch* is a collection of assertions that are certified simultaneously.
- The *batch duration* is the time the batch spans. The authors recommend a batch duration of one hour, meaning that all assertions collected within this hour are certified simultaneously in one batch.
- A *batch tree head* is the Merkle Tree root node over all assertions of one batch.
- An *inclusion proof* is a proof that a certain assertion is included in a batch. The proof consists of the hashes required to rebuild the path up to the batch tree head.
- A *validity window* is the range of consecutive batch tree heads valid at a time.
- A *certificate* combines an assertion with an inclusion proof.

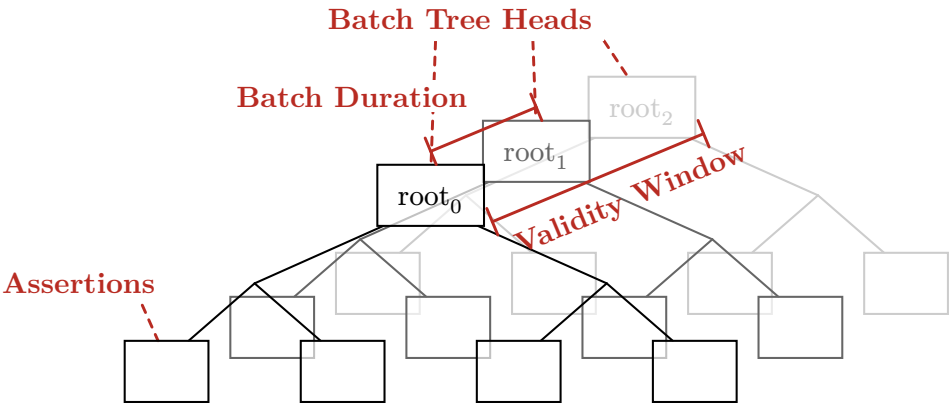

Figure 5: This figure illustrates the terms batch duration, batch tree head, validity window, and assertion

With this terminology, the following explains the certificate issuance flow depicted in Figure 4

1. First, the AP requests a certificate from the CA. Due to its frequency, this operation should be automated using the ACME protocol, for example.
2. Every time a batch becomes ready, the CA builds the Merkle Tree, signs the whole validity window, which includes the new batch tree head, with a PQ algorithm, and publishes the tree to the Transparency Services.
3. The CA also sends the inclusion proof back to the AP, which can subsequently use it to authenticate against RPs that trust this batch.
4. Monitors mirror all assertions published to the Transparency Services and check for fraudulent behavior. This can include, but is not limited to, notifying domain owners about certificates issued.
5. RPs regularly update their trust anchors to the most recent batch tree heads validated by their trusted Transparency Service(s).
6. When connecting to an AP, the RP signals which trust anchors it supports, i.e., which tree heads it trusts.
7. If the AP has a valid MTC certificate for one of the supported trust anchors, it will send this instead of a classical X.509 certificate.

The following sections elaborate on the responsibilities and objectives of the components involved.

3.1 Certification Authority

A CA is defined by the following publicly known parameters that cannot change. In particular, the Transparency Service trusts certain CAs and uses these parameters to validate the signed validity windows it receives from the CAs.

- `hash`: The hash function used to build the Merkle Tree. Currently, only SHA-256 is supported.
- `issuer_id`: A Trust Anchor Identifier (TAI) as defined in [49]. That is a relative Object Identifier (OID) under the prefix `1.3.6.1.4.1`. Organizations append their Private Enterprise Number (PEN) registered at the Internet Assigned Numbers Authority (IANA).
- `public_key`: The Transparency Services use the public key to validate the signed validity window.
- `start_time`: Is the issuance time of the first batch as POSIX timestamp [50, pp. 113].
- `batch_duration`: The time between two batches given in full seconds.
- `lifetime`: The number of seconds a batch is valid. It must be a multiple of `batch_duration`.
- The `validity_window_size` defines the number of tree heads that are valid simultaneously. It is calculated as the `lifetime` divided by the `batch_duration`.

The authors of the Internet-Draft suggest a `batch_duration` of one hour and a `lifetime` of 14 days. This results in a `validity_window_size` of 336.

As with the current certificate infrastructure, the CA is responsible for checking the assertions it certifies. In particular, the CA must verify that the AP requesting the certificate effectively controls the domain names and IP addresses to which the certificate is issued. The exact mechanism is outside the scope of this work but usually involves completing a challenge by setting a DNS record or serving a file at a specific URL.

All assertions the CA is willing to certify are accumulated in a batch. A batch is always in one of three states: *pending*, *ready*, or *issued*. A pending batch is not yet due, i.e., the `start_time` + `batch_number` · `batch_duration` is bigger than the current time. A batch in the ready state is due at the current time but has not yet been issued. This will typically be a small time frame in which the CA builds the Merkle Tree and signs the validity window. Subsequently, the batch transfers to the issued state, i.e., the CA published the signed validity window and Abridged assertions. As an invariant, all batches before the latest issued one must be issued as well; no gaps are allowed.

Every time a batch becomes ready, the CA converts all assertions it found to be valid into Abridged assertions by hashing the – possibly large – signature key in that assertion. Afterward, it builds a Merkle Tree as depicted in Figure 6. Lastly, the CA signs a `LabeledValidityWindow` that contains the null-terminated domain separator string "`Merkle Tree Crts ValidityWindow\0`", the `issuer_id`, the `batch_number`, and all Merkle Tree root hashes that are currently valid. The domain separator allows the protocol

to be extended in the future, if the CA would need to sign different structs with the same key. One example could be introducing a revocation mechanism that requires the CA to sign some data with the same key. Signing the entire validity window instead of each tree root individually has two advantages: For one, if a client or Transparency Service is behind more than one tree head, only a single signature needs to be transferred instead of multiple, which saves bandwidth and computational effort for the signature verification. The second benefit is that it complicates split-view attacks. A CA would have to keep the split view for an entire validity window instead of just a single tree head, which increases the chances of it being noticed.

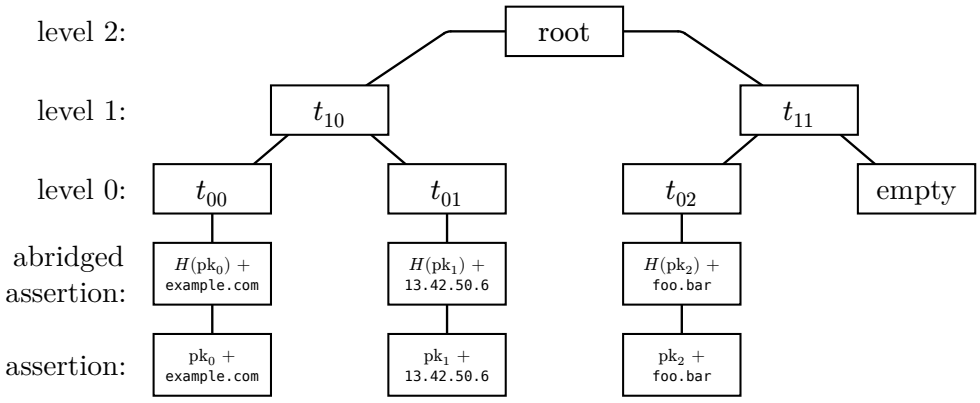

Figure 6: Example Merkle Tree for three assertions

3.2 The Role of the Transparency Service

While transparency was an afterthought in the current certificate infrastructure, it is an integral part of the MTC architecture. The main task of the Transparency Service is to validate and mirror the signed validity windows produced by CAs and serve them to RPs as well as monitors. To check a signed validity window, the Transparency Service fetches the latest signed validity window and all Abridged assertions from a CA. It then checks that the signature of the validity window matches the public key of that CA. By rebuilding the Merkle Tree from the Abridged assertions, the Transparency Service ensures that CA produced certificates for exactly what the CA serves as Abridged assertions. Due to the collision resistance of the hash function, it is computationally infeasible for the CA to secretly include another assertion, leave one out, or modify an assertion.

Conceptually, the Transparency Service is a single instance. In practice, though, it should consist of multiple services hosted by independent organizations. This reduces the chance that a CA can collude with a single Transparency Service to provide a split view. Also, the draft authors imagine that, in practice, browser vendors would run such

a Transparency Service for their product and use their update mechanism to frequently provide the most recent tree heads [8].

This link between the Transparency Service and RP includes one significant design decision: Either the Transparency Service forwards only the tree heads to the RP, or it includes the CA signature as well. Section 4.2 elaborates on how that influences how much data is distributed. Omitting the CA signature significantly reduces update bandwidth and eliminates the need for the client to perform PQ signature verification. Consequently, the RP must trust the Transparency Service to check the CA signature adequately, and it requires the RP to have a secure channel with the Transparency Service. Depending on how this channel is designed, it may require interaction with PQ signatures on the client side nevertheless. Additionally, a malicious Transparency Service could provide a split view to a client without the need to collude with a CA. At the same time, if the browser vendor runs the Transparency Service, it is anyway in a position to decide which connection to trust, even without potentially modifying the trusted roots.

3.3 Negotiation in TLS

To use the MTC architecture, the RP and AP have to negotiate using it. For that, the Internet-Draft refers to the `server_certificate_type` and `client_certificate_type` of RFC 7250 [51]. On a high level, the RP sends them as an extension of the `ClientHello` with all supported certificate types, and the AP communicates the chosen certificate type to the RP as an encrypted extension.

Additionally, the RP has to communicate to the AP which trust anchor it supports. For that, the MTC specification refers to an Internet-Draft called *TLS Trust Anchor Identifiers* [49]. It allows the RP to send the newest batch tree heads it supports to the AP, such that the AP can choose to send a certificate that the RP trusts. In particular, this allows the AP to know which MTC certificate to send during a certificate rotation.

In practice, it is not possible to send the whole list of known trust anchors to an AP for two main reasons: size and privacy. The list of all supported trust anchors is potentially large, considering that the trust anchor mechanism is not exclusively designed for MTC but explicitly also for other mechanisms, such as X.509. Assuming that just 50 CAs would participate in this mechanism with an average identifier length of four bytes, each `ClientHello` would carry 250 additional bytes, 200 for the identifiers and 50 for the encoding with length prefixes. The second concern is that a server can use the detailed information about the client for fingerprinting. Especially with the quickly changing MTC system, users might have recognizable trust stores, depending on when they pulled the latest tree heads from the Transparency Service.

To circumvent these downsides, the RP has two options. The AP can create a Service Binding (SVCB) DNS record listing all the trust anchors it supports, which is a short and not privacy-sensitive list. Based on this information, the RP can decide which trust anchor to offer to the AP during the handshake. Requiring information from the DNS complicates the deployment, but the *Encrypted Client Hello* relies on a SVCB DNS

record as well [52] and is deployed in practice already [53], [54], [55], [56]. The second option a RP has is to guess trust anchors an AP may support and do a retry if the guess was not correct. The main downside is that a retry causes an additional round trip and, therefore, higher latency.

4 Comparison of MTC with the Current WebPKI

The introduction to PKI in Section 2.2 and the explanation of MTC in Chapter 3 make it obvious that there are significant differences between these architectures. This chapter presents the results of our analysis of the differences and the advantages and disadvantages of the architectures.

The most obvious change is the significant reduction of the certificate lifetime. The authors of MTC propose a lifetime of 14 days. In contrast, as of October 2024, TLS leaf certificates for the WebPKI may be issued for at most 13 months, i.e., 398 days [15], [16]. Often, they are issued only for 90 days, which is still more than six times as long as MTC's proposed lifetime. At the same time, the validity periods of classical certificates will likely decrease further. In January 2025, Let's Encrypt announced optional support for certificates with a validity of six days [57]. In October 2024, Apple published a proposal to the CA/Browser Forum suggesting a gradual reduction of the maximum certificate lifetime to 45 days by September 2027 [58]. It is unclear if this proposal will be accepted, but the maximum certificate lifetime will only decrease in the future, possibly approaching the certificate lifetime of MTC.

Another notable difference is that the MTC draft explicitly ignores certificate revocation. This is a direct result of the short certificate lifetimes; if certificates live as long as it takes for a revocation to propagate effectively, certificate revocation is no longer necessary. Eliminating the need for a revocation mechanism is a clear improvement over the current WebPKI, which continuously suffers from ineffective revocation mechanisms [59], [60], [61]. Chrome does not check OCSP or CRLs but relies on a custom summary called *CRLSets*, which contains a small subset of all revoked certificates curated by Google [62]. In contrast, Firefox does still check OCSP responses, but the CA/Browser Forum changed its recommendation to support OCSP in their baseline requirements to optional OCSP support in version 2.0.1, effective as of March 2024 [19]. As OCSP entails high operational costs for CAs, it is likely that OCSP will further lose relevance. Let's Encrypt announced to end their OCSP support gradually in 2025 [63]. Instead, the CA/Browser forum tightens the requirements for CRLs, and Mozilla has been working on accumulating all revoked certificates into a small list called *CRLite* since 2017 [64], [65]. However, as of version 132 from October 2024, Firefox does not enable this mechanism by default.

A significant downside of MTC compared to the classical certificate infrastructure is the longer issuance times. There are two aspects to this: First, the certificate issuance itself takes up to `batch_duration` seconds, i.e., one hour assuming the default values, and second, the time the new tree heads propagate to a relevant number of RPs. The first one will not make up for the major part of the difference in practice. For both X.509 and MTC certificates, the CA must validate beforehand that the AP has effective control over the domain. This validation process often involves DNS record propagation or HTTP page propagation across multiple servers and data centers, especially for large-scale deployments [66], [67]. Therefore, classical certificate issuance can also take up to an hour, though in optimized configurations, it can work much faster. The second part, the propagation delay of new tree heads to the RPs, is more relevant. X.509 certificates are trusted by RPs immediately after issuing them. In contrast, verifying MTCs requires the RP to be up-to-date with the batch tree head. In practice, we do not expect updates from the Transparency Service to the RP to happen substantially more frequently than every six hours [68]. Therefore, the delay until a new MTC is broadly usable may be up to a few days in the worst case.

To determine how big the impact of the long issuance delay is, it is helpful to understand which circumstances require fast certificate issuance. In such a situation, the Internet-Draft assumes the existence of a fallback mechanism for fast issuance. This could be an X.509 certificate or another, future mechanism that allows for fast issuance. The drawback of large certificate chains is only temporary, until RPs updated their trust stores to incorporate the new tree heads, enabling them to utilize the size-efficient MTC mechanism again. There are two main reasons why a fast issuance is required: a new domain and an unplanned move of the domain. A scenario in which an expired certificate must be renewed quickly because of a forgotten, manual renewal is very unlikely, as MTC requires a high level of automation anyway.

In [68], L. Heimberger estimates the likelihood of those fallbacks. For that, she uses the fact that all certificates must be logged to a transparency log to be accepted by the major browser, which allows her to analyze all current and expired certificates. Heimberger divided domains into two categories: Top domains and random domains. This is interesting because the most visited websites are more likely to be well-maintained than those visited less often. The analysis she performed potentially has a high rate of false positives, but it is useful to have an idea of the order of magnitude anyway. Assuming a propagation delay of three days, the top domains have a chance of 0.004 % of hitting a fallback, while the random domains have a chance of 0.09 %. This indicates that the chance of hitting a fallback is very unlikely, and thus, the longer issuance delays will barely affect the daily operations.

4.1 Certificate Size

On a large scale, every byte saved during a TLS handshake is a relevant reduction, as a handshake takes place before almost every connection. Cloudflare published a notable

statistic regarding the number of bytes transferred from server to client [69]. Their statistic only considers QUIC connections, as they likely originate from browsers. This fits nicely, as the MTC architecture is also mainly designed for browser-like applications. For non-resumed QUIC connections, the median number of transferred bytes is 7.8 kB, and the average is 395 kB. The big difference between the median and average indicates that a few data-heavy connections heavily influence the average, while there is a high volume of small connections. This allows a rough estimate that about 40 % of the bytes transferred from the server to the client are for the certificate chain in at least half of the non-resumed QUIC connections.

Therefore, we investigate the main improvement of MTC over classical X.509 certificates in this section, namely the size reduction of the TLS handshake. Initially, we focus on the authentication-related cryptographic material exchanged during the handshake. This means we do not include the bytes that encode the domain name, key usage constraints, validity timestamps, and similar. We also ignore the bytes required to establish a shared key for the record layer, which encrypts and authenticates the payload messages. Hence, an X.509 handshake contains the following components: One signature for active handshake authentication, two signatures for SCTs, optionally one signature for an OCSP staple, one signature of the intermediate CA on the EE certificate, and one signature of the root CA on the intermediate CA. In addition, the EE and intermediate certificates contain one public key each. The root certificate is typically not sent to the RP, as it is expected to know it already. To sum this up, we counted six signatures and two public keys. The last case in Table 2, marked in yellow, is a special case. It uses KEMTLS and, therefore, sends a key encapsulation instead of a signature and stores the public key of the KEM in the certificate instead of a public key for signature generation. For our analysis, we ignore this fact, as it serves the same objective, namely actively authenticating the handshake.

Table 2 contains one optimistic and one conservative but realistic estimate for each, a PQ and non-PQ-secure setup. Additionally, it contains one setup for KEMTLS. The optimistic estimate assumes the usage of 256-bit ECDSA signatures and keys across the whole chain. About 24 % of all currently active certificates are issued for an ECDSA key, with about 53 % using a 384-bit and 47 % using a 256-bit key length. The remaining 76 % of all current EE certificates use an RSA algorithm. [17] For the root CAs stored in the Firefox root program, the numbers are a bit different. 44 % (78) use a 4096-bit RSA key, 26 % (46) use a 2048-bit RSA key, 27 % use a 384-bit ECDSA key and only 2 % (4) use a 256-bit ECDSA key [28]. Without browser telemetry data, it is not possible to judge the most common combinations just from the percentage of certificates issued and the configuration of root CAs, as there is a big imbalance between which CAs and certificates are heavily used and which are not. We tried to get an impression by manually checking the certificate chains for the top 10 domains according to Cloudflare Radar [70]. The results in Table 7 show that the landscape of used signatures is diverse. The significance is very limited, though, as five of the ten top domains do not serve a website and are purely used for Application Programming Interface (API) calls or to serve static assets like pictures and videos (root-servers.net, googleapis.com, gstatic.com,

tiktokcdn.com, amazonaws.com). The remaining five domains serve a website, but it seems likely that the majority of calls still originate from API calls, which may use different certificate chains. Moreover, the server may change the presented certificate based on a specific subdomain, the user agent, and other factors, further complicating a holistic view. Nevertheless, we are convinced that the chosen combinations represent adequate examples.

The signatures for SCTs is more uniform. RFC 6962 [11], the specification of certificate transparency, only allows either a 256-bit ECDSA or a RSA signature. The Chrome source code solely contains logs that sign with a 256-bit ECDSA [71]. Therefore, we assume a 256-bit ECDSA for the SCTs in all cases.

Table 3 shows example sizes for various parameters used for an MTC setup. From the number of columns, it becomes obvious that an MTC contains way less asymmetric cryptography. The certificate contains a single key used to protect the integrity of the handshake. Together with the length of the inclusion proof, they determine the size of the authentication related cryptography. The size of the inclusion proof logarithmically depends on the number of certificates in a batch. To estimate the size of the inclusion proof, we checked the number of active certificates for the biggest CA, Let's Encrypt. According to their statistics, there were about 420 million active certificates in October 2024, which matches observations based on certificate transparency logs [17], [72]. The logs further reveal that there are around $2 \cdot 10^9$ active certificates in total. For the first estimate of the proof length, we take Let's Encrypt's recommendation to renew certificates every 60 days. Knowing that certificates issued by Let's Encrypt are always valid for 90 days, we can deduce that there exist around $420 \cdot 10^6 \cdot \frac{60}{90} = 280 \cdot 10^6$ authenticating parties using the services of Let's Encrypt. In an MTC setup, APs are recommended to renew their certificates every ten days. Assuming that a batch lasts for one hour, each batch contains $\frac{280 \cdot 10^6}{10 \cdot 24} = 1.16 \cdot 10^6$ certificates. To accommodate this number of assertions, the Merkle Tree requires $\lceil \log_2 1.16 \cdot 10^6 \rceil = 21$ level, resulting in a proof length of 21 hashes. The current draft only allows SHA-256 as the hashing algorithm, and future iterations are unlikely to extend the length of the digest, even if the algorithm is changed. Therefore, the proof length for this scenario is $21 \cdot 32$ bytes $= 672$ bytes.

The second scenario indicates a worst-case scenario, assuming a big increase in APs or centralization to a few certificate authorities. We map the $2 \cdot 10^9$ currently active certificates to $2 \cdot 10^9$ APs, which is very conservative as it ignores the transition periods, which we considered in the first scenario. Assuming again that each AP renews its certificate every ten days and that the batch size is one hour, the above calculation results in a proof length of 736 bytes. It is interesting to realize that for every doubling of APs, the proof size grows by 32 bytes, as the tree depth grows logarithmically.

Comparing Table 2 and Table 3 reveals the size advantages of MTC, especially when using PQ algorithms. Focusing on the classical case first: In the best X.509 case, when using only 256-bit ECDSA for all signatures, MTC performs slightly worse in terms of the number of authentication bytes. While the X.509 case requires 448 authentication-related bytes, MTC requires 768 bytes, an absolute difference of 320 bytes, i.e., the MTC requires 71.43 % more bytes than the X.509 certificate. Comparing MTC to a mostly

RSA-based certificate, MTC demonstrates its advantages, as the X.509 certificate grows to 1,728 bytes. An MTC certificate with the same RSA-2048-bit algorithm and the conservative estimate of two billion active APs requires 490 bytes – or 28 % – less.

Signatures				Public Keys		∑	PQ
Handshake	**SCT + OCSP**	**EE**	**Intermediate**	**EE**	**Intermediate**		
ECDSA-256	ECDSA-256	ECDSA-256	ECDSA-256	ECDSA-256	ECDSA-256		
64	192	64	64	32	32	**448**	✗
RSA-2048	ECDSA-256	RSA-2048	RSA-4096	RSA-2048	RSA-2048		
256	192	256	512	256	256	**1,728**	✗
ML-DSA-44	ML-DSA-44	ML-DSA-44	ML-DSA-44	ML-DSA-44	ML-DSA-44		
2,420	7,260	2,420	2,420	1,312	1,312	**17,144**	✓
ML-DSA-44	ML-DSA-44	ML-DSA-44	SLH-DSA-128s	ML-DSA-44	ML-DSA-44		
2,420	7,260	2,420	7,856	1,312	1,312	**22,580**	✓
ML-KEM-768	ML-DSA-44	ML-DSA-44	ML-DSA-44	ML-KEM-768	ML-DSA-44		
1,088	7,260	2,420	2,420	1,184	1,312	**15,684**	✓

Table 2: Bytes of authentication-related cryptographic material exchanged during the TLS handshake for various algorithms in the X.509 infrastructure.

Handshake	Public Key	Proof Length	∑	PQ
ECDSA	ECDSA	280M active APs		
64	32	672	**768**	✗
RSA-2048	RSA-2048	2B active APs		
256	256	736	**1,238**	✗
ML-DSA-44	ML-DSA-44	280M active APs		
2,420	1,312	672	**4,404**	✓
ML-DSA-44	ML-DSA-44	2B active APs		
2,420	1,312	736	**4,468**	✓
ML-KEM-768	ML-KEM-768	280M active APs		
1,088	1,184	672	**2,944**	✓
ML-KEM-768	ML-KEM-768	2B active APs		
1,088	1,184	736	**3,008**	✓

Table 3: Bytes of authentication-related cryptographic material exchanged during the TLS handshake using MTC.

Moving on to the PQ algorithms, the drastic improvement of MTC shows up. Compared to the best X.509 case using only ML-DSA signatures, MTC saves 12,740 bytes or 12,676 bytes, resulting in a reduction by 74.31 % or 73.94 % depending on the number of active APs. Moreover, it seems realistic that a CA would use SLH-DSA instead of ML-DSA due to its higher security guarantees. This further increases the advantage of MTC to 80.50 % or 80.21 %, saving 18,176 or 18,112 bytes, respectively. When replacing the ML-DSA key and signature with Module-Lattice-Based Key-Encapsulation Mechanism (ML-KEM), the handshake is 1,460 bytes smaller, independent of MTC or X.509. Nevertheless, the relative gain of KEMTLS is bigger for MTC, as it exchanges fewer bytes in the baseline scenario.

In addition to size improvements related to authentication cryptography, MTC brings additional size improvements by using a new certificate format. X.509 is based on Abstract Syntax Notation One (ASN.1) and certificates are transferred in Distinguished Encoding Rules (DER) encoding. The MTC Internet-Draft defines a new certificate format called *Bikeshed* certificate. The name is meant as a placeholder, and the authors aim to replace it before it potentially becomes a standard. DER uses a type-length-value encoding, meaning that each value in the certificate explicitly has a type and length encoded. On the contrary, the encoding of MTC is more efficient because the types and lengths of fields are implicit, i.e., fixed, where possible. Besides the encoding, the Bikeshed certificate type saves bytes by omitting information that is superfluous in the new setting. The following fields are not stored in a Bikeshed certificate, that are not already covered by the size considerations above:

- Not before timestamp
- Not after timestamp
- CRL endpoint
- OCSP endpoint
- SCT timestamps and log IDs
- Key usage restrictions
- Subject and authority key identifier

To give an example: The certificate chain for `www.google.com`[1] has 2,486 bytes in the DER format. The chain contains 256-bit ECDSA, RSA-2048, and RSA-4096 bit keys and signatures. Summing them up, the authentication-related bytes transmitted in the certificate chain result in 1,248 bytes. Note that this does not contain the OCSP staple or handshake signature included in Table 2 as they are not included in the certificate chain itself. In comparison, a comparable Bikeshed certificate with a 256-bit ECDSA key would contain 704 authentication-related bytes, assuming 280 million active APs. The full certificate would be 785 bytes in size. Thus, the X.509 certificate chain has an overhead of 1,238 bytes, or 99 %, while the Bikeshed certificate has an overhead of 81 bytes, or 12 %. Even though we only analyzed a single example, it indicates that the

[1] SHA-256 fingerprint
`37:9A:80:C9:25:2C:66:A1:BB:89:D6:C0:C8:83:33:39:55:1D:E6:0F:D3:75:58:5C:F9:A3:18:37:03:57:A0:D6`

X.509/ASN.1 format produces a significant overhead, that can be reduced by introducing a new certificate format. An analysis of the certificate chains of the top websites implies that certificates are often even bigger than our example. D. Jackson investigated the size of certificate chains sent by roughly 75,000 of the Tranco top sites ranking [74]. It reveals that the 5th percentile of certificate chains is 2,308 bytes big, and the median certificate chain has 4,032 bytes. Applying existing certificate compression algorithms, this reduces to 1,619 bytes and 3,243 bytes, respectively [73]. Consequently, even though X.509 certificates require fewer authentication-related bytes if they completely rely on the size-efficient ECDSA algorithm, the inclusion of additional attributes and inefficient encoding result in MTCs being smaller in practice, not only for PQ algorithms but also for classical algorithms.

4.2 Update Mechanism Considerations

As with many optimizations, one does not get the results from Section 4.1 without a trade-off. The MTC architecture requires the RP to regularly update the tree heads it trusts, as shown in Step 5 of Figure 4. To pull the updates, the RP regularly requires a connection to the Transparency Service.

This update mechanism is the reason the MTC architecture cannot replace all X.509-based PKIs. For the TLS use cases, the updates are feasible. There are a few common use cases for TLS: For public websites served via HTTPS and visited by a browser, the browser requires a connection to the internet anyway. For public APIs, the connecting RP requires an internet connection as well. On the contrary, non-TLS use cases that rely on X.509 certificates are not covered by the MTC architecture, which is also clearly stated in the Internet-Draft. For example, smart cards could not regularly update the stored certificates, as they do not have an internet connection. Signing documents or code would not work either, as an entity verifying the signature would need to remember all trusted batch tree heads of the past. In other words, validating the signature produced by the certified key should happen temporally close to the certificate issuance, in the order of a few weeks at most.

For the use cases that allow a regular update, an important metric for the update mechanism is the amount of data that needs to be transferred from the Transparency Service to the RPs. We base this estimation on a web surfing use case with the following assumptions: We assume 150 trusted root CAs, which is somewhere between the number of CAs currently in the root store of Firefox and Chrome [28], [29]. Furthermore, we assume each CA uses a batch duration of one hour and a lifetime of 14 days, as recommended in the Internet-Draft [8, Section 5.1]. According to a recent post by D. O'Brien, working in the Chrome Security team at Google, Chrome strives for an update frequency of six hours or less [68]. Therefore, we assume six hours as the default browser update frequency for MTC tree heads. Lastly, we assume each CA to use SLH-DSA-128s to sign their validity window as the security guarantees for this algorithm are better compared to ML-DSA, which is relevant for a long-lasting key.

In addition to the basic assumptions, the update size depends on what exactly a RP pulls from the Transparency Service. The straightforward way is to regularly pull all signed validity windows of all trusted root CAs. Each validity window contains 7,856 bytes for the signature, 4 bytes for the batch number, and $24 \cdot 14 \cdot 32 = 10,752$ bytes for the three heads. Multiplying this with 150 trusted CAs, each update transfers around 2.8 megabytes, independent of the update cadence. As an optimization, the transfer could only contain the tree heads that the RP does not know yet. This reduces the bytes transferred for the tree heads to $6 \cdot 32 = 192$ bytes if a RP updates exactly every six hours. The signature would match the most recent batch number transferred, as it covers all valid batch tree heads anyway. In other words, the Transparency Service does not need to transfer one signature for each batch tree head but only one per update per CA. Together, this results in $150 \cdot (7,856 + 4 + 192) \approx 1.2$ megabyte for each update every six hours. During a day, that accumulates to 4.8 megabytes per RP. A more extreme optimization requires full trust in the update mechanism and Transparency Service. In such circumstances, the update can omit the CA signatures and save significant update bandwidth that way. For a six-hour update interval, each update contains $150 \cdot (4 + 192) = 29.4$ kilobytes, adding up to 117.6 Kilobytes per day. Compared to transferring the signatures, this saves 97.6 % in update bandwidth. The shorter the update interval, the more advantageous it is to omit the signature, as it needs to be transferred once per update.

As mentioned, omitting the CA signatures requires trust in the Transparency Service and update mechanism. It is important to note that the browser vendor likely operates the Transparency Service that the browser uses to retrieve its updates. In practice, users must trust their browser vendor in the first place to not build in any backdoors or install untrusted CAs. To mitigate potential damage from this trust relation, a browser vendor could set up a verifiable, transparent update log that all updates must be pushed to before they can be installed by the browser. A similar setup – namely firmware transparency – is described as part of the Trillian project containing software to build a transparency log, mostly used for Certificate Transparency (CT) [75]. However, the precise realization is not straightforward as the present transparency log implementations rely on classical signatures such as RSA or ECDSA. Additionally, the mechanism to bootstrap the updates requires some engineering, as it cannot be assumed that the browser knows recent MTC roots that could be used to set up a TLS-based update connection. Potentially, the update mechanism requires large X.509 certificates with PQ cryptography, at least in some cases.

All the update sizes scale linearly with the number of active CAs. This means that if we assume only 15 CAs that support the MTC architecture, we can reduce the estimates on the update size for all scenarios by 90 % to 280 kilobyte for a full update, 120 kilobytes for an update every six hours including CA signatures, and only three kilobytes for an update every six hours without the CA signatures. Table 4 provides an overview of the numbers mentioned in the above text.

Assuming only 15 CAs is likely a reasonable estimate, as only the biggest CAs are presumably willing to invest the necessary resources in such a fundamental change.

MTC serves purely as an optimization of the existing X.509 ecosystem. Thus, small CAs that do not participate can still issue working certificates. These CAs often exist because of various industry and government policies, which are generally covered by X.509 certificates. For the foreseeable feature, these policies will likely not be updated to include MTC.

	Per update		Per day	
	150 CAs	**15 CAs**	**150 CAs**	**15 CAs**
A full update for every fetch	2,800.00 kB	280.00 kB	11,200.00 kB	1,120.00 kB
Only new tree heads + signature	1,200.00 kB	120.00 kB	4,800.00 kB	480.00 kB
Only new tree heads	29.40 kB	2.94 kB	117.60 kB	11.76 kB

Table 4: Required update bandwidth from the Transparency Service to each RP

To evaluate the size estimates, it is helpful to set them in relation to the bandwidth required for browser updates nowadays. Therefore, we inspected the update size and frequency for Chrome and Firefox. Table 8 and Table 9 in the Appendix provide an overview of the recent release cadence for Chrome and Firefox, respectively. On average, Chrome releases a minor update every week and a major release roughly every four weeks. Likewise, Firefox releases a major update every four weeks, but the minor updates are slightly less frequent, with ten days in between on average. For Firefox, Table 9 also contains the update sizes for subsequent updates, averaging to 13.2 MB. At the same time, Google claims that a minor update takes approximately 3 MB to 5 MB and a major update takes approximately 10 MB to 15 MB [76]. As a result, we assume that the update bandwidth per Firefox user is about 1.3 MB on a daily average. Similarly, the update bandwidth per Chrome user is about $3 \cdot 4$ MB $+ 1 \cdot 13$ MB $= 25$ MB per 28 days, i.e., approximately 900 kB per day.

Comparing the bandwidth for current browser updates with the bandwidth for the tree heads reveals that the updates are of reasonable size. Starting with the first row of Table 4, it becomes clear that the naive update mechanism to send the full validity window including the signature is impractical for 150 active CAs. For a six-hour update cadence, the update mechanism must transfer 11 MB per user and day, a whole order of magnitude more than the present update transfers. For a smaller ecosystem with 15 CAs, even this naive approach could work out. The update capacity for the MTC update mechanism would be similarly dimensioned as for the present browser update. In other words, the update bandwidth would need to be doubled. Nevertheless, the second row shows that the straightforward optimization of distributing only data unknown to the client significantly reduces the required bandwidth. While an MTC ecosystem with 150 CAs would still require about five times more bandwidth, an ecosystem with 15 CAs adds only 50 % more update bandwidth to existing update mechanisms. Additionally, the daily update sizes in Table 4 assume that an update actually happens every six hours. In practice, the schedule might aim for this update frequency, but updates will

not happen that frequently, for example, because a computer is turned off during the night or because of a metered connection. This results in less data being transferred over the day.

The third row, which moves the signature checks to the Transparency Service, shows that the update size can be marginal. Even for 150 CAs, this scenario adds only 9 % to the regular Firefox update bandwidth. As discussed, this requires trust in the Transparency Service and update mechanism. As the browser vendor will likely operate the Transparency Service, there already exists a trust relation, such that omitting the signature checks in the client device seems reasonable.

When reflecting on the update sizes, one additional interesting observation is to compare them with the data transferred for an ordinary page visit. Therefore, we visited four pages we assumed for daily usage and measured the transferred data for an initial page load without user interaction while already logged in. The results are not representative but nevertheless serve as an indication. Loading the web pages of Gmail and Outlook transferred 6.6 MB and 10.6 MB, respectively. Loading The New York Times webpage (not logged in) and Google search transferred 2.7 MB and 0.8 MB, respectively. This demonstrates that even the worst-case scenario with a full update of 150 CAs every six hours transfers just as much data as a single page visit at Outlook and should, therefore, not cause serious problems from an end-user perspective.

4.3 Common File Structure

Besides small update sizes, it is desirable to store MTC-related data in a common directory within an Operating System (OS). Having a common place for certificates on a single machine has multiple advantages. Firstly, it reduces the number of updates required in the MTC architecture. Instead of every application pulling its updates, the OS can take care of it for various applications that depend on up-to-date tree heads. Furthermore, applications do not have to implement the update logic themselves. This saves development resources and reduces the attack surface as there exist fewer different implementations.

Nowadays, Linux-based operating systems such as Debian, RHEL, or Android store certificates in a well-known location for other programs to access [77]. We use the X.509 file structure of Debian as an inspiration to suggest a common file structure. Figure 7 outlines the file structure we propose for a RP. The absolute path (`/etc/ssl/mtc`) might vary per distribution. The structure thereafter is more interesting. We suggest that each CA lives in its subdirectory, with the Issuer ID as the directory name. The Issuer ID for MTCs is an OID so that directory names would look like `123.54.2`. The directory contains the CA parameters, the validity window's root hashes, and, optionally, the validity window's signature. As mentioned above, the signature is unnecessary if the RP trusts the Transparency Service and update mechanism. In this case, the Transparency Service is not operated by a browser vendor but may be operated by the OS vendor. Still, the argument remains that a user needs to trust its OS vendor regardless and

may, therefore, skip synchronizing the signature. In the proposed structure, the validity window contains the same data as specified in the Internet-Draft, namely the batch number and the hashes of all valid tree heads. The CA parameters contain the following information:

- The issuer ID, i.e., the OID of the CA.
- The signature scheme used to sign the validity windows.
- The public key of the CA. It must match the signature scheme.
- The proof type used for inclusion proof in the certificates. Currently, the only option is a SHA-256-based Merkle Tree inclusion proof.
- The start time of the CA, i.e., the time the CA was set up. This is required to calculate the validity of a certificate based on its batch number.
- The batch duration. This is also required to calculate a certificate's validity based on its batch number.
- The validity window size. Again, this is required to calculate a certificate's validity based on its batch number.

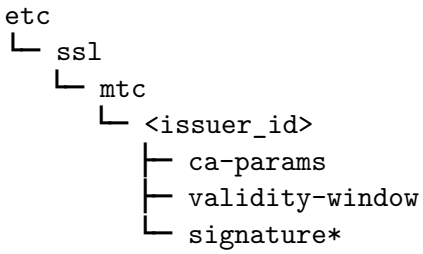

Figure 7: Proposed file structure on a RP. The signature only exists on RPs that are willing to perform the PQ signature check themselves. The public key of the CA is part of the CA parameters.

For a server setup, aiming for a homogeneous file structure is likely not as important. Nevertheless, it is worth making explicit what data is required by an AP to function in the MTC architecture. Figure 8 provides an example file structure that could be used. As for the file structure of the RP, we propose to create one directory per Issuer ID. Within that, the valid MTC certificates are stored, named as `<batch_number>.mtc`. Adhering to the recommended parameters, there are either one or two valid certificates at a time because old and new certificates overlap. For the AP, the only relevant information from the CA parameters are the start time, batch duration, and validity window size, which can be used to calculate whether a certificate has expired. The Issuer ID is included in the certificates, but it is likely handy to include it in the CA parameters, nevertheless. We propose keeping the format for the CA parameters the same for RP and AP so that they can share the same parser logic. At the same time, storing some information on the AP that is not strictly required, does not seem to entail significant downsides.

```
<some_dir>
└─ <issuer_id>
   ├─ 0.mtc
   ├─ 240.mtc
   ├─ ca-params
   └─ private-key.pem
```

Figure 8: Example file structure on an AP.

4.4 CPU Usage

The previous sections focused on the bytes that need to be transferred for a PQ secure server authentication. This section focuses on the computation required for server authentication in both systems, the classical, X.509 based and the MTC based. A low computational effort benefits client devices, even though most have sufficient resources for complex computations. Nevertheless, battery-powered devices may last longer and the available computing power can be used for different tasks. For servers, which often handle numerous TLS connections, computational efficiency is also important, as they may need to be equipped with more powerful and, therefore, expensive hardware if TLS handshakes are significantly more laborious.

We base our estimates on the SUPERCOP project. SUPERCOP is an acronym for *System for Unified Performance Evaluation Related to Cryptographic Operations and Primitives*. SUPERCOP publishes a database with benchmarks for various cryptographic primitives on various hardware configurations [78]. All performance metrics we use were measured on the same machine with an AMD Ryzen 7 7700 with eight CPU cores at 3.8 GHz. Unfortunately, there are no metrics for the final PQ signature algorithms available in the database yet. Therefore, we used the benchmarks of the corresponding, preliminary algorithm versions. For example, we used the metrics from Dilithium with level two security parameters from the third round of the NIST post-quantum competition instead of ML-DSA-44.

It quickly becomes clear that a client verifying an MTC-based server authentication requires fewer signature verifications than an X.509-based server authentication. To verify an X.509 certificate chain, the client must typically verify two SCTs, maybe an OCSP staple, one signature in the EE certificate, and one signature in the intermediate certificate. Additionally, the client must verify the handshake signature. To verify an MTC certificate, the client must traverse up the Merkle Tree up to the root node, but does not require any asymmetric cryptography, assuming the CA signature was verified either ahead of time or by the Transparency Service. Just as for the X.509 system, the client must verify the handshake signature nevertheless.

To estimate computational costs associated with the tree traversal, we assume a single hash operation with an input length of 4096-byte to create the abridged assertion. Additionally, we consider 21 or 23 hash operations, each with a 576-byte input, to

reconstruct the internal nodes up to the root. The 21 or 23 level correspond to 280 million or one billion active APs for a single CA, as described in Section 4.1.

Table 5 approximates the CPU cycles required for verifying an X.509 certificate chain with the same parameters as in Section 4.1. A first observation is that verifying RSA signatures is less computationally expensive than verifying ECDSA signatures. Possibly more surprising is that the PQ secure ML-DSA signature verification is less expensive than an ECDSA signature verification. Therefore, a certificate chain using ML-DSA for all signatures requires only 27 % of the computation of a fully ECDSA-based certificate chain. Moreover, both PQ scenarios are less computationally expensive for a client than the classical scenarios.

Nevertheless, using the MTC architecture additionally decreases the computational costs for a client. Table 6 displays the approximated CPU cycles required for validating an MTC with the same parameters as in Section 4.1. As mentioned, only a single signature verification for the handshake is required. The second variable that determines the computational cost is the number of active APs for a CA. Comparing an ECDSA X.509 certificate chain with an MTC that holds an ECDSA key, reveals that the MTC uses only 19 % of the computation the certificate chain requires. Comparing the same ECDSA certificate chain with an ML-DSA MTC with 280 million active APs demonstrates that the MTC case uses only 6.8 % of the computation the certificate chain requires. Moreover, comparing the PQ use cases with each other, we observe a reduction of 73 % to 85 % in the advantage of MTC.

Signatures				$\sum$	**PQ**
Handshake	**SCT + OCSP**	**EE**	**Intermediate**		
ECDSA-256	ECDSA-256	ECDSA-256	ECDSA-256		
255,095	765,285	255,095	255,095	**1,530,570**	❌
RSA-2048	ECDSA-256	RSA-2048	RSA-4096		
44,881	765,285	44,881	120,280	**975,327**	❌
ML-DSA-44	ML-DSA-44	ML-DSA-44	ML-DSA-44		
68,674	206,022	68,674	68,674	**412,044**	✅
ML-DSA-44	ML-DSA-44	ML-DSA-44	SLH-DSA-128s		
68,674	206,022	68,674	341,835	**685,205**	✅

Table 5: CPU cycles for verifying the server identity on a client device when using X.509 certificates. The numbers are based on the SUPERCOP database [48].

Handshake signature	Tree Traversal	$\sum$	PQ
ECDSA-256	280M active APs		
255,095	35,734	**290,829**	✗
RSA-2048	2B active APs		
44,881	38,337	**83,218**	✗
ML-DSA-44	280M active APs		
68,674	35,734	**104,408**	✓
ML-DSA-44	2B active APs		
68,674	38,337	**107,011**	✓

Table 6: CPU cycles for verifying the server identity on a client device when using MTC. The numbers are based on the SUPERCOP database [48], [79].

In terms of computation required, using MTC or X.509 does not result in any difference for the server. The signatures contained in the X.509 certificate are computed ahead of time by the CA and Transparency Logs and are treated as opaque bytes by the server. The same is true for the MTC. The certificate is mostly opaque to the server and, most importantly, sent as-is. The server only computes the handshake signature during the handshake, which is necessary in the same manner for MTC and X.509.

5 Development Insights

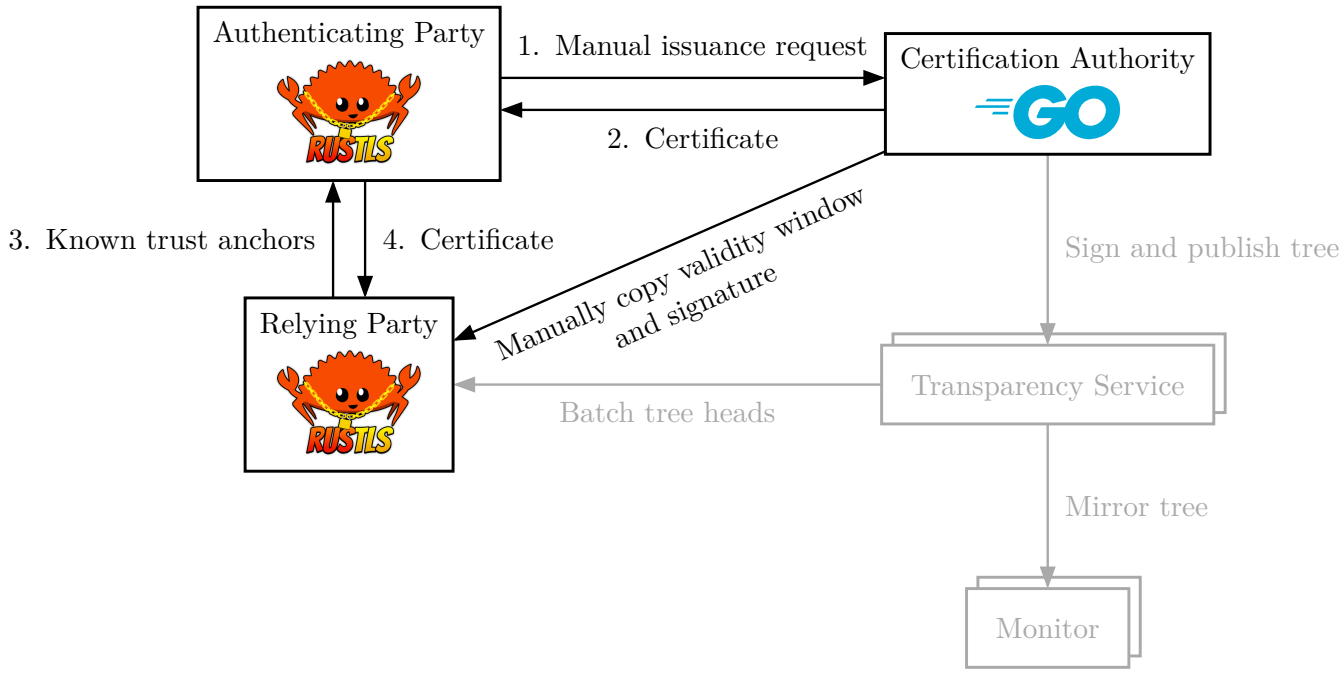

Figure 9: Overview about implemented components

The objective of this work was to establish an MTC-based TLS connection between an AP and RP for the first time and to contribute enhancements or address errors or ambiguities in the MTC Internet-Draft along the way. Figure 9 provides an overview of the implemented components and their interactions. The Transparency Service and Monitor are grayed out as we bypass them for this proof-of-concept setup and instead copy over the validity window and signature directly from the CA. As the icons indicate, we based the AP and RP on the *Rustls* project [80]. For the CA, we added to the development of an existing CA implementation that is written along the MTC specification [81]. The CA is implemented in the programming language Go.

We chose to use Rustls for multiple reasons. Firstly, writing a whole TLS implementation ourselves seems overcomplicated for this work and fails to demonstrate that the new MTC system can be integrated well with existing software. Therefore, we decided to modify an existing implementation. Rustls is a comparably modern implementation

of the TLS protocol and is cleanly implemented. One reason is that it never supported TLS 1.1 or older, which helps keep the code base clean and organized. Nevertheless, Rustls is a serious project that gained adoption in big production deployments [82], [83]. Furthermore, Rustls is written in the programming language Rust, which, in contrast to C used in other famous TLS implementations such as OpenSSL, BoringSSL, or wolfSSL, provides memory safety. Additionally, Rust's strong type system makes catching possible mistakes in the implementation comparably easy already during compilation. As a side note: Rustls does rely on optimized assembly and C implementations for cryptographic operations from the *aws-lc* library [84]. Besides the advantages of Rust itself, we also avoided using the same programming language as for the CA implementation. This requires rewriting some common parts, such as parsing the binary certificate format and checking the signature. However, this additional work ensures that neither of the implementations behaves differently from expected and actually revealed errors in the CA implementation.

The integration of MTC into Rustls necessitated numerous modifications. First, we added the negotiation mechanism for the certificate type based on RFC 7250 [51]. The negotiation mechanism relies on extensions exchanged during the `ClientHello` and `ServerHello` messages. This adaption entailed several changes to the Rustls code base, as it needs to keep state about which certificate type was negotiated. Previously, Rustls assumed X.509 certificates and related structures like stapled OCSP responses at various places. In addition to negotiating the certificate type, we implemented the negotiation mechanism for the Trust Anchor Identifiers (TAIs) as described in Section 3.3. Therefore, we extended the certificate selection logic to first match on the requested TAIs and fall back to the previously used certificate selection based on the Server Name Indication (SNI). We simplified the TAI negotiation so that the client does not preselect the TAIs it requests in the `ClientHello` based on a DNS query, simplifying the implementation and testing.

For checking X.509 certificates, Rustls uses the *WebPKI* [85] library. Similarly, we developed an MTC verifier library [86]. Specifically, it reads the supported Trust Anchors from `/etc/ssl/mtc` and verifies the CA signature over the validity window as proposed in Section 4.3. On startup, the library automatically loads all validated root tree heads into memory and also allows triggering a reload during runtime. For each TLS handshake that negotiated to use MTC as certificate type, our library

1. parses the certificate,
2. rebuilds the path through the Merkle Tree,
3. checks if the recomputed tree head matches the stored tree head,
4. and checks that the certificate falls in the latest validity window based on the stored CA parameters and batch number in the certificate.

If there are no errors, the certificate validates. Note that no signature validation is necessary during the certificate validation, as the CA signature was checked when loading the tree heads into memory. This does not mean there is no signature check during the

entire handshake. If no optimization such as KEMTLS is used, the `CertificateVerify` message still contains a signature over the messages exchanged up to this point.

Along the way, we identified some issues in the specification and CA implementation. First, we found a mismatch between the test vectors provided in the draft specification due to the use of a 16-bit instead of 8-bit length encoding for DNS names. The test vectors served as examples for assertions and abridged assertions for given inputs. We adapted the CA implementation and standard accordingly [87], [88], [89].

While we worked on this thesis, the Internet-Draft switched to using Trust Anchor Identifiers (TAIs) to identify the batches. Before, MTC contained an Issuer ID as an opaque byte string and a batch number. During this switch, the authors of the proposed standard forgot to update the definitions for the hash nodes of the Merkle Tree; we fixed this inconsistency. Additionally, we removed the batch number from the hash input, as it is included in the newly added TAI-based `batch_id`. Moreover, we introduced a more concise naming convention distinguishing `issuer_id` and `batch_id` to make it clear where only the OID part for the issuer is used and where the batch number is appended to the issuer ID [90]. Lastly, we also adapted the CA implementation to the TAIs [91], [92].

When implementing the parser for the TLS `Certificate` message, we noticed that a consistent way of embedding the certificate in the TLS message – independent of the type – keeps the parsing logic free of external state. Up to that point, the bytes of the MTC were embedded into the `Certificate` message without a length prefix. Strictly seen, a length prefix is unnecessary if the parser knows to interpret the certificate as MTC as it contains all length information needed. However, in practice, the parsing happens without knowledge of the negotiated certificate type even though the application as a whole is already aware of it. Interpreting the certificate bytes is postponed to a later stage and possibly passed on to an external library such as the Rustls WebPKI library or our MTC verification library. Therefore, allowing the parser to treat the certificate as opaque bytes with a given length prefix is advisable. The classical X.509 certificates and the `RawPublicKey` certificate type from RFC 7250 [51] already use a 24-bit length prefix. We streamlined the MTC draft specification to embed the MTC in a 24-bit length prefixed byte array as well [93].

Moreover, we replaced a pre-standard version of Dilithium with ML-DSA in the CA implementation [94]. This was required to verify the CA signature in our MTC verification library. The CA implementation used an implementation of the third round of the NIST post-quantum signature competition, which has slight incompatibilities with the final specification. As Rust did not have a library of Dilithium in the same round-three state available, the upgrade to the official ML-DSA became necessary, for which libraries exist in Go and Rust.

Further, we opened a discussion on simplifying the certificate type negotiation, but the proposal turned out not to be practical [95]. The idea was to combine the certificate type negotiation with the negotiation of the trust anchor. As the trust anchors negotiation mechanism works not only for MTC but also for X.509 and possibly other certificate types, we proposed that the peer contains the selected trust anchor in the TAI extension of the `Certificate` message. So far, the negotiation mechanism merely indicates that

one of the proposed trust anchors was selected, but it does not specify which one. By changing this indication to include the selected TAI, the peer could deduce the certificate type and therefore a separate certificate type negotiation would be superfluous. However, D. Benjamin identified some issues that might arise from the fact that not all certificates participate in the TAI negotiation mechanism [95]. Therefore, some niche cases are not properly covered. For example, if a server sends a fallback certificate that does not participate in the TAI negotiation, or of which the TAI is unknown to the client, the client does not know what certificate type to expect. As a result, the client cannot parse the certificate even if the client would accept it anyway, such as a widely accepted X.509 certificate. Consequently, we closed this discussion without additional modifications.

Lastly, we also suggested adding the file structure explained in Section 4.3 to the Internet-Draft [96]. As explained earlier, we hope to achieve a more uniform MTC ecosystem from that. As of this writing, the discussion has not started on that topic yet, so it is unclear if this proposal will be incorporated into the standard [96].

The development efforts resulted in a successful connection TLS handshake between an example client and server based on our modified Rustls version. We can use the CA implementation to create certificates, validity windows, and signatures, which we copy manually to the directories as proposed in Section 4.3. The server loads the MTC certificates next to the fallback X.509 and serves them to the client if negotiated. The client uses our MTC verifier implementation to read the available batch tree heads from the disk, validate the signatures, and validate the MTC certificates. Section B.1 compares the exchanged handshake messages on a byte level with the text in the Internet-Drafts, illustrating that our implementation adheres to the draft specifications. This was necessary as our implementation is the first, and interoperability tests with other implementations are, therefore, impossible. At the same time, we showed interoperability between the existing CA implementation in Go and our MTC verifier written in Rust.

6 Conclusion and Outlook

Constant achievements in building quantum computers endanger many asymmetric cryptography systems used today. This includes the signatures used in X.509-based certificates, which are used for server identification in TLS connections. Replacing all signatures in the X.509-based architecture with PQ-secure signature schemes results in a large expansion of the certificate sizes. This ultimately results in slower connections and more data to be transferred for each TLS handshake.

D. Benjamin, D. O'Brien, and B. Westerbaan propose Merkle Tree Certificates (MTCs) to supplement the current X.509 architecture, which reduces the number of signatures to shrink the size of certificates. As a trade, certificates cannot be used immediately and the MTC architecture requires a regular update channel between the Transparency Service and the Relying Party (RP).

In this thesis, we analyzed theoretical improvements in data transmission when introducing MTC. We showed that MTC saves about 74 % to 80 % of the bytes related to the cryptographic server authentication compared to X.509 certificates when using PQ-secure signature schemes. The actual improvement is even more significant, as MTCs use more efficient encoding and require fewer additional attributes in the certificate, such as not before / not after timestamps or CRL and OCSP endpoints. As a result, PQ-secure MTC certificates are around the same size as today's median certificate chains, which are non-PQ-secure.

In favor of small certificates, the MTC architecture introduces an update mechanism between the Transparency Service and the RP. We listed three update scenarios with either 150 or 15 CAs and argued that the new update mechanism does not harm the MTC architecture too much. Firstly, we think that 15 MTC CAs is a realistic estimate based on the fact that MTCs are not meant as a replacement but an optional optimization of the current WebPKI. Therefore, many CAs will refrain from implementing these significant changes into their operation as many of them serve small use cases that do not amortize the effort to adopt MTC. The second argument we made is that the Transparency Services are likely operated by the browser vendors, which a user must unavoidably trust anyway. Therefore, the Transparency Service can usually check the CA signatures, ultimately saving a lot of update bandwidth. This results in about 12 kB update bandwidth per day and RP. This is only a small addition compared to the about 900 kB to 1,300 kB per day for application updates in Chrome and Firefox. A single TLS handshake with MTC instead of a PQ X.509 certificate chain amortizes the daily updates. Other scenarios require a bigger update bandwidth, but we expect them to be

relevant for only a few instances (client that perform signature checks), or far in the future if at all (150 CAs).

In addition to the size analysis, we estimated the computational cost associated with X.509 and MTC. We pointed out that there is no difference for a server. Still, clients can save about 81 % to 93 % in computational cost per handshake when using classical signature algorithms and about 73 % to 85 % when using PQ-safe signature algorithms. This is mainly because clients have to perform way fewer signature verifications, which are computationally expensive. Instead, clients have to perform hash operations to rebuild the Merkle Tree. Because hash operations are much more lightweight than signature verifications, the client saves computational resources, which in turn helps with a longer battery life or frees up resources for other tasks.

To explore the practicality of the MTC architecture, we modified the TLS library *Rustls* to support MTCs. This included the negotiation mechanisms for the certificate type and the specific trust anchor, i.e., the specific MTC batch. Additionally, we developed a library that validates MTCs and integrated this into Rustls. We successfully performed a handshake between the modified client and server and analyzed it on a byte level to conform with the specification.

Overall, our work showed that the MTC architecture has the potential to mitigate or even overcompensate the performance penalty associated with introducing PQ secure algorithms for server identification. Nevertheless, there are still some open points that should be investigated further. One point could be to investigate the memory usage associated with the usage of the X.509 and MTC architectures. Another challenging task is to design an update mechanism that safely transfers the batch tree heads from the Transparency Service to the RP. To be practical, it must be reasonably small, i.e., the overhead to create a secure channel must not be too big. At the same time, this update protocol must be secured against quantum computers to create an end-to-end secure architecture. Lastly, it must not solely rely on MTCs, as it is designed to bootstrap MTC.

Finally, it is up to the major technology companies to run real-world experiments and use their telemetry collection mechanism to gather information that shows how the mostly theoretical numbers from this work translate to big deployments. From what I perceived from the community, I expect this will happen in 2025.

A Abbreviations

ACME – Automatic Certificate Management Environment
AP – Authenticating Party
API – Application Programming Interface
ASN.1 – Abstract Syntax Notation One
BGP – Border Gateway Protocol
CA – Certification Authority
CRL – Certificate Revocation List
CT – Certificate Transparency
DER – Distinguished Encoding Rules
DNS – Domain Name System
ECDSA – Elliptic Curve Digital Signature Algorithm
EE – End-Entity
FN-DSA – FFT (Fast-Fourier transform) over NTRU-Lattice-Based Digital Signature Algorithm: PQ signature algorithm, previously known as FALCON
HTTP – Hypertext Transfer Protocol
HTTPS – Hypertext Transfer Protocol Secure
IANA – Internet Assigned Numbers Authority
IETF – Internet Engineering Task Force
IMAP – Internet Message Access Protocol
IP – Internet Protocol
KEM – Key-Encapsulation Mechanism
KEMTLS: An alternative to the TLS 1.3 handshake that uses KEMs instead of signatures for server authentication
LDAP – Lightweight Directory Access Protocol
MAC – Message Authentication Code
ML-DSA – Module-Lattice-Based Digital Signature Algorithm: PQ signature algorithm, previously known as CRYSTALS-Dilithium
ML-KEM – Module-Lattice-Based Key-Encapsulation Mechanism: PQ KEM algorithm, previously known as CRYSTALS-Kyber
MTC – Merkle Tree Certificate
NIST – National Institute of Standards and Technology
OCSP – Online Certificate Status Protocol
OID – Object Identifier
OS – Operating System

PEN – Private Enterprise Number
PKI – Public Key Infrastructure
PQ – Post-Quantum

RP – Relying Party
RPKI – Resource Public Key Infrastructure
RSA – Rivest–Shamir–Adleman: Widely adopted asymmetric crypto system developed by Ron Rivest, Adi Shamir and Leonard Adleman. First published in 1977. Not secure against quantum computers.
SCT – Signed Certificate Timestamp
SHA – Secure Hash Algorithm
SLH-DSA – Stateless Hash-Based Digital Signature Algorithm: PQ signature algorithm, previously known as Sphincs+
SMTP – Simple Mail Transfer Protocol
SNI – Server Name Indication
SSL – Secure Sockets Layer
SVCB – Service Binding
TAI – Trust Anchor Identifier
TLS – Transport Layer Security
VPN – Virtual Private Network

B Appendix

B.1 Byte-Level Analysis of Handshake Messages

The implementation of the TLS negotiation mechanisms necessary for MTC cannot be tested against an independently developed peer, as there is no other implementation. Instead, we manually analyzed the messages exchanged between the client and server using Wireshark. The setup contains a simple TLS server and client, both built upon the modified Rustls version supporting MTC. The server has an MTC certificate and a fallback X.509 certificate available. The client had the CA parameters, a validity window, and the signature over this validity window available. On startup, the client validated the signature over the validity window and parsed it if the CA signature was valid. For each request, it adds a `server_certificate_type` and `trust_anchors` extension to the `ClientHello` message, specified in RFC 7250 and an ongoing Internet-Draft, respectively [49], [51].

The relevant parts of the `ClientHello` message are shown in Figure 10 and Figure 11. The top part of Figure 10 shows that Wireshark detected an extension to the `ClientHello` that it does not recognize. The identifier is 64512 or, in hexadecimal representation, `0xfc00`. This is the identifier we assigned in our implementation, as IANA has not allocated an identifier for this extension yet. The payload of this extension is shown in Figure 11. It starts with a two-byte length prefix encoding the total length of the `TrustAnchorIdentifierList` in bytes. Each list item has some part in common. It consists of a one-byte length prefix for this specific item, followed by the item, i.e., `TrustAnchorIdentifier` itself. Each starts with the sequence `3e 0c 0f`, which is the binary OID encoding for 62.12.15, the Issuer ID we used for the test CA. The byte thereafter encodes the **batch number**. Our implementation lacks an optimization in which only the most recent batch number known to the client is sent, and the server implicitly knows that all older batches are also known to the client.

Going back to the second part of Figure 10, we can see that Wireshark correctly parsed the `server_certificate_type` extension and recognized that the client advertises support for X.509 but prefers `0xe0`. The preference is expressed by the order in the list. We chose the code point `0xe0` to represent MTC in our implementation, as IANA has not yet assigned a value.

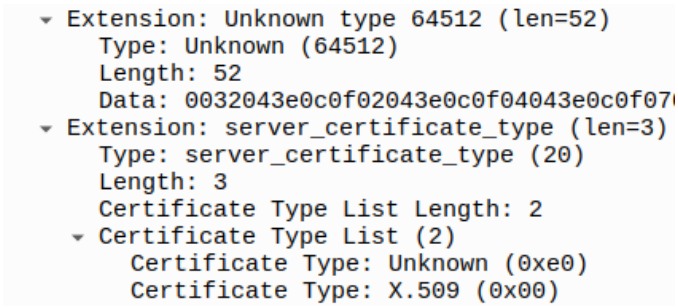

Figure 10: `trust_anchors` and `server_certificate_type` extensions parsed by Wireshark for `ClientHello`

```
0000   00 32 04 3e 0c 0f 02 04 3e 0c 0f 04 04 3e 0c 0f
0010   03 04 3e 0c 0f 01 04 3e 0c 0f 06 04 3e 0c 0f 08
0020   04 3e 0c 0f 07 04 3e 0c 0f 0a 04 3e 0c 0f 09 04
0030   3e 0c 0f 05
```

Figure 11: Extension data in `trust_anchors` extension

As a reply to the `ClientHello`, the server sends a couple of handshake messages, namely `ServerHello`, `Change Cipher Spec`, `Encrypted Extensions`, `Certificate`, `Certificate Verify`, and `Finished`. Most of these messages stay the same with MTC. The only differences are the `server_certificate_type` extension sent in the `Encrypted Extensions` message and the `Certificate` message. Figure 12 and Figure 13 show the parsed messages in Wireshark and the bytes of the `Certificate` message, respectively. The top of Figure 12 shows that the server selected the certificate type with code point `0xe0`, i.e., MTC. The second part shows that Wireshark had trouble parsing the `Certificate` message and, therefore, highlights errors in yellow. This is expected, as Wireshark does not understand the certificate type negotiation and cannot parse MTC. Therefore, Figure 13 displays the payload bytes of the `Certificate` message. It starts with a three-byte length prefix[2] followed by the Merkle Tree Certificate itself. We will not analyze the certificate itself byte-by-byte, as it is generated by the Go-based CA application and successfully parses in the Rust application, which is developed independently to a large extent. As the length prefix specifies, the MTC is `0x8c`, i.e., 140 bytes long. After that follows the two-byte length prefix for the extension list and the two-byte extension type. As in the `ClientHello` message, `0xfc00` encodes the `trust_anchors` extension. This extension exists because

[2]This is different from the Internet-Draft in version 3, the newest available as of writing. Instead, this is specified in our Pull Request `#95`, which will be published with the next draft version [93].

> If the authenticating party sends a certification path that matches the relying party's trust_anchors extension, as described in Section 4.2, the authenticating party MUST send an empty trust_anchors extension in the first CertificateEntry of the Certificate message
>
> — D. Benjamin, D. O'Brien, and B. Westerbaan [8]

After the extension type follows a two-byte length prefix that encodes the extension length, which is zero, meaning the extension is empty.

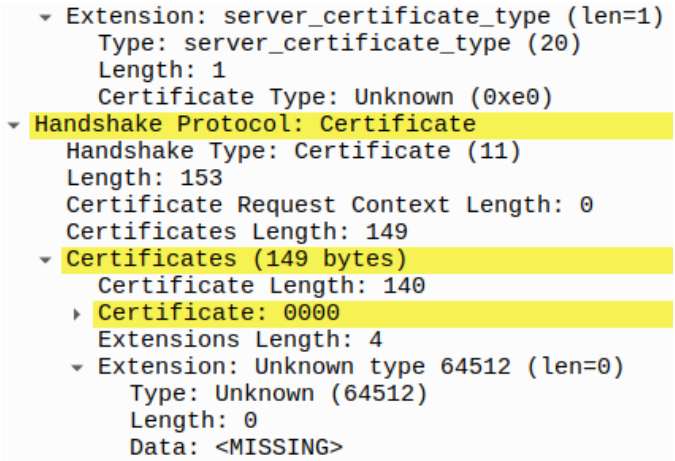

Figure 12: `server_certificate_type` in encrypted extensions and `Certificate` message parsed by Wireshark for `ServerHello`. The yellow color indicates that Wireshark detected an error decoding the certificate chain. This is expected as the message contains an MTC certificate.

```
0000   00 00 8c 00 00 00 45 04 03 00 41 04 36 ed 7b dd
0010   4a 3a 42 8c e0 59 83 64 95 43 18 5a 4f 7e d6 1e
0020   64 27 5a df b6 a4 b9 d5 06 83 b2 d2 7b 63 03 94
0030   bf ae 07 92 d8 93 db be f7 25 8d 39 28 c5 34 04
0040   67 27 5b 5c 37 ce 56 2d db fb 69 7f 00 10 00 00
0050   00 0c 00 0a 09 6c 6f 63 61 6c 68 6f 73 74 04 3e
0060   0c 0f 0a 00 2a 00 00 00 00 00 00 00 00 00 20 b5
0070   a1 82 2b 21 3f 4f 84 9d 32 f4 7e d4 de c7 60 96
0080   c2 0b 7e ca fe a8 bc 98 4e 92 02 01 e5 e5 13 00
0090   04 fc 00 00 00
```

Figure 13: Payload of `Certificate` message

B.2 Tables

Domain	Handshake	SCT	EE	Intermediate	SHA-256 fingerprint
google.com	256-bit ECDSA	2x 256-bit ECDSA	RSA-2048	RSA-4096	`37:9A:80:C9:25:2C:66:A1:BB:89:D6:C0:C8:83:33:39:55:1D:E6:0F:D3:75:58:5C:F9:A3:18:37:03:57:A0:D6`
apple.com	RSA-2048	4x 256-bit ECDSA	RSA-2048	RSA-2048	`8B:29:CD:F1:D9:4E:D6:19:13:19:BF:47:AB:05:20:16:8D:0D:21:D5:80:3E:5E:CA:A2:FE:40:A7:BA:BE:1B:AD`
facebook.com	256-bit ECDSA	3x 256-bit ECDSA	RSA-2048	RSA-2048	`AA:52:70:47:1F:CB:25:A5:47:0D:2F:04:21:52:23:2A:80:7C:EE:D5:C0:D0:F8:41:54:B4:C3:C7:EF:FA:84:B4`
microsoft.com	RSA-2048	3x 256-bit ECDSA	RSA-4096	RSA-2048	`5E:7E:E1:BF:7B:02:DE:64:07:57:84:02:E6:8F:30:E4:07:4A:4C:68:04:DA:E9:B7:12:50:70:E2:6E:A5:6B:F0`
cloudflare.com	256-bit ECDSA	2x 256-bit ECDSA	256-bit ECDSA	384-bit ECDSA	`5A:18:79:DD:30:77:B1:51:E6:96:E2:BA:6D:D6:9F:E2:77:EB:2E:BE:D6:82:D2:00:1E:A2:05:DB:94:A1:09:FA`

Table 7: Signature types in certificate chains delivered at 06.11.2024 for user agent `Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:132.0) Gecko/20100101 Firefox/132.0` at a Dutch internet connection

Major version	Release Date	Days since last release	Days since last major release
131	December 10, 2024	7	
131	December 3, 2024	14	
131	November 19, 2024	7	
131	November 12, 2024	7	28
130	November 5, 2024	7	
130	October 29, 2024	7	
130	October 22, 2024	7	
130	October 15, 2024	7	35
129	October 8, 2024	7	
129	October 1, 2024	7	
129	September 24, 2024	7	
129	September 17, 2024	7	27
128	September 10, 2024	8	
128	September 2, 2024	5	
128	August 28, 2024	7	
128	August 21, 2024	8	29
127	August 13, 2024	7	
127	August 6, 2024	7	
127	July 30, 2024	7	
127	July 23, 2024	7	
126	July 16, 2024		
Average:		7.35	29.75

Table 8: Last few Chrome release dates [97]

Version	Release Date	Days since last release	Days since last major release	Update size
133.0	November 26, 2024	14	28	17 MB
132.0.2	November 12, 2024	8		9 MB
132.0.1	November 4, 2024	6		10 MB
132.0	October 29, 2024	15	28	20 MB
131.0.3	October 14, 2024	5		9 MB
131.0.2	October 9, 2024	8		10 MB
131.0	October 1, 2024	14	28	14 MB
130.0.1	September 17, 2024	14		9 MB
130.0	September 3, 2024	14	28	18 MB
129.0.2	August 20, 2024	7		9 MB
129.0.1	August 13, 2024	7		8 MB
129.0	August 6, 2024	11	28	16 MB
128.0.3	July 26, 2024	3		8 MB
128.0.2	July 23, 2024	14		8 MB
128.0	July 9, 2024			33 MB
Average:		10	28	13.2 MB

Table 9: Last few Firefox release dates [98] and sizes of consecutive updates [99]

C Bibliography

[1] "SSL Ratios (public) - Mozilla Data Documentation." Accessed: Oct. 04, 2024. [Online]. Available: https://docs.telemetry.mozilla.org/datasets/other/ssl/reference.html

[2] D. Stebila, S. Fluhrer, and S. Gueron, "Hybrid key exchange in TLS 1.3," Internet Engineering Task Force, Internet-Draft draft-ietf-tls-hybrid-design-09, Sep. 2023. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-tls-hybrid-design/09/

[3] D. Adrian, "X25519Kyber768 key encapsulation for TLS - Chrome Platform Status" Apr. 28, 2023. Accessed: Oct. 04, 2024. [Online]. Available: https://chromestatus.com/feature/5257822742249472

[4] "Firefox Nightly 125.0a1, See All New Features, Updates and Fixes" Feb. 19, 2024. Accessed: Oct. 04, 2024. [Online]. Available: https://www.mozilla.org/en-US/firefox/125.0a1/releasenotes/

[5] "Worldwide Adoption & Usage | Cloudflare Radar." Accessed: Sep. 23, 2024. [Online]. Available: https://radar.cloudflare.com/adoption-and-usage

[6] P. Schwabe, D. Stebila, and T. Wiggers, "Post-Quantum TLS Without Handshake Signatures," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, in CCS '20. New York, NY, USA: Association for Computing Machinery, Nov. 2020, pp. 1461–1480. doi: 10.1145/3372297.3423350.

[7] D. Adrian, E. Stark, and R. Dickson, "tldr.fail" Sep. 28, 2023. Accessed: Mar. 10, 2024. [Online]. Available: https://tldr.fail/

[8] D. Benjamin, D. O'Brien, and B. Westerbaan, "Merkle Tree Certificates for TLS," Internet Engineering Task Force, Internet-Draft draft-davidben-tls-merkle-tree-certs-03, Sep. 2024. Accessed: Sep. 19, 2024. [Online]. Available: https://datatracker.ietf.org/doc/draft-davidben-tls-merkle-tree-certs-03

[9] A. J. Menezes, P. C. v. Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. Boca Raton: CRC Press, 1997. doi: 10.1201/9780429466335.

[10] I. Karonen, "Answer to "What is the purpose of using different hash functions for the leaves and internals of a hash tree?"" Mar. 17, 2012. Accessed: Dec. 04, 2024. [Online]. Available: https://crypto.stackexchange.com/a/2107

[11] B. Laurie, A. Langley, and E. Kasper, "Certificate Transparency," Internet Engineering Task Force, Request for Comments RFC6962, Jun. 2013. doi: 10.17487/RFC6962.

[12] W. S. Ford, S. Chokhani, S. S. Wu, R. V. Sabett, and C. R. Merrill, "Internet X.509 Public Key Infrastructure Certificate Policy and Certification Practices Framework," Internet Engineering Task Force, Request for Comments RFC3647, Nov. 2003. doi: 10.17487/RFC3647.

[13] G. Huston, R. Loomans, and G. G. Michaelson, "A Profile for X.509 PKIX Resource Certificates," Internet Engineering Task Force, Request for Comments RFC6487, Feb. 2012. doi: 10.17487/RFC6487.

[14] "What Is Public Key Infrastructure (PKI) & How Does It Work? | Okta" Aug. 29, 2024. Accessed: Sep. 21, 2024. [Online]. Available: https://www.okta.com/identity-101/public-key-infrastructure/

[15] R. Sleevi, "Certificate Lifetimes" Jun. 23, 2020. Accessed: Sep. 22, 2024. [Online]. Available: https://chromium.googlesource.com/chromium/src/+/b9ee8e9f75279615be7c153ce2520a40cec4c14f/net/docs/certificate_lifetimes.md

[16] "About upcoming limits on trusted certificates" Aug. 21, 2023. Accessed: Sep. 22, 2024. [Online]. Available: https://support.apple.com/en-us/102028

[17] "Merkle Town." Accessed: Sep. 21, 2024. [Online]. Available: https://ct.cloudflare.com/

[18] J. Aas, "Why ninety-day lifetimes for certificates?" Nov. 09, 2015. Accessed: Sep. 22, 2024. [Online]. Available: https://letsencrypt.org/2015/11/09/why-90-days.html

[19] I. Barreira and C. Wilson, "Baseline Requirements for the Issuance and Management of Publicly-Trusted TLS Server Certificates" Oct. 02, 2024. Accessed: Oct. 31, 2024. [Online]. Available: https://github.com/cabforum/servercert/blob/d820f37f9e1550805c210dcaf5162b7f86ccfb69/docs/BR.md

[20] S. Galperin, C. Adams, M. Myers, R. Ankney, and A. N. Malpani, "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP," Internet Engineering Task Force, Request for Comments RFC2560, Jun. 1999. doi: 10.17487/RFC2560.

[21] M. Prince, "OCSP Stapling: How CloudFlare Just Made SSL 30% Faster" Oct. 29, 2012. Accessed: Sep. 22, 2024. [Online]. Available: https://blog.cloudflare.com/ocsp-stapling-how-cloudflare-just-made-ssl-30

[22] M.-C. Koch, "E-Rezept: Regelmäßige TI-Störungen, Betroffene genervt von Informationspolitik" Mar. 12, 2024. Accessed: Nov. 27, 2024. [Online]. Available: https://www.heise.de/news/eHealth-TI-regelmaessig-gestoert-Betroffene-von-Informationspolitik-genervt-9651330.html

[23] "How Do Browsers Handle Revoked SSL/TLS Certificates?" Feb. 24, 2021. Accessed: Sep. 22, 2024. [Online]. Available: https://www.ssl.com/blogs/how-do-browsers-handle-revoked-ssl-tls-certificates/

[24] D. E. Eastlake 3rd, "Transport Layer Security (TLS) Extensions: Extension Definitions," Internet Engineering Task Force, Request for Comments RFC6066, Jan. 2011. doi: 10.17487/RFC6066.

[25] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," Internet Engineering Task Force, Request for Comments RFC8446, Aug. 2018. doi: 10.17487/RFC8446.

[26] M. Mimoso, "First Let's Encrypt Free Certificate Goes Live" Sep. 15, 2015. Accessed: Sep. 22, 2024. [Online]. Available: https://threatpost.com/first-lets-encrypt-free-certificate-goes-live/114675/

[27] R. Barnes, J. Hoffman-Andrews, D. McCarney, and J. Kasten, "Automatic Certificate Management Environment (ACME)," Internet Engineering Task Force, Request for Comments RFC8555, Mar. 2019. doi: 10.17487/RFC8555.

[28] "Mozilla Included CA Certificate List" Nov. 06, 2024. Accessed: Nov. 06, 2024. [Online]. Available: https://ccadb.my.salesforce-sites.com/mozilla/IncludedCACertificateReport

[29] *Chrome root store*. Version: 17. Accessed: Nov. 06, 2024. [Online]. Available: https://chromium.googlesource.com/chromium/src/+/main/net/data/ssl/chrome_root_store/root_store.md

[30] N. Van Der Meulen, "DigiNotar: Dissecting the First Dutch Digital Disaster," *Journal of Strategic Security*, vol. 6, no. 2, pp. 46–58, Jun. 2013, doi: 10.5038/1944-0472.6.2.4.

[31] "How CT Works : Certificate Transparency." Accessed: Mar. 23, 2024. [Online]. Available: https://certificate.transparency.dev/howctworks/

[32] M. Hutchinson, A. Barisani, A. Cutter, and F. Valsorda, *From Witnessing to Transparent Ecosystems*, (Oct. 09, 2024). Accessed: Dec. 02, 2024. [OnlineVideo]. Available: https://www.youtube.com/watch?v=v9cgvZXRRZU

[33] A. Cutter, J. Hou, M. Hutchinson, R. Ng, P. Boneff, and J. Kukkonen, "transparency-dev/armored-witness" Nov. 25, 2024. Accessed: Dec. 02, 2024. [Online]. Available: https://github.com/transparency-dev/armored-witness/tree/3cd18d29f76933a4bc7e0093b04e4d93a299ce8e

[34] D. O'Brien, "Certificate Transparency Enforcement in Google Chrome" Jul. 02, 2018. Accessed: Sep. 22, 2024. [Online]. Available: https://groups.google.com/a/chromium.org/g/ct-policy/c/wHILiYf31DE/m/iMFmpMEkAQAJ

[35] "Apple's Certificate Transparency policy – Apple Support (UK)" Nov. 30, 2023. Accessed: Sep. 22, 2024. [Online]. Available: https://support.apple.com/en-gb/103214

[36] R. Oppliger, *SSL and TLS: Theory and Practice*, Third edition. in Artech House information security and privacy series. Artech House, 2023. ISBN: 978-1-68569-015-1.

[37] D. Benjamin, "Modernizing Transport Security" Oct. 15, 2018. Accessed: Sep. 23, 2024. [Online]. Available: https://security.googleblog.com/2018/10/modernizing-transport-security.html

[38] M. Thomson, "Removing Old Versions of TLS" Oct. 15, 2018. Accessed: Sep. 23, 2024. [Online]. Available: https://blog.mozilla.org/security/2018/10/15/removing-old-versions-of-tls

[39] K. Pflug, "Modernizing TLS connections in Microsoft Edge and Internet Explorer 11" Oct. 15, 2018. Accessed: Sep. 23, 2024. [Online]. Available: https://blogs.windows.com/msedgedev/2018/10/15/modernizing-tls-edge-ie11/

[40] C. Wood, "Deprecation of Legacy TLS 1.0 and 1.1 Versions" Oct. 15, 2018. Accessed: Sep. 23, 2024. [Online]. Available: https://webkit.org/blog/8462/deprecation-of-legacy-tls-1-0-and-1-1-versions/

[41] M. Fischlin, F. Günther, B. Schmidt, and B. Warinschi, "Key Confirmation in Key Exchange: A Formal Treatment and Implications for TLS 1.3," in *2016 IEEE Symposium on Security and Privacy (SP)*, May 2016, pp. 452–469. doi: 10.1109/SP.2016.34.

[42] National Institute of Standards and Technology, "Module-Lattice-Based Digital Signature Standard," National Institute of Standards and Technology, FIPS204, Aug. 2024. doi: 10.6028/NIST.FIPS.204.

[43] National Institute of Standards and Technology, "Stateless Hash-Based Digital Signature Standard," National Institute of Standards and Technology, FIPS205, Aug. 2024. doi: 10.6028/NIST.FIPS.205.

[44] J.-P. Aumasson *et al.*, "SPHINCS+ | Submission to the NIST post-quantum project, v.3.1," Jun. 2022. Accessed: Nov. 27, 2024. [Online]. Available: https://sphincs.org/data/sphincs+-r3.1-specification.pdf

[45] E. Karabulut and A. Aysu, "FALCON Down: Breaking FALCON Post-Quantum Signature Scheme through Side-Channel Attacks," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, Dec. 2021, pp. 691–696. doi: 10.1109/DAC18074.2021.9586131.

[46] S. Zhang, X. Lin, Y. Yu, and W. Wang, "Improved Power Analysis Attacks on Falcon," in *Advances in Cryptology – EUROCRYPT 2023*, C. Hazay and M. Stam, Eds., Cham: Springer Nature Switzerland, 2023, pp. 565–595. doi: 10.1007/978-3-031-30634-1_19.

[47] B. Westerbaan, "The state of the post-quantum Internet" Mar. 05, 2024. Accessed: Mar. 13, 2024. [Online]. Available: https://blog.cloudflare.com/pq-2024

[48] D. J. Bernstein and T. Lange, "Measurements of public-key signature systems on one machine: amd64; Zen 4 (a60f12); 2023 AMD Ryzen 7 7700; 8 x 3800MHz; hertz, supercop-20241022." Accessed: Dec. 20, 2024. [Online]. Available: http://bench.cr.yp.to/results-sign/amd64-hertz.html

[49] B. Beck, D. Benjamin, and D. O'Brien, "TLS Trust Anchor Identifiers," Internet Engineering Task Force, Internet-Draft draft-beck-tls-trust-anchor-ids-02, Oct. 2024. Accessed: Nov. 15, 2024. [Online]. Available: https://datatracker.ietf.org/doc/draft-beck-tls-trust-anchor-ids-02

[50] "IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(TM)) Base Specifications, Issue 7," *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)*, pp. 1–3951, Jan. 2018, doi: 10.1109/IEEESTD.2018.8277153.

[51] P. Wouters, H. Tschofenig, J. I. Gilmore, S. Weiler, and T. Kivinen, "Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)," Internet Engineering Task Force, Request for Comments RFC7250, Jun. 2014. doi: 10.17487/RFC7250.

[52] E. Rescorla, K. Oku, N. Sullivan, and C. A. Wood, "TLS Encrypted Client Hello," Internet Engineering Task Force, Internet-Draft draft-ietf-tls-esni-22, Sep. 2024. Accessed: Nov. 26, 2024. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-tls-esni

[53] "Firefox 119.0, See All New Features, Updates and Fixes" Oct. 24, 2023. Accessed: Nov. 26, 2024. [Online]. Available: https://www.mozilla.org/en-US/firefox/119.0/releasenotes/

[54] "TLS Encrypted Client Hello · Issue #46 · WebKit/standards-positions" Aug. 12, 2022. Accessed: Nov. 26, 2024. [Online]. Available: https://github.com/WebKit/standards-positions/issues/46

[55] A. van der Mandele, A. Ghedini, C. Wood, and R. Mehra, "Encrypted Client Hello - the last puzzle piece to privacy" Sep. 29, 2023. Accessed: Nov. 26, 2024. [Online]. Available: https://blog.cloudflare.com/announcing-encrypted-client-hello

[56] "TLS Encrypted Client Hello (ECH) - Chrome Platform Status" Dec. 12, 2023. Accessed: Nov. 26, 2024. [Online]. Available: https://chromestatus.com/feature/6196703843581952

[57] J. Aas, "Announcing Six Day and IP Address Certificate Options in 2025" Jan. 16, 2025. Accessed: Jan. 24, 2025. [Online]. Available: https://letsencrypt.org/2025/01/16/6-day-and-ip-certs/

[58] C. Wilson, "SC-081: Introduce Schedule of Reducing Validity and Data Reuse Periods by clintwilson" Oct. 10, 2024. Accessed: Oct. 30, 2024. [Online]. Available: https://github.com/cabforum/servercert/pull/553

[59] A. Gable, "A New Life for Certificate Revocation Lists" Sep. 07, 2022. Accessed: Oct. 31, 2024. [Online]. Available: https://letsencrypt.org/2022/09/07/new-life-for-crls/

[60] "GRC's | Chrome's CRLSet Effectiveness Evaluation" May 08, 2014. Accessed: Oct. 31, 2024. [Online]. Available: https://www.grc.com/revocation/crlsets.htm

[61] MorrocMaster, "Firefox - The only browser doing certificate revocation checks right" Mar. 10, 2024. Accessed: Oct. 31, 2024. [Online]. Available: https://www.reddit.com/r/firefox/comments/1bb89vl/firefox_the_only_browser_doing_certificate/

[62] "CRLSets." Accessed: Oct. 31, 2024. [Online]. Available: https://www.chromium.org/Home/chromium-security/crlsets/

[63] J. Aas, "Ending OCSP Support in 2025" Dec. 05, 2024. Accessed: Jan. 24, 2025. [Online]. Available: https://letsencrypt.org/2024/12/05/ending-ocsp/

[64] J. Larisch, D. Choffnes, D. Levin, B. M. Maggs, A. Mislove, and C. Wilson, "CRLite: A Scalable System for Pushing All TLS Revocations to All Browsers," in *2017 IEEE Symposium on Security and Privacy (SP)*, May 2017, pp. 539–556. doi: 10.1109/SP.2017.17.

[65] J. C. Jones, "The End-to-End Design of CRLite" Jan. 09, 2020. Accessed: Nov. 06, 2024. [Online]. Available: https://blog.mozilla.org/security/2020/01/09/crlite-part-2-end-to-end-design

[66] "Challenge Types" Feb. 13, 2023. Accessed: Oct. 31, 2024. [Online]. Available: https://letsencrypt.org/docs/challenge-types/

[67] "How long does it take to issue an SSL certificate?." Accessed: Oct. 31, 2024. [Online]. Available: https://support.dnsimple.com/articles/how-long-to-issue-ssl-certificate/

[68] L. Heimberger, B. Westerbaan, and D. O'Brien, "MTC Fallback Estimates" Aug. 26, 2024. Accessed: Oct. 31, 2024. [Online]. Available: https://github.com/davidben/merkle-tree-certs/issues/89

[69] B. Westerbaan and L. Valenta, "A look at the latest post-quantum signature standardization candidates" Nov. 07, 2024. Accessed: Jan. 24, 2025. [Online]. Available: https://blog.cloudflare.com/another-look-at-pq-signatures/

[70] "Domain Rankings Worldwide | Cloudflare Radar" Nov. 06, 2024. Accessed: Nov. 06, 2024. [Online]. Available: https://radar.cloudflare.com/domains

[71] *log_list.json - Chromium Code Search*. Version: ddf98ab. Accessed: Nov. 14, 2024. [Online]. Available: https://source.chromium.org/chromium/chromium/src/+/ddf98abcbd6101328e98ffd05ab07d6e8741f518:components/certificate_transparency/data/log_list.json

[72] "Let's Encrypt Stats." Accessed: Nov. 07, 2024. [Online]. Available: https://letsencrypt.org/stats/

[73] D. Jackson, "Abridged Compression for WebPKI Certificates," Internet Engineering Task Force, Internet-Draft draft-ietf-tls-cert-abridge-02, Sep. 2024. Accessed: Nov. 25, 2024. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-tls-cert-abridge-02

[74] V. Le Pochat, T. Van Goethem, S. Tajalizadehkhoob, M. Korczynski, and W. Joosen, "Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation," in *Proceedings 2019 Network and Distributed System Security Symposium*, San Diego, CA: Internet Society, 2019. doi: 10.14722/ndss.2019.23386.

[75] "Trilian - Firmware Transparency | google/trillian-examples" Nov. 13, 2024. Accessed: Nov. 13, 2024. [Online]. Available: https://github.com/google/trillian-examples/tree/ded41b7d3a733970b3a40db93e16f692ee9ea62f/binary_transparency/firmware

[76] "Manage Chrome updates (Windows) - Chrome Enterprise and Education Help." Accessed: Dec. 11, 2024. [Online]. Available: https://support.google.com/chrome/a/answer/6350036?hl=en#zippy=%2Chow-often-are-google-update-tasks-performed%2Cwhat-size-are-chrome-browser-updates%2Chow-often-does-google-update-check-for-updates

[77] F. Valsorda, *root_linux.go / golang/go*. Version: b01cb72. Accessed: Nov. 13, 2024. [Online]. Available: https://github.com/golang/go/blob/c759ea7471de5a62c88325981e8c86beab78bed8/src/crypto/x509/root_linux.go

[78] D. J. Bernstein and T. Lange, "SUPERCOP." Accessed: Dec. 26, 2024. [Online]. Available: https://bench.cr.yp.to/supercop.html

[79] D. J. Bernstein and T. Lange, "Measurements of hash functions on one machine: amd64; Zen 4 (a60f12); 2023 AMD Ryzen 7 7700; 8 x 3800MHz; hertz, super-

cop-20241022." Accessed: Dec. 20, 2024. [Online]. Available: http://bench.cr.yp.to/results-hash/amd64-hertz.html
[80] J. Birr-Pixton *et al.*, "rustls/rustls: A modern TLS library in Rust" Oct. 17, 2024. Accessed: Nov. 15, 2024. [Online]. Available: https://github.com/rustls/rustls
[81] B. Westerbaan and M. Pohl, "Merkle Tree Certificates for TLS" Nov. 15, 2024. [Online]. Available: https://github.com/bwesterb/mtc
[82] J. Aas, "More Memory Safety for Let's Encrypt: Deploying ntpd-rs" Jun. 24, 2024. Accessed: Nov. 15, 2024. [Online]. Available: https://letsencrypt.org/2024/06/24/ntpd-rs-deployment/
[83] J. Aas, "Rustls Gains OpenSSL and Nginx Compatibility" May 08, 2024. Accessed: Nov. 15, 2024. [Online]. Available: https://www.memorysafety.org/blog/rustls-nginx-compatibility-layer/
[84] D. Benjamin *et al.*, "aws-lc." Accessed: Jan. 27, 2024. [Online]. Available: https://github.com/aws/aws-lc
[85] B. Smith *et al.*, "rustls/webpki: WebPKI X.509 Certificate Validation in Rust" Nov. 02, 2024. Accessed: Nov. 15, 2024. [Online]. Available: https://github.com/rustls/webpki
[86] M. Pohl, "pohlm01/mtc-verifier" Nov. 15, 2024. Accessed: Nov. 15, 2024. [Online]. Available: https://github.com/pohlm01/mtc-verifier
[87] M. Pohl, "Fix prefix length for DNSName by pohlm01 · Pull Request #2 · bwesterb/mtc" Sep. 06, 2024. Accessed: Nov. 19, 2024. [Online]. Available: https://github.com/bwesterb/mtc/pull/2
[88] M. Pohl, "Adopt tests for fixed length prefix in #2 by pohlm01 · Pull Request #4 · bwesterb/mtc" Sep. 11, 2024. Accessed: Nov. 19, 2024. [Online]. Available: https://github.com/bwesterb/mtc/pull/4
[89] M. Pohl, "Corrects test vectors based on a wrong length encoding of the DNS name by pohlm01 · Pull Request #90 · davidben/merkle-tree-certs" Sep. 12, 2024. Accessed: Nov. 19, 2024. [Online]. Available: https://github.com/davidben/merkle-tree-certs/pull/90
[90] M. Pohl, "Consistently use TAI instead of issuer_id by pohlm01 · Pull Request #91 · davidben/merkle-tree-certs" Sep. 15, 2024. Accessed: Nov. 19, 2024. [Online]. Available: https://github.com/davidben/merkle-tree-certs/pull/91
[91] M. Pohl, "Adopt `TrustAnchorIdentifiers` from v03 of MTC draft by pohlm01 · Pull Request #5 · bwesterb/mtc" Sep. 11, 2024. Accessed: Nov. 19, 2024. [Online]. Available: https://github.com/bwesterb/mtc/pull/5
[92] M. Pohl, "Adopt to davidben/merkle-tree-certs#91 by pohlm01 · Pull Request #8 · bwesterb/mtc" Sep. 20, 2024. Accessed: Nov. 19, 2024. [Online]. Available: https://github.com/bwesterb/mtc/pull/8
[93] M. Pohl, "Use `<1..2^24-1>` encoding for all certificate types by pohlm01 · Pull Request #95 · davidben/merkle-tree-certs" Oct. 14, 2024. Accessed: Nov. 19, 2024. [Online]. Available: https://github.com/davidben/merkle-tree-certs/pull/95

[94] "Replace Dilithium with ML-DSA by pohlm01 · Pull Request #9 · bwesterb/mtc" Nov. 15, 2024. Accessed: Nov. 19, 2024. [Online]. Available: https://github.com/bwesterb/mtc/pull/9

[95] M. Pohl, "Adaptions to supersede `server/client_certificate_type` extensions · Issue #76 · davidben/tls-trust-expressions" Oct. 15, 2024. Accessed: Nov. 19, 2024. [Online]. Available: https://github.com/davidben/tls-trust-expressions/issues/76

[96] M. Pohl, "Default file structure for RP and AP · Issue #97 · davidben/merkle-tree-certs" Oct. 30, 2024. Accessed: Nov. 19, 2024. [Online]. Available: https://github.com/davidben/merkle-tree-certs/issues/97

[97] "Chrome Releases." Accessed: Dec. 11, 2024. [Online]. Available: https://chromereleases.googleblog.com/search/label/Desktop%20Update?updated-max=2024-12-11T10:45:00-08:00

[98] "Mozilla Firefox Release Notes." Accessed: Dec. 11, 2024. [Online]. Available: https://www.mozilla.org/en-US/firefox/releases/

[99] "Directory Listing: /pub/firefox/releases/." Accessed: Dec. 11, 2024. [Online]. Available: https://ftp.mozilla.org/pub/firefox/releases/