Towards Sustainable Evolution for the TLS Public-Key Infrastructure

Taeho Lee
ETH Zürich
kthlee@inf.ethz.ch

Christos Pappas
ETH Zürich
pappasch@inf.ethz.ch

Pawel Szalachowski
SUTD
pawel@sutd.edu.sg

Adrian Perrig
ETH Zürich
aperrig@inf.ethz.ch

ABSTRACT

Motivated by the weaknesses of today's TLS public-key infrastructure (PKI), recent studies have proposed numerous enhancements to fortify the PKI ecosystem. Deploying one particular enhancement is no panacea, since each one solves only a subset of the problems. At the same time, the high deployment barrier makes the benefit-cost ratio tilt in the wrong direction, leading to disappointing adoption rates for most proposals.

As a way to escape from this conundrum, we propose a framework that supports the deployment of multiple PKI enhancements, with the ability to accommodate new, yet unforeseen, enhancements in the future. To enable mass adoption, we enlist the cloud as a "centralized" location where multiple enhancements can be accessed with high availability. Our approach is compatible with existing protocols and networking practices, with the ambition that a few changes will enable sustainable evolution for PKI enhancements. We provide extensive evaluation to show that the approach is scalable, cost-effective, and does not degrade communication performance. As a use case, we implement and evaluate two PKI enhancements.

1 INTRODUCTION

Encrypted traffic is increasing rapidly as the Transport Layer Security (TLS) protocol is becoming ubiquitous. For instance, the "Let's Encrypt" project [1] acts as an open Certificate Authority (CA), issuing free certificates for any domain; after about a year of operation, it supports more than 22 million active certificates [2, 3]. Major hosting providers like Amazon and CloudFlare also offer their clients easy and free access to TLS certificates [4, 5]. Moreover, new mobile and Internet-of-Things (IoT) devices have functional TLS stacks, further fueling the rapid increase of TLS clients.

ASIA CCS '18, June 4–8, 2018, Incheon, Republic of Korea
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5576-6/18/06…$15.00
https://doi.org/10.1145/3196494.3196520

Amidst its surging popularity, securing the TLS ecosystem has become increasingly important. In recent years, multiple PKI enhancements (PKIEs) have been proposed that improve certain weaknesses of the TLS PKI. For example, to foster the transparency and accountability of the PKI, log-based approaches suggest publicly verifiable logs that monitor CAs [6, 7]. Building on this idea, recent proposals additionally require multiple trusted entities to assert the validity of a certificate [8, 9]. Revocation systems keep and disseminate information for non-expired certificates that have been invalidated (e.g., due to leaked private keys) [10–15]. Another recent proposal enables domains to express security policies about their TLS certificates and to detect misbehavior by publishing policies to public logs [16].

Although the literature is ripe with PKIE proposals, we can hardly say that they had much of an impact in practice. Revocation systems are indicative examples of enhancements with disappointing deployment rates: OCSP stapling [14] has been considered a promising revocation system, but only 3% of all certificates are served by servers deploying it [17]. Certificate Revocation Lists (CRLs) is the most well-established way to disseminate revocation information, yet it is disabled in all mobile browsers and most desktop browsers [17]. The main reasons behind the observed stagnation include the strain of updating multiple entities (clients, and/or servers, and/or CAs), the infrastructure and management costs of deployment, the inability of some proposals to scale to today's needs, and performance penalties to communication sessions.

Only recently, Google's Certificate Transparency has moved towards higher rates of adoption [18]. This is feasible only thanks to Google's unique position in the TLS ecosystem with its dominance in the web-browser market.[1] However, the fate of which PKIEs get deployed should not lie in the hands of a few stakeholders; this is crucial given that there is no magic bullet for all threats nor consensus on which threats are more alarming [19].

Given today's situation, it is hard to envision a healthy PKI ecosystem with multiple enhancements that can be used according to the needs of all involved parties (clients, domains, and CAs). The need to "design for choice" [20, 21] and construct evolvable systems [22–24] is widely accepted in the networking community, yet it has not been applied to the PKI ecosystem.

[1]http://www.w3schools.com/browsers/

To escape from today's conundrum, we propose a different approach to deploy and operate existing and future PKIEs. Our approach pushes the deployment of PKI Enhancements to the Cloud (PEC). Deploying PKIEs as a cloud service can drive mass adoption since they can be accessed with high availability. Furthermore, our approach requires a *one-time change* for web servers, yet support is enabled for existing and future PKIEs; a client-only deployment model is also possible if servers are not updated. We think that our approach hits a sweet spot in the benefit-cost ratio for PKIE deployment and will thus help in overcoming today's stagnation.

In summary, we make the following contributions. First, we present PEC, a generic framework that is incrementally deployable and enables the adoption of multiple PKI enhancements in a secure, scalable, and cost-efficient way. Second, we implement two prominent PKIEs as use cases: we optimize and improve the security properties offered by the default deployment model of Certificate Transparency [6], and we implement a recent revocation proposal [15].

Furthermore, we have fully implemented the cloud service that hosts PKIEs and we have extended the OpenSSL library to support PEC. We provide extensive evaluation of PEC, showing that it introduces negligible performance overhead—the average latency inflation of the TLS handshake is less than 2%. Additionally, we show that PEC can drive mass adoption for PKIEs in a cost-effective way, with an estimate of about $100K per month for processing all TLS connections in the U.S.

2 LESSONS LEARNED FROM RELATED WORK

In TLS, certificates are used by the clients to verify the identities of servers. Certificates are issued by trusted CAs and sent to the clients by the servers during the TLS handshake.[2] Based on the certificate, the client has to decide whether the connection is trustworthy before sending data.

To improve the security of the TLS PKI, many PKI enhancements (PKIEs) that try to increase the trustworthiness of certificates have been proposed. In general, PKIEs share a common theme in that they create verifiable assertions about certificates, and that these assertions are inspected by the clients during the handshake. From a deployment perspective, PKIEs differ in how these assertions are delivered to the clients. We briefly present the design space of deployment models and explain their main difficulties.

In server-driven deployments, the server periodically fetches information about the status of its certificate from a PKIE provider. Then, the server sends this information along every new TLS session setup (e.g., as in OCSP stapling [14]). The low popularity of the model is attributed to the reluctance of administrators to reconfigure their servers given the uncertain benefits of deploying one PKIE. Furthermore, administrators must reconfigure their servers for every new enhancement.

In client-driven deployments, the client can be pre-loaded with server information (e.g., revocations of certificates) and updated periodically to keep the information fresh. For example, browser vendors supply clients with a limited revocation list (e.g., CRLSet [11] and OneCRL [12]). However, this approach comes with shortcomings: 1) the ever-increasing number of certificates make the PKIE databases large and partial information introduces security holes, and 2) hardware-constrained devices may not implement the additional functionality.

[2]We focus on the most common setting in which only the server is authenticated to the client.

Alternately, clients can directly query the PKIE provider regarding the server's information. This model is adopted by schemes like CRL [10], OCSP [13], notary systems [25, 26], and some log-based systems [27, 28]. Other client-driven proposals, such as TrustBase [29] and CertShim [30], focus mainly on the deployment strategy of new PKIEs at the client's operating system in a way that is transparent to legacy applications. The client-driven model may introduce latency during connection setup, since the client can query the PKIE provider only after receiving the server's information in the handshake. Additionally, this approach harms clients' privacy as they have to contact an untrusted third party and disclose the domains that they contact.

A recent proposal, RITM, suggests to implement PKIE services as a network functionality [15]. A middlebox is pre-loaded with necessary PKIE information and monitors traffic for new TLS connections. Based on the server's information in the handshake, the middlebox sends the additional PKIE information to the client. This model does not require server reconfiguration, does not introduce storage or processing overheads for the clients, and does not noticeably increase the latency of the connection. However, it introduces substantial infrastructure costs (provisioning for peak network utilization) and management costs in terms of personnel needed to administer the middleboxes [31]. Furthermore, misconfigurations and physical failures are common, leading to availability problems. Finally, this model lacks the property of evolvability since it offers a static set of PKIEs and introducing a new PKIE requires massive upgrades.

3 PEC OVERVIEW

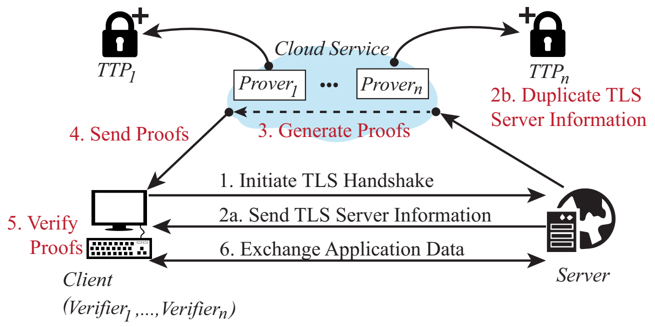
The health of TLS relies upon the diversity of PKIEs and their assertions about domain certificates (e.g., revocation status, presence in certificate logs, domain policies). To this end, we define a unified framework that, once deployed, can support multiple PKIEs.

3.1 Design Choices

We outline the design decisions that we have made based on the lessons learned from related work (Section 2).

1) Our starting point is to push the deployment location of PKIEs to the cloud. The idea of outsourcing network functionality is not novel [31]; it has already been used for security purposes (e.g., firewalls, intrusion detection systems, and VPN gateways) and performance purposes (e.g., load balancers and WAN optimizers). Following this trend, we build on the appealing properties of cloud deployments: flexibility to scale up or down based on traffic needs, pay-by-use with low personnel/management costs, and instantaneous updates. Furthermore, cloud deployments boost evolution by providing flexibility for new services with low prospective costs.

2) Our approach provides flexibility to clients in that they can explicitly indicate which PKIE services they request for every connection and which cloud service they contact. We highlight two points: i) Flexibility does not come at a higher deployment cost for servers:


TTP$_1$
Cloud Service
Prover$_1$
Prover$_n$
TTP$_n$
2b. Duplicate TLS Server Information
4. Send Proofs
3. Generate Proofs
1. Initiate TLS Handshake
2a. Send TLS Server Information
6. Exchange Application Data
5. Verify Proofs
Client (Verifier$_1$,...,Verifier$_n$)
Server
Figure 1: System Model.

adopting our framework is a *one-time change* for servers since they do not have to be aware of each requested PKIE. ii) Furthermore, from a deployment perspective, PEC is deployable without requiring server upgrades. However, there are additional benefits (e.g., lower latency) when servers also support PEC.

3) Only the server's side of the TLS handshake is sent to the cloud service. We observe that only the server-related information is needed to decide if a session should (not) be accepted, thus only this is sent to the service; application data is exchanged over the normal client-server path. This minimizes the traffic volume sent to the cloud.

As a consequence of our design choices, new PKIEs can be introduced as a service, without the strain of updating servers many times. At the same time, clients can choose which PKIEs they want according to their needs. These benefits come at a minimal latency overhead for connections (Section 6.2) and with low operational costs (Section 6.3.2). Furthermore, clients can choose which cloud service to trust with their domain history, thus providing flexibility with respect to privacy. In Section 7, we describe how clients can obtain even stronger privacy guarantees by hiding their domain history from the cloud service.

3.2 System Model

3.2.1 Entities in PEC. We introduce the entities that are needed for the operation of PEC.

- Trusted Third Party (TTP): Each PKIE is associated with a TTP that manages authentic information for a certain attribute of TLS certificates. Our goal is to relay this information securely to clients, so that they can decide whether to establish (or not) a new TLS session. Clients have the public key of the TTP in order to verify the authenticity of the relayed information.
- PKIE Prover: Generates proofs about attributes of servers' certificates based on the content provided by TTPs. Specifically, it proves the presence or absence of an attribute of a certificate. Provers are software components hosted by the PEC service.
- PEC Cloud Service: The service runs multiple provers, since each PKIE has its own prover. In addition, it is responsible to relay the information from the provers to the clients. The client can request service from multiple provers hosted by the service. The PEC service is running on the infrastructure of a cloud provider; for simplicity we use the term cloud service.
- Verifier: Verifies the information generated by the prover, ensuring that authentic information from the TTP has been relayed intact; we emphasize that verifiers trust TTPs and not provers. The verifier is a software component running at the client, and we have integrated it into the client's TLS stack. A client may contain multiple verifiers, one for every PKIE.
- Client/Server: A client can request service from multiple provers hosted by the cloud service, and contains the corresponding verifiers, one for every PKIE. A server optionally supports PEC; a supporting server sends the server-related information (including the certificate) to the cloud service.

3.2.2 Requirements for PKIEs. In order to be compatible with PEC, supported PKIEs must enable provers to produce complete and fresh information about the queried attribute, i.e., a *positive* or *negative* proof, indicating whether at the given time the attribute is satisfied or not. The notion of freshness for the provided information is defined by the TTP; freshness is necessary to mitigate replay attacks (Section 7).

Furthermore, the produced proofs must be efficient in terms of generation-verification time and size. Generation and verification times must be sufficiently low to not increase the latency of the connection setup. The size of the proof must be sufficiently low to not impose a significant transmission delay and bandwidth overhead.

One can imagine PKIEs that would not be compatible with our framework, but all the common PKIEs we know of are easily deployable on top of it. As a use case, we have implemented two prominent PKIEs (Section 5), and we provide examples of other PKIEs that are compatible with PEC in Appendix C.

3.2.3 Communication Flow. In PEC, we extend the TLS handshake to support cloud-hosted PKIEs. The handshake is extended from a bilateral procedure between the client and the server, to a trilateral one that involves also the cloud service: provers running at the cloud service, feed the client (verifier) with proofs about attributes of the server's certificate. The client proceeds with exchanging application data once it successfully verifies the information received from the cloud service.

Figure 1 describes the system model with the extended TLS handshake. Provers load and periodically update information from the corresponding TTPs. The client initiates the TLS connection to the server (`ClientHello` message), as in standard TLS (Step 1). In its `ClientHello` message, the client indicates which PKIE services it requires for the handshake. The server proceeds with the TLS handshake by sending its information (`ServerHello`, `Certificate`, `ServerHelloDone`) to the client as usual (Step 2a); for brevity, we will refer to all three messages from the server as the `ServerHello`. In addition, the server sends the `ServerHello` to the cloud service together with the PEC parameters indicated by the client (Step 2b). The prover (of one or more PKIEs depending on the client's parameters) generates the corresponding proofs (Step 3). Then, the cloud service sends the proofs to the client (Step 4) for the handshake to complete. The verifier, implemented in the TLS stack of the client, verifies the proofs (Step 5) and based on the result it decides whether it is safe to proceed and exchange application data with the server (Step 6).

3.2.4 Threat Model. The goal of the adversary is to convince clients to accept a rogue certificate (e.g., a revoked certificate or a certificate absent from the CT log), or to deny a valid certificate. The adversary can compromise the provers, the cloud service, and the communication between any of the involved entities; the adversary can delay, drop, modify, and inject traffic between all communicating entities. We assume that TLS and the cryptographic primitives we use are secure. Clients and servers are trusted to follow the operations of our protocols.

4 PROTOCOL DETAILS

We provide more details about the modifications to the TLS protocol (Section 4.1) and the operations performed by the cloud service (Section 4.2).

4.1 Extending TLS

In order to implement our protocol in a backwards-compatible way, we introduce the PEC TLS extension. The TLS stack of clients and servers can advertise additional functionality they support through *TLS extensions* [32]: the client advertises supported extensions in the `ClientHello` message, by indicating the extension type and other extension-related information. The server responds with the same extension type and its own information in the `ServerHello` message. If the server does not recognize the extension, it drops it from the `ServerHello`, informing the client it does not support the functionality. We start with the case where both the client and the server recognize the extension; then, we describe the client-only deployment case.

Client-Server Deployment. The client initiates the handshake with the PEC TLS extension that contains the required information about itself and about the requested PKIEs. The information is the following:

(1) Client Identifier: An identifier used by the cloud service to support multiple clients behind Network Address Translation (NAT) devices that share a single public address (Section 4.2.2).

(2) Requested PKIEs: A list of PKIEs that the client requests for the TLS handshake. To guarantee support for multiple PKIEs, the client provides a list of PKIE identifiers that are used to distinguish among PKIEs; such identifiers can be assigned by the IETF. The list of PKIEs is encoded in the Type-Length-Value (TLV) format with the type field being the PKIE identifier; the value field includes additional data that the client passes to the PKIE, and the length field indicates the size of the value field.

(3) Location of the cloud service: Address of the cloud's point-of-presence (PoP) to which the server should send its `ServerHello` message. This enables clients to choose a location that is close to them.

This information is preceded by a type code and a length field for the PEC extension, since TLS extensions are also encoded in the TLV format.

Figure 2a describes the exchange of TLS handshake messages between the client, the server, and the cloud service. The client sends out the `ClientHello` with the PEC extension (Step 1) and waits for the reply from the server. The server receives the `ClientHello` from the client and replies with the `ServerHello` (Step 2a). In addition, based on the client's information in the PEC extension, the server forwards the `ServerHello` to the cloud's PoP (Step 2b). Once the client receives the `ServerHello` (including the `Certificate`, `ServerKeyExchange`, and `ServerHelloDone` messages), it proceeds as usual by sending out the `ClientKeyExchange` and `ChangeCipherSpec (CCS)` messages (Step 3); then, it waits for the server's `CCS` message. During the handshake's message exchange, the client waits for the proof from the cloud service. If the proof has not arrived, the client halts the connection after receiving the server's `CCS` message (Step 4). Once the proof arrives (Step 5), the client validates it and decides if it is safe to proceed and typically starts exchanging application data with the server (Step 6).

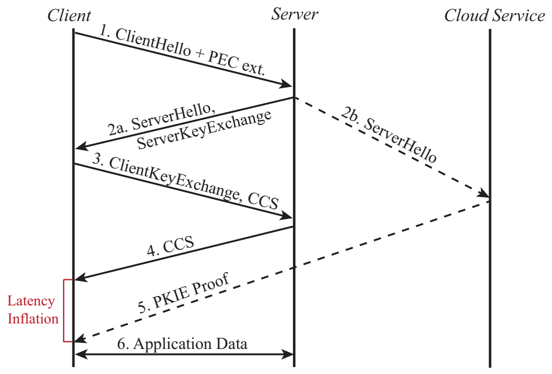
Halting the connection until the proof from the cloud arrives, guarantees security at the cost of a potential latency inflation. If the client would proceed to exchanging application data while waiting for the proof (and deciding later if the connection is safe), it would introduce an attack window that an adversary could exploit by delaying the proof. Therefore, the connection halts until the verification arrives. Note that it is possible for the proof to arrive (Step 5) chronologically before Step 4 or even Step 2a; the figure shows the worst case in which there is an additional latency for the connection setup.

Our goal is to minimize the additional latency, since the use of TLS already has a perceptible latency impact on clients [33]. Therefore, our modifications to the TLS state machine halt the connection (if necessary) at the latest possible point prior to data exchange. The server sends its information to the cloud service after receiving the `ClientHello`. The client verifies the server's certificate, but it is unlikely that the proof has arrived in the meantime due to the additional network and processing latency overheads. Thus, the client proceeds with the connection (Step 3) and halts before exchanging data. Our evaluation verifies that this approach introduces minimal latency for the TLS handshake (Sections 6.2 and 6.3.1).

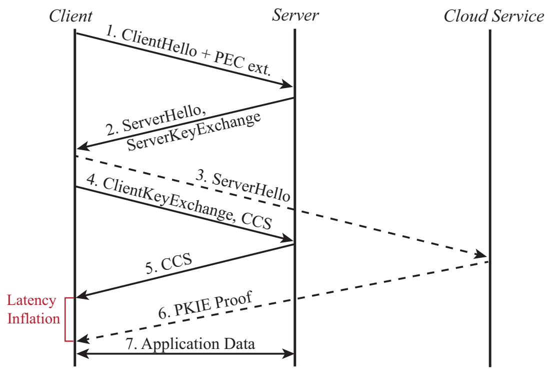
Client-only Deployment. We have assumed that servers recognize the PEC extension and send their information to the cloud service. However, at an early deployment stage, servers will probably not be aware of PEC.

Figure 2b describes the connection setup for the client-only deployment case. The client initiates the connection (Step 1) and the server responds with its information as usual (Step 2). Once the client receives the server's message, it sends the `ServerHello` to the cloud service for processing (Step 3) and waits for a response, similar to the client-server deployment scenario. The connection halts after the server's `CCS` message arrives (Step 5), if the proof has not arrived.

This approach introduces a potential latency inflation that is higher than in the client-server deployment case. Specifically, the server's information is sent to the cloud service once it is received by the client, i.e., half a client-to-server round-trip-time (RTT) later, compared to when the server is PEC compliant. The latency inflation in this case depends on the client-server and client-cloud latency difference.



(a) Client-Server Deployment.



(b) Client-only Deployment.

Figure 2: The steps of the TLS handshake in PEC.

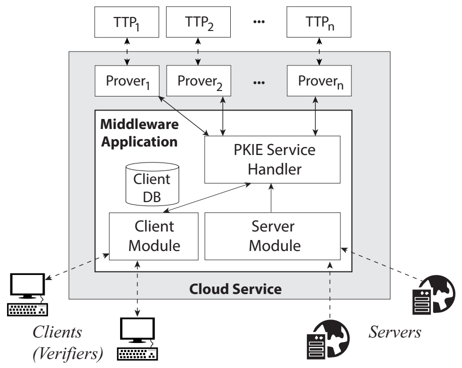


Figure 3: Architecture of the cloud service.

4.2 Cloud Operations

Our handshake deviates from the traditional bilateral handshake in that both the client and the server communicate with the cloud. This requires functionalities at the cloud service and protocols between the client and the cloud, and between the server and the cloud.

4.2.1 Cloud-Service Architecture. We describe the required cloud-service components to support multiple PKIEs. In order to support multiple provers as a single service, there is common functionality shared among the provers. This functionality is implemented by a *middleware application*.

Figure 3 shows the architecture of the cloud service. Specifically, the middleware maintains client information in a database and handles all communication with clients through the client module (Section 4.2.2). The server module receives the `ServerHello` messages from the servers and passes them to the PKIE Service Handler; for a client-only deployment, the server's information is received by the client module.

The handler reconstructs incoming packets into flows and parses the TLS messages they carry. Then, it calls the required provers based on the list of PKIE services indicated by the client. It collects the generated proofs and calls the client module to send the proofs to the clients.

Multiple provers can operate on top of the middleware application. Each prover periodically fetches information from its corresponding TTP. We highlight that operations in the cloud service are not trusted; only the information produced by TTPs.

4.2.2 Client-Cloud Protocol. The TLS stack of the client is listening for incoming proofs from the cloud service. Once the PKIE provers have processed the server's information and generated the corresponding proof(s), they are sent to the clients through the client module in the middleware.

The communication between the client and the cloud is implemented using UDP with a lightweight reliability protocol to handle packet loss. This reliability is necessary since packet loss would lead to a deadlock as the client is waiting for proof from the cloud before sending data. We discuss the tradeoff between performance and reliability in Section 8.1.

There is one complication in the client-cloud protocol: for clients behinds NATs, their address information is opaque to the cloud. Our protocol implements a client-identification mechanism with hole punching to traverse the NAT and to demultiplex multiple clients behind one public IP address.

Before opening a TLS session, the client initiates a client-registration process with the cloud service: the client sends a client registration request to the cloud service. The client module responds with a randomly generated 4-byte identifier that the client will use in the PEC extension of outgoing connections. Together with the client identifier, the client module also registers the source port observed in the registration request; it will be used as the destination port in future messages from the cloud to the client. Henceforth, the client will send periodic keepalive messages to the cloud (using the same source port) to maintain the NAT state up to date.

For the client-cloud communication, we considered the alternative approach of conducting the whole TLS handshake over the cloud: the server sends its information *only* to the cloud—not to the client—and the cloud forwards it to the client along with the corresponding proofs piggybacked: the message from the cloud would look as if it was sent from the server. We abandoned this approach due to practical limitations: 1) The cloud would have to spoof the server's address, which may lead to packet dropping, if ingress filtering is deployed [34]. 2) It may increase latency, since the `ServerHello` is redirected to the client through the cloud.[3]

[3]This statement is not entirely correct, since latency may also decrease if the cloud provider has a well-provisioned internal network with PoPs close to the client and the server.

4.2.3 Server-Cloud Protocol. The communication procedure between the server and the cloud is straightforward: the server encapsulates its information (Step 2b in Figure 2a) in a separate packet and sends it to the cloud. The IP destination address and the destination port are the ones specified by the client in the `ClientHello` extension. For the communication, a lightweight reliable communication based on UDP is used to counter packet loss, as for the client-cloud communication (Section 8.1).

5 CASE STUDIES

Existing PKIEs can be directly deployed on PEC, or can be easily modified to be compatible with PEC. As use cases, we consider two PKIEs: an improved version of Certificate Transparency (Section 5.1) and RITM (Section 5.2). We highlight that PEC is a generic deployment solution that is not restricted to the two use cases, but it can support multiple other PKIEs (Section 2).

5.1 Certificate Transparency

Certificate Transparency (CT) [6] is the most prominent log-based PKIE. Its main goal is to make the PKIE ecosystem more transparent by logging all certificates issued by CAs. The CT log is implemented with a hash tree [35] as its main data structure. This log construction is efficient in proving that a given leaf (certificate), is part of the tree (presence proof) with a processing complexity proportional to the logarithm of the number of logged certificates.

We improve CT in two ways: 1) We introduce absence proofs [8, 27], i.e., proving that a certificate is not part of the CT tree. 2) We reduce CT's storage overhead and fit the CT tree in memory, enabling efficient generation of proofs. PEC can support CT as it is used today, but our improvements are standalone contributions that improve CT's performance overall (whether it is used as a cloud-based PKIE or not).

We augment CT with a hash tree in which leaf nodes are lexicographically sorted hashes of issued certificates. We call this tree a *sorted tree* and it is reconstructed at every log's update (if new certificates were added). In every update, the root of the sorted tree is timestamped and signed, and is also appended as the last element of the original CT tree; we call this resulting tree the *extended tree*. For every update, provers contact the log to obtain the current root of the sorted tree and to obtain hashes of recently issued certificates; this information suffices to build the latest version of the tree.

The described modification provides two benefits: 1) It ensures that our improvement is compatible with other CT-related systems [36, 37] since the extended tree is consistent with the content of the original CT tree. 2) It enables the decoupling of the sorted tree from the extended tree in order to take advantage of its reduced size; this is possible as the root of the sorted tree is timestamped and signed.

Figure 4 shows the extended CT tree. The tree on top appends certificates ($c_1, \ldots, c_7$) in chronological order of issuance and has as its last leaf the root of the sorted tree (r_1). The sorted tree at the bottom has only hashes of certificates as leaf nodes, in lexicographical order. Note that the log maintains the extended tree, while provers maintain only the sorted tree.

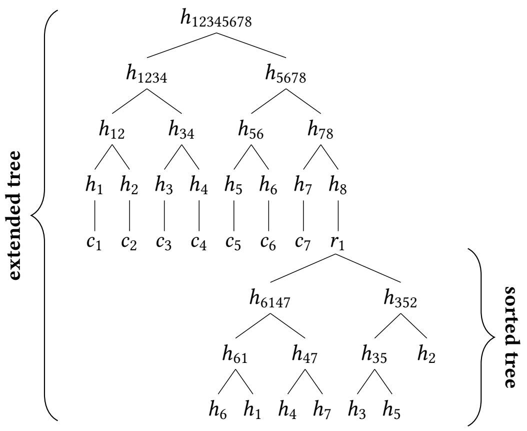


Figure 4: Example of the extended CT data structure. The log maintains the extended tree while provers maintain only the sorted tree.

The proofs produced from the sorted tree are sufficient to conclude that a certificate is (not) logged. For the example in Figure 4, to prove that the certificate c_6 is logged, a prover shows to a verifier the nodes h_1, h_{47}, h_{352} and the signed r_1 with its timestamp. During the TLS handshake in PEC, the verifier (client) receives these nodes and the certificate c_6 with its signed certificate timestamp (SCT). First, it verifies the certificate and the SCT; then, from the proof it reconstructs the root of the sorted tree (r_1) and verifies its signature and freshness. The new session is established if: (a) the presence proof is provided, or (b) the absence proof is provided but the SCT is not older than the maximum merge delay (MMD).

Our extension provides a stronger security property than pure CT, as the client can provably examine whether a certificate is (not) logged. Furthermore, the log cannot misbehave by issuing an SCT and not adding the certificate to the tree.

5.2 Revocation

Revocation is a fundamental process for PKI, as it is needed when non-expired certificates have to be invalidated. Revocation systems [10–15] are complementary to CT, since CT is an append-only log of issued certificates; there is no information about the revocation status of the certificate.

We have implemented RITM [15] that uses a trusted revocation log. The revocation log is implemented as a hash tree where leaves are the lexicographically sorted serial numbers[4] of revoked certificates. Thus, to prove that a certificate is not revoked, the prover produces an absence proof for the certificate's serial number. A client accepts a certificate if it is successfully verified and there is an absence proof for the certificate's serial number.

6 EVALUATION

We have implemented all required functionality for a working prototype. Specifically, we have augmented the OpenSSL library to accommodate the PEC TLS extension, the communication modules with the cloud, and the TTPs, provers, and verifiers for CT and

[4]Serial number is a unique integer assigned by the CA to each issued certificate.

	Presence Proofs			Absence Proofs		
	Max.	Min.	Avg.	Max.	Min.	Avg.
Gener. (μs)	135	51	60	200	81	92
Verif. (μs)	396	181	189	465	227	237
Size (bytes)	1018	1018	1018	1466	1070	1146

Table 1: Processing times and proof sizes for the improved version of CT.

RITM. In addition, we have implemented the required cloud service, i.e., the client/server modules, the handler, and the provers for CT and revocation.

6.1 Microbenchmarks

We conduct microbenchmarks for the operations of the PKIEs that we have implemented. Our implementation uses SHA-256 [38] as the default hash function (with 32-byte long hashes), Ed25519 [39] as the digital signature scheme (with 64-byte long signatures), and 4-byte long Unix timestamps. All operations are executed on the Intel(R) Core(TM) i7-4790 CPU.

6.1.1 Certificate Transparency. To test the integration of CT with PEC, we use the following experimental setup. For the CT log, we use Symantec's log[5] that contains 2,129,920 certificates with a storage footprint of 12.5 GB (including metadata). Based on this log, we construct a sorted tree requiring only 68 MB of storage—more than a 180-fold decrease in storage requirements. To keep the sorted tree in memory, about 293 MB of RAM is needed.

We investigate the processing time required for proof generation by the prover and proof verification by the verifier; these operations are performed for every TLS connection. The obtained times are presented for both presence and absence proofs, with presence proofs being the common case for CT. We also present the length of presence and absence proofs. Each result is obtained after running 1000 random samples. Table 1 shows the results.

As the results show, provers need on average 60 and 92 μs for generation of a presence and absence proof, respectively. The average verification time is 189 and 237 μs correspondingly. The sum of the generation and verification times is the entire processing overhead for the TLS connection setup using CT, and the numbers indicate that it is negligible compared to the network latency even for hosts close to each other. Moreover, the proof size is relatively short (about 1 KB on average), which indicates that the transmission latency will be low as well. We emphasize that by using the hash tree, the proof generation and verification times, as well as the proof size, increase logarithmically with the number of leaf nodes in the tree.

6.1.2 Revocation. We perform the same benchmarks for the revocation system described in Section 5.2. For revocation data, we consider Comodo's largest revocation list that contains 53,164 revoked serial numbers and requires 1.9 MB of storage.[6] The storage overhead for the provers is the same and the corresponding hash tree with the revocations requires about 8.2 MB of RAM.

[5]ct.ws.symantec.com
[6]http://bit.ly/294UNVr

	Presence Proofs			Absence Proofs		
	Max.	Min.	Avg.	Max.	Min.	Avg.
Gener. (μs)	83	38	39	90	61	65
Verif. (μs)	277	165	167	233	202	207
Size (bytes)	800	798	799	1282	848	919

Table 2: Processing times and proof sizes for the implemented revocation system.

Table 2 shows the performance results for our implementation (each operation was executed 1000 times). Provers need 39 and 65 μs on average to generate a presence and an absence proof respectively, with absence proofs being the common case. To verify a presence and an absence proof, verifiers need on average 167 and 207 μs respectively. The length of a presence proof is on average 799 bytes and of an absence proof 919 bytes.

Similar to CT, the results indicate that the processing and bandwidth overhead of the revocation PKIE is not a notable concern for the performance of a connection setup.

6.2 Amazon Deployment

We analyze the impact of PEC on the connection setup latency, using Amazon EC2. We create a virtual machine (VM) at each of the 14 different data centers (spread across four continents). On each VM, we install our modified TLS stack and the cloud service so that each VM can function as a client/server and as a cloud service.

First, we evaluate the impact of PEC on the latency of the TLS handshake. We select 3 out of the 14 VMs to operate as the client, the server, and the cloud service, using two different selection strategies: 1) We choose all 3 VMs randomly; we refer to this as the random selection model. We have experimented with 700 different combinations for the random model. 2) We first randomly choose two VMs that serve as the client and the server, and then choose the cloud VM based on the lowest latency from the client; we call this the smart selection model. This model provides a more realistic scenario, since clients are most likely to choose a PoP nearby. We experimented with more than 150 different combinations for the smart selection model.

For each selection model and configuration we take three measurements: 1) Latency of the TLS handshake without our framework; this serves as the baseline measurement. 2) Latency when the client and server deploy PEC, which we call the client-server deployment model. 3) Latency when only the client deploys PEC, which we call the client-only deployment model. To reduce noise and improve accuracy, each measurement is performed 20 times and the average is used as the latency value.

To put the latency data into perspective, we compute the relative latency inflation: we normalize the additional latency due to PEC based on the baseline latency without PEC. Figure 5 shows the relative inflation as a cumulative distribution function (CDFs) for the four cases: the combination of the two selection models (random, smart) with the two deployment cases (client-server, client-only). The shape of the markers ('o' or 'x') is used to distinguish the selection models and the style of the line (solid or dashed) is used to distinguish the deployment cases.

We make the following four observations based on Figure 5. First, the latency inflation is higher for the client-only deployment case

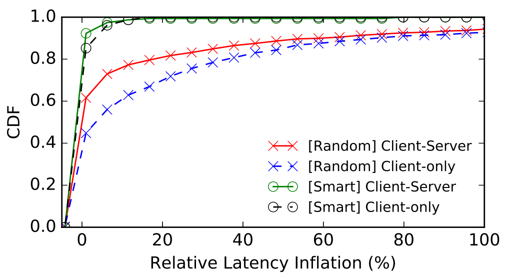


Figure 5: Latency Inflation of the TLS Handshake for the Amazon Deployment.

(comparison of solid lines to dashed lines). This is expected since the `ServerHello` is sent to the cloud at a later stage, once the client has received it from the server (see Figure 2a and Figure 2b).

Second, the latency inflation is significantly higher for the random selection model compared to that of the smart model. This is due to the higher probability of choosing a combination of 3 VMs where the client-cloud latency or the server-cloud latency is much higher than the client-server latency. For instance, the latency inflation is more than 130% when the client and server are located in East Asia (e.g., Korea and Japan), but the cloud VM is located in the U.S. East Coast (e.g., Ohio).

Third, for some experiments there is a latency deflation up to 5% (note that the x-axis does not start at 0). Such a deflation is not possible for our protocol, since the TLS connection setup does not complete until the client receives the `CCS` message from the server, even if the client receives the proof from the cloud in the meantime. We observe the small deflation rate because of measurement noise due to changes in network conditions.

Finally, for the smart model the latency inflation is less than 1% for 85% of the measurements in the client-only deployment and for 91% of the client-server deployment case. This result suggests that the latency overhead due to PEC is negligible for the majority of TLS connections.

6.3 Large-Scale Simulation

We try to simulate a large-scale deployment scenario by measuring the connection setup latency and by estimating the operational costs to the cloud provider.

6.3.1 Latency. In Section 6.2, we have shown that the latency inflation due to PEC is less than 2% in over 90% of cases, as long as the cloud is located close to the client. In this section, we evaluate latency inflation for a wider geographical distribution of clients and servers. Additionally, we analyze how the footprint of the cloud provider, i.e., the number and distribution of its PoPs, influences the latency inflation.

For this experiment, we use RIPE Atlas[7] for a more realistic distribution of clients and servers. Atlas provides a testbed with more than 9,000 nodes (probes and anchors) across 180 countries. We randomly choose 400 probes to simulate clients, and we choose all 245 anchors to simulate servers. We use anchors to simulate the servers since they are typically installed in better provisioned networks that are more amenable to incoming measurement traffic (e.g., ICMP ping requests).

We also evaluate the impact of the cloud provider's footprint. We use Amazon (based on the 14 VMs we have allocated) and Akamai's CDN network as cloud providers with a small and a large footprint respectively; for Akamai, we assume that computing facilities are collocated with the CDN edge servers.

For this experiment, we face the following two constraints. First, we cannot run our modified OpenSSL library on Atlas nodes, since they allow only few types of measurements (e.g., ping, traceroute, DNS queries). To overcome this limitation and compute the latency inflation, we base our analysis on latency measurements. This approach ignores the processing overhead of PEC, but this overhead is negligible compared to network latencies (Section 6.1). Thus, we use ping measurements to estimate the latency between two entities, assuming that the one-way latency is half the RTT. Then, we project the latency measurements to the number of RTTs required to complete the connection setup.

Second, to simulate a deployment based on Akamai's CDN, we need to determine the edge servers that are closest to the clients. To this end, we leverage Akamai's DNS-based solution: we generate DNS queries from the RIPE probes that simulate the location of the clients. The DNS responses contain the addresses of the Akamai servers that are closest to the clients and we use them as the cloud's PoPs.

Figure 6 shows the relative latency inflation for the four cases: combination of two deployment models (client-server, client-only) for the two different cloud providers (Akamai, Amazon). The shape of the markers ('o' or 'x') is used to distinguish the cloud providers and the style of the line (solid or dashed) is used to distinguish the deployment cases. Only the upper 20% of the CDF is shown in the Figure.

We make four observations: 1) Overall, latency inflation due to PEC is less than 0.5% for 90% of the cases. This result is similar to the result for the Amazon deployment with smart selection (Section 6.2). 2) Latency inflation is lower when the cloud has a larger footprint (comparison of lines with 'x' markers against 'o' markers). 3) Latency inflation is lower when both client and server deploy, similar to what we have seen in Section 6.2 (comparison of solid lines to dashed lines). 4) The client-server deployment model on a smaller cloud footprint has a lower latency inflation than the client-only deployment model on a larger cloud footprint. This suggests that the deployment model is more important than the cloud footprint in minimizing latency inflation.

6.3.2 Cost Estimation. We estimate the operational costs to the cloud provider for a large-scale deployment of PEC. Due to the difficulty of obtaining accurate statistics about traffic patterns, we use conservative estimates in order to compute upper bounds. We assume an Amazon cloud deployment and we use Amazon standard pricing, but for large deployments much lower prices would be negotiated. To compute the monthly cost, we need the inbound and outbound traffic volume and the number of computing resources

[7] https://atlas.ripe.net/

	Data In (TB/day)	**Data Out** (TB/day)	**Data Cost**	**Computing Cost**	**Total Cost**
Iceland	0.163	0.071	$194	$15	$209
Spain	18.57	8.15	$17,760	$390	$18,150
U.S.	140.7	61.77	$104,888	$2970	$107,858
Global	1740.66	764.18	$1,211,236	$36,615	$1,247,851

Table 3: Estimated monthly operational costs (traffic volume, computing resources, and their sum) to the cloud provider.

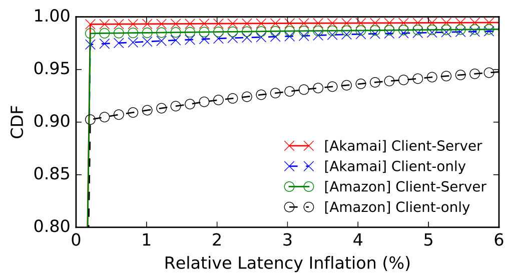


Figure 6: Latency Inflation of the TLS Handshake for the Large-Scale Simulation.

to support the TLS connection rate.[8] We assume that all TLS connections obtain service from both our implemented PKIEs (CT and RITM).

TLS Traffic Patterns. We analyze a private 24-hour trace of actual HTTPS traffic from SWITCH, a major educational ISP in Switzerland.[9] The data contains more than 1.2 million unique IPs and more than 74 million HTTPS entries. We compute the HTTPS connection setup rate per client and consider the maximum rate we observe. Then, we project the computed traffic rate to larger client bases in order to get intuition about large-scale deployments.

For the inbound traffic, the cloud receives `ServerHello` messages encapsulated in UDP packets. To estimate the traffic volume, we conduct TLS handshakes with Alexa's top 1000 domains and we compute an average `ServerHello` message length of 4553 bytes. For the outbound traffic, the length of each message is constant as we use the sum of the maximum size of CT and revocation proofs from our microbenchmarks (Section 6.1).

Computing Resources. In order to estimate the processing costs, we compute the number of Amazon server instances required to process a given load. We use Amazon's t2.medium instances,[10] which are capable of hosting the data structures needed for CT and RITM in their RAM. We conduct a throughput experiment and measure that one machine can serve 1900 connections per second for the common case, i.e., certificates that exist in the CT log and are not revoked.

Case Studies. We compute the operational costs for three different countries, ranging from small to large populations of active Internet users (Iceland, Spain, and the U.S.). We get the active Internet users for a country through the country's population and the Internet penetration percentage [40]. Moreover, we compute global operational costs based on the global number of Internet users (approx. 3.55 billion users) [41]. Note that by projecting the connection rate that we observed to a larger population, we obtain an upper bound for the aggregate traffic rate: the estimated rate per user is based on unique IP addresses, not based on individual users. Since we consider multiple users behind a NAT as a single user, the actual rate per user is lower than the rate per IP address.

Table 3 shows the estimated operational costs per month. Specifically, it shows the daily inbound and outbound data, the monthly traffic charges based on the outbound data, and the monthly computing costs required for PKIE processing. Our estimations provide some insight about the order of magnitude of the costs: the cost for a small country can easily be covered even by the smallest institutions, whereas the cost for larger countries and even the global traffic is well within reach of big corporations interested in securing the PKI (e.g., coalition of content providers or CAs).

7 SECURITY CONSIDERATIONS

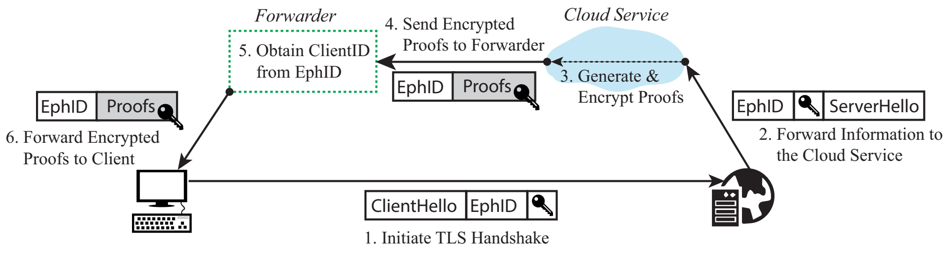
Privacy Enhancements. Our framework provides flexibility to users with respect to privacy: users can choose which cloud service to use and thus to which entity they disclose contacted domains. For example, an enterprise may deploy its own PKIE cloud service so that it does not disclose contacted domains to an untrusted cloud service (e.g., one operated by Google). This model provides higher flexibility compared to CRL or OCSP, in which contacted domains are revealed to the certificate's CA.

In addition, we design a privacy extension that provides even stronger privacy guarantees by preventing a third entity from linking contacted domains to the corresponding users. We start with the following observations: i) Clients' privacy suffers because their IP addresses and the domain information is disclosed to the same entity, i.e., the cloud service. ii) The client's IP and the domain's information are used for two orthogonal tasks: the latter is used to generate proofs for the domain's certificate and the former is used to send the proofs to the client.

Our privacy extension decouples the proof generation from sending the proofs so that the client's address and the domain is not disclosed to a single entity at the same time. To this end, we define a new entity—the *forwarder*—that sends the proofs to the clients. Furthermore, splitting the functionalities between the cloud service and the forwarder must satisfy the constraint that each entity must have access only to the information that is necessary to perform its functionality: client addresses must be hidden from the cloud service and generated proofs must be hidden from the forwarders.

We achieve this information unlinkability by making three changes to the PEC TLS extension (Figure 7). First, the address of a forwarder is added so that the cloud can forward generated proofs to the forwarder; our design enables the client to choose the forwarder. Second, the client generates a symmetric key per connection and sends it to the server in plaintext. The key is forwarded to the

[8]Inbound traffic in Amazon is not charged, but we provide it for completeness.
[9]https://www.switch.ch
[10]Two virtual CPUs and 4GB memory for $15 per month.



Forwarder
5. Obtain ClientID from EphID
4. Send Encrypted Proofs to Forwarder
Cloud Service
3. Generate & Encrypt Proofs
EphID
Proofs
EphID
Proofs
6. Forward Encrypted Proofs to Client
EphID
ServerHello
2. Forward Information to the Cloud Service
ClientHello
EphID
1. Initiate TLS Handshake

Figure 7: Communication flow with the privacy enhancement.

cloud service and it is then used to encrypt the proofs. Third, we substitute the client identifier with an *ephemeral identifier* (EphID) that is used by the forwarder to identify the client.

The ephemeral identifier is issued by the forwarder to its clients and can be linked to the client only by the forwarder. This prevents the cloud from linking the client to its contacted domains. Specifically, the forwarder issues multiple ephemeral identifiers, and clients use a different one for each TLS connection. Clients prefetch multiple ephemeral identifiers to avoid additional latency during connection establishment. Equation 1 shows how ephemeral client identifiers are generated: k is a local secret key known only to the forwarder, and $E_k()$ is a CCA-secure encryption scheme (e.g., AES-GCM). This construction enables stateless and efficient mappings between a client identifier and the corresponding ephemeral identifier.

$$EphID = E_k(ClientID) \qquad (1)$$

The cloud service is also modified to handle the privacy extension. It encrypts generated proofs using the symmetric key that is generated by the client and is included in the PEC TLS extension. Then, it sends the encrypted proofs with the ephemeral identifier to the forwarder. However, the symmetric key is not included in the message so that the proofs remain hidden from the forwarder.

Upon reception of encrypted proofs, the forwarder decrypts the ephemeral identifier (with key k) and obtains the client identifier. Then, the forwarder sends the encrypted proofs to the client together with the ephemeral identifier. The ephemeral identifier is used by the client to look up the one-time symmetric key that was used by the cloud service to encrypt the proofs.

We highlight two points about the privacy enhancement: i) It is not used by default, thus the additional overhead is paid only by users that select it. ii) It involves only efficient symmetric-key operations to keep the latency and performance overhead low.

Downgrade Attacks. To prevent a client from receiving PKIE services, an adversary may tamper with `ClientHello` and/or `ServerHello` messages by removing the PEC extension header. Regardless of which message has been affected, the client assumes that the server does not deploy PEC and then contacts directly the cloud (as in the client-only deployment scenario, Section 4.1). Additionally, the downgrade will be detected either by the client or the server (depending on which message was changed) after exchanging the TLS `Finished` messages that contain hashes over all previously exchanged messages.

Alternately, an adversary may block the communication channel between the client and the cloud in an attempt to force an insecure connection. In this case, the client does not proceed with the connection setup without proof from the cloud service and drops the connection.

Replay Attacks. An adversary may replay older proofs so that clients make their decision based on outdated information. For instance, an adversary may replay a revocation absence proof for a certificate that was only recently revoked, forcing the client to establish an insecure connection.

We emphasize that a replay attack is not an attack against the cloud service, but an attack against a PKIE: the prover nor the cloud service should be trusted to perform replay prevention; they could as well be the adversaries that perform replay attacks.

Thus, we argue that the TTP should prevent replay attacks by including signed freshness statements in the content that is provided to the prover. A freshness statement could include information such as a timestamp and an expiration time, which is passed to the verifier from the prover, and enables the verifier to judge the validity of the proof. For example, in CT the log signs with its private key the root of the tree together with a timestamp. Based on the trusted timestamp and the MMD, the verifier can decide whether the information is stale or not. This approach does not completely eliminate replay attacks, since there exists an attack window for replaying proofs, but the attack window is confined by the frequency of the freshness statements.

Reflection Attacks. A malicious client can use PEC to launch a reflection attack against a victim host. Specifically, the malicious client generates a `ClientHello` message with the address of its victim as the cloud address in the PKIE TLS extension and sends the `ClientHello` message to the server. Then, the server reflects the `ServerHello` message to the victim.

However, this reflection attack is harder to launch and less effective than typical reflection attacks (e.g., DNS or NTP based attacks). Unlike other reflection attacks, the client must make a TCP connection to the server, creating additional burden for the client compared to other attacks that typically leverage UDP-based services. Yet, it is less effective since the reflection attack using PEC has a smaller amplification factor. Specifically, the amplification factor is approximately 10 since an average `ClientHello` message with a PKIE TLS extension is about 400 bytes, and an average `ServerHello` message is 4553 bytes (Section 6.3.2). However, reflection attacks based on

DNS can have an amplification factor of 28 and attacks based on NTP up to 550.[11]

8 PRACTICAL CONSIDERATIONS

In this section, we discuss practical considerations of PEC from a technical perspective. We discuss other practical considerations, such as business models in the appendix.

8.1 System Reliability

Our framework introduces an additional entity and a modified communication flow that may affect the reliability establishing TLS sessions. Namely, reliability can be affected by the availability of the cloud service, packet loss for traffic to/from the cloud, and UDP traffic that is blocked. We describe how we deal with each issue.

Cloud-service Availability. We leverage cloud deployments for their high availability, yet an outage is a credible possibility [42].

In order to track the availability of the cloud service, we rely on the keepalive messages sent by clients to the cloud service. Keepalive messages have a dual role, as i) they maintain the NAT state up to date, and ii) they trigger a response from the cloud indicating that the service is running. If the cloud stops responding to the periodic keepalive messages, clients get promptly informed and switch to another cloud service, similar to falling back to a secondary DNS server.

In case of the privacy extension, users track the availability both of the cloud service and the forwarder. Note that users can track directly the availability of the cloud service, since they do not disclose any other information about contacted domains.

Packet Loss. We have chosen UDP as the transport protocol between the client-cloud and server-cloud communication. This design decision favors connection performance due to the reduced latency (no round-trip for connection setup). As a consequence, we must explicitly handle packet loss, since it would lead to a deadlock if traffic is dropped at any of the two communication segments.

To counter packet loss, we use a lightweight reliability protocol on top of UDP. We use per-packet sequence numbers and timers to detect loss and reordered packets, and we use the standard TCP mechanism to update the retransmission timeout [43]. Specifically, we leverage loss recovery as implemented in QUIC [44], but without the functionalities for flow control and congestion control since only a few number of packets is sent.

Blocked UDP traffic. The use of UDP raises connectivity concerns since users are often behind middleboxes that block UDP traffic; this would cause availability problems for the client-cloud communication. Studies performed before the design of QUIC demonstrate that over 90% of clients can successfully create outbound UDP connections [45]. In case clients cannot connect to the cloud over UDP, they fallback to TCP as is done in Chrome when QUIC cannot be used [46]. Furthermore, to avoid the latency overhead of TCP, the client starts a TCP connection with the cloud and sends periodic keepalive messages (packets with null data). Whenever the cloud service generates proofs for a connection of the client it sends them over the active TCP connection.

[11] https://www.us-cert.gov/ncas/alerts/TA14-017A

On top of our reliability mechanisms, we envision that browser vendors will build in additional fallback solutions when proofs are not delivered. Browser vendors are reluctant to implement solutions that sacrifice availability for security and therefore they usually provide warnings to the clients. A similar approach can apply also for our framework, so that clients will receive appropriate warning messages, if proofs from the cloud service are not delivered.

8.2 Deployment

Client-Side Middleboxes. Our solution favors a client-centric approach since it has proven easier to update clients than servers. However, there are cases in which it is difficult to update clients (e.g., hardware-constrained devices like in IoT settings). For such clients, PKIE middleboxes can be deployed in clients' networks to offer the PKIE functionality. Furthermore, ISPs can push the PKIE functionality as updates to home routers that are distributed by them to the customers, even eliminating the necessity of deploying new hardware for PKIE middleboxes.

A PKIE middlebox keeps state per TLS session and implements common middlebox operations: 1) Using deep-packet-inspection, it intercepts TLS handshakes and redirects the server's certificate to the cloud. 2) It halts the server's CCS message until the proof arrives from the cloud. 3) It forwards the server's CCS message if the verification is successful. The latency inflation of this approach is similar to the client-only deployment, since the middlebox is placed in the client's network.

However, the increase in latency may affect the TCP state machine of the server: if the proof is delayed, the server will start retransmitting its CCS message since no acknowledgment from the client has been received. Our evaluation (Section 6.3.1) shows how it is possible to minimize the latency inflation if the cloud has a large footprint with PoPs close to the Internet edge.

Browser Updates. Clients need to be updated to support PEC. Specifically, this can be achieved by updating the TLS libraries of browsers. However, we do not have to rely on browser vendors to implement verifiers. Instead, entities that offer PKIE provers can develop the corresponding verifiers and offer them as browser plugins. If a PKIE becomes popular, it can then be integrated into the core browser application.

9 OTHER RELATED WORK

APLOMB [31] is an enterprise-focused solution to outsource middlebox processing to the cloud. The authors show that enterprises can transfer most of their in-house middleboxes to highly available cloud services and benefit from reduced costs for infrastructure, personnel, and management. We apply this principle for a different purpose—to enable mass adoption of multiple PKIEs at a location where they can be accessed universally. Additionally, our evaluation confirms the cost-effectiveness of cloud deployments.

TLSDeputy [47] is a cloud-based system that supports residential clients (e.g., IoT devices) to perform TLS operations (e.g., certificate validation). The proposal builds on OpenFlow-enabled residential switches and cloud-based controllers and middleboxes. This approach introduces substantial latency for all flows due to the switch-controller communication (especially because the controller is placed in the cloud). Furthermore, outsourcing the actual decision for connection acceptance to the cloud implies a weak and impractical threat model: the controller, the middlebox, the switch, and the communication channels between them must be trusted.

10 CONCLUSIONS

We believe that sustainable evolution is the key to a healthy TLS PKI. To this end, we have proposed a framework that leverages elastic and highly available computing resources in the cloud to deploy PKI enhancements. We have shown that our solution introduces little overhead to TLS handshakes—less than 2% on average—and that it is cost-effective; the operational costs are well within reach for interested corporations. Given a viable deployment path for new enhancements, we would be intrigued to see how the PKI ecosystem will evolve in the future.

11 ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful feedback. The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013)/ERC grant agreement 617605; from the IITP grant funded by the Korean government (MSIT) (No.2016-0-00157, Development of self-certifying ID based trustworthy networking technology); and, from SUTD SRG ISTD 2017 128 grant. We also gratefully acknowledge support by ETH Zürich and the Zürich Information Security Center (ZISC).

REFERENCES

[1] "Let's Encrypt!" "https://letsencrypt.org/".

[2] A. Manousis, R. Ragsdale, B. Draffin, A. Agrawal, and V. Sekar, "Shedding Light on the Adoption of Let's Encrypt," *arXiv preprint arXiv:1611.00469*, 2016.

[3] M. Aertsen, M. Korczyński, G. Moura, S. Tajalizadehkhoob, and J. v. d. Berg, "No domain left behind: is Let's Encrypt democratizing encryption?" *arXiv preprint arXiv:1612.03005*, 2016.

[4] "AWS Certificate Manager Pricing," "http://amzn.to/2k7NyO0".

[5] "Introducing Universal SSL," "http://bit.ly/1rvItNz".

[6] B. Laurie, A. Langley, and E. Kasper, "Certificate Transparency," Tech. Rep., 2013.

[7] EFF, "Sovereign Keys: A Proposal to Make HTTPS and Email More Secure," "http://bit.ly/2jSH9Jd", 2011.

[8] T. H.-J. Kim, L.-S. Huang, A. Perrig, C. Jackson, and V. Gligor, "Accountable Key Infrastructure (AKI): A Proposal for a Public-key Validation Infrastructure," in *Proc. of the ACM International Conference on World Wide Web (WWW)*, 2013.

[9] D. Basin, C. Cremers, T. H.-J. Kim, A. Perrig, R. Sasse, and P. Szalachowski, "ARPKI: Attack Resilient Public-Key Infrastructure," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2014.

[10] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile," RFC 5280, IETF, 2008.

[11] A. Langley, "Revocation Checking and Chrome's CRL," "http://bit.ly/2k7DCE9", 2015.

[12] "Mozilla's Revocation Plan," "https://wiki.mozilla.org/CA:RevocationPlan".

[13] S. Santesson, M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams, "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP," RFC 6960, IETF, 2013.

[14] Y. Pettersen, "The Transport Layer Security (TLS) Multiple Certificate Status Request Extension," RFC 6961, IETF, 2013.

[15] P. Szalachowski, L. Chuat, T. Lee, and A. Perrig, "RITM: Revocation in the Middle," in *Proc. of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2016.

[16] P. Szalachowski, S. Matsumoto, and A. Perrig, "PoliCert: Secure and Flexible TLS Certificate Management," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2014.

[17] Y. Liu, W. Tome, L. Zhang, D. Choffnes, D. Levin, B. Maggs, A. Mislove, A. Schulman, and C. Wilson, "An End-to-End Measurement of Certificate Revocation in the Web's PKI," in *Proc. of the ACM Internet Measurements Conference (IMC)*, 2015.

[18] "Certificate Transparency Will Be Mandatory in Chrome," "http://ibm.co/2eGf9SC".

[19] H. Tschofenig and T. Gondrom, "Standardizing the Next Generation Public Key Infrastructure," in *Proc. of the Workshop on Improving Trust in the Online Marketplace*, 2013.

[20] T. Wolf, J. Griffioen, K. L. Calvert, R. Dutta, G. N. Rouskas, I. Baldine, and A. Nagurney, "Choice As a Principle in Network Architecture," in *Proc. of the ACM SIGCOMM Conference*, 2012.

[21] X. Yang, D. Clark, and A. W. Berger, "NIRA: A New Inter-domain Routing Architecture," *IEEE/ACM Trans. Netw.*, 2007.

[22] D. Han, A. Anand, F. Dogar, B. Li, H. Lim, M. Machado, A. Mukundan, W. Wu, A. Akella, D. G. Andersen, J. W. Byers, S. Seshan, and P. Steenkiste, "XIA: Efficient Support for Evolvable Internetworking," in *Proc. of the USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2012.

[23] R. R. Sambasivan, D. Tran-Lam, A. Akella, and P. Steenkiste, "Bootstrapping Evolvability for Inter-Domain Routing," in *Proc. of the ACM Workshop on Hot Topics in Networks (HotNets)*, 2015.

[24] S. Ratnasamy, S. Shenker, and S. McCanne, "Towards an Evolvable Internet Architecture," in *Proc. of the ACM SIGCOMM Conference*, 2005.

[25] D. Wendlandt, D. G. Andersen, and A. Perrig, "Perspectives: Improving ssh-style host authentication with multi-path probing." in *Proc. of the USENIX Annual Technical Conference (ATC)*, 2008.

[26] M. Marlinspike, "Convergence," *http://convergence.io*, 2011.

[27] M. D. Ryan, "Enhanced Certificate Transparency and End-to-End Encrypted Mail," in *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2014.

[28] V. Cheval, M. Ryan, and J. Yu, "DTKI: a new formalized PKI with no trusted parties," *arXiv preprint arXiv:1408.1023*, 2014.

[29] M. O'Neill, S. Heidbrink, S. Ruoti, J. Whitehead, D. Bunker, L. Dickinson, T. Hendershot, J. Reynolds, K. Seamons, and D. Zappala, "TrustBase: An Architecture to Repair and Strengthen Certificate-based Authentication," in *Proc. of the USENIX Security Symposium (USENIX Security)*, 2017.

[30] A. Bates, J. Pletcher, T. Nichols, B. Hollembaek, D. Tian, K. R. Butler, and A. Alkhelaifi, "Securing SSL Certificate Verification Through Dynamic Linking," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2014.

[31] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making Middleboxes Someone else's Problem: Network Proc.ssing As a Cloud Service," in *Proc. of the ACM SIGCOMM Conference*, 2012.

[32] D. E. 3rd, "Transport Layer Security (TLS) Extensions: Extension Definitions," RFC 6066, IETF, 2011.

[33] D. Naylor, A. Finamore, I. Leontiadis, Y. Grunenberger, M. Mellia, M. Munafò, K. Papagiannaki, and P. Steenkiste, "The Cost of the "S" in HTTPS," in *Proc. of the ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2014.

[34] P. Ferguson and D. Senie, "Network Ingress Filtering: Defeating Denial of Service Attacks which Employ IP Source Address Spoofing," RFC 2827, IETF, 2000.

[35] R. C. Merkle, "A Digital Signature Based on a Conventional Encryption Function," in *Proc.edings of Advances in Cryptology*, 1988.

[36] L. Nordberg, D. Gillmor, and T. Ritter, "Gossiping in CT," *Internet-Draft draft-linus-trans-gossip-ct-04*, 2017.

[37] M. Chase and S. Meiklejohn, "Transparency Overlays and Applications," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2016.

[38] NIST, "FIPS 180-4, Secure Hash Standard," http://bit.ly/1nIPyYX, 2012.

[39] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang, "High-speed High-security Signatures," *Journal of Cryptographic Engineering*, 2012.

[40] "Internet Users by Country," "http://bit.ly/1ywyEl8", 2016.

[41] "Internet Users," "http://bit.ly/RdZ6QH".

[42] AWS, "Summary of the Amazon S3 Service Disruption in the Northern Virginia," "https://aws.amazon.com/message/41926/", 2017.

[43] V. Paxson, M. Allman, J. Chu, and M. Sargent, "Computing TCP's Retransmission Timer," RFC 6298, IETF, Jun. 2011.

[44] Google, "QUIC Loss Detection and Congestion Control," "http://bit.ly/2pNN5Tw", 2011.

[45] J. Roskind, "Quick UDP Internet Connections," "http://bit.ly/2rjBgpb", 2013.

[46] R. Hamilton, J. Iyengar, I. Swett, and A. Wilk, "QUIC: A UDP-Based Secure and Reliable Transport for HTTP/2," "http://bit.ly/2qs0kMz", 2016.

[47] C. R. Taylor and C. A. Shue, "Validating security protocols with cloud-based middleboxes," in *Proc. of the IEEE Conference on Communications and Network Security (CNS)*, 2016.

[48] E. Rescorla, "The transport layer security (TLS) protocol version 1.3," 2016.

[49] J. G. Beekman, J. L. Manferdelli, and D. Wagner, "Attestation Transparency: Building secure Internet services for legacy clients," in *Proc. of the ACM Asia Conference on Computer and Communications Security (AsiaCCS)*, 2016.

[50] D.-Y. Yu, A. Ranganathan, R. J. Masti, C. Soriente, and S. Capkun, "SALVE: Server Authentication with Location Verification," in *Proc. of the ACM Conference on Mobile Computing and Networking (MobiCom)*, 2016.

[51] A. Abdou and P. van Oorschot, "Server Location Verification and Server Location Pinning: Augmenting TLS Authentication," *arXiv preprint arXiv:1608.03939*, 2016.

A TLS ISSUES

Session Resumption. A client who has previously negotiated a session with a server may perform an abbreviated handshake, called a session resumption—an expedited method to reconnect with the server. Session resumption introduces a security implication because the client implicitly accepts the server's certificate, which was received during the initial TLS handshake.

In this case, the security policy that is followed is up to the client: the client may trust the proof that was received during the initial handshake and proceed normally with session resumption. Alternatively, the client may contact the cloud service for a fresh proof before proceeding with session resumption. This approach may incur additional latency (RTT to the cloud).

TLS 1.3. TLS 1.3 [48]—the latest version of the protocol (still work in progress)— introduces differences to the TLS handshake in order to reduce connection latency by one RTT. Since the connection latency is decreased, PEC increases the relative latency inflation.

We perform the same latency evaluation as in Section 6.3.1. We also use the same RTT measurements between RIPE Atlas and our Amazon instances. Our simulation shows that PEC increases latency of TLS 1.3 by 6.5% on average. Differently put: PEC reduces the latency benefit of TLS 1.3 over TLS 1.2 by 3%, which indicates that PEC does not cancel the latency benefits of TLS 1.3 over TLS 1.2.

B BUSINESS MODELS

Although we have specified the components of the cloud service, we have not specified the entities that operate each component. We believe that different business models may evolve, based on today's practices. In what we consider the most realistic scenario, a large corporation that operates its own cloud infrastructure and acts as a TTP can operate the whole system: the cloud infrastructure to the cloud service to the TTP.

Alternately, a cloud provider can offer part of the PEC cloud service as a service. Specifically, the cloud provider offers the basic services that are needed (e.g., middleware application) so that third parties can operate their provers.

Moreover, a cloud provider can simply offer its computing and networking resources to interested parties (Infrastructure-as-a-Service model). A coalition of entities would come together to operate their provers and coordinate to run the middleware application.

C OTHER PKIES

We demonstrated the operation of PEC using two PKIEs, however, there are many PKIEs that follow the model described in Section 3.2.2.

PoliCert [16] is a log-based approach that enables domain owners to specify fine-grained security policies about the use of their certificates. Implementing a PoliCert prover on PEC can help domains to make their security policies transparent and confine the range of malicious certificates that an attack can forge. In a similar fashion, PEC combined with Attestation Transparency [49], can provide verifiable information to users about what services exist and what they do.

In addition, PEC can foster the deployment of novel proposals that leverage location verification as an additional authentication factor in TLS [50, 51]. PEC can serve as a centralized relay of authentic location information obtained from the corresponding TTP.