CAPS: Smoothly Transitioning to a More Resilient Web PKI

Stephanos Matsumoto
Olin College of Engineering
smatsumoto@olin.edu

Jay Bosamiya
Carnegie Mellon University
jaybosamiya@cmu.edu

Yucheng Dai
Carnegie Mellon University
yuchengd@andrew.cmu.edu

Paul van Oorschot
Carleton University
paulv@scs.carleton.ca

Bryan Parno
Carnegie Mellon University
parno@cmu.edu

ABSTRACT

Many recent proposals to increase the resilience of the Web PKI against misbehaving CAs face significant obstacles to deployment. These hurdles include (1) the requirement of drastic changes to the existing PKI players and their interactions, (2) the lack of signaling mechanisms to protect against downgrade attacks, (3) the lack of an incremental deployment strategy, and (4) the use of inflexible mechanisms that hinder recovery from misconfiguration or from the loss or compromise of private keys. As a result, few of these proposals have seen widespread deployment, despite their promise of a more secure Web PKI. To address these roadblocks, we propose Certificates with Automated Policies and Signaling (CAPS), a system that leverages the infrastructure of the existing Web PKI to overcome the aforementioned hurdles. CAPS offers a seamless and secure transition away from today's insecure Web PKI and towards present and future proposals to improve the Web PKI. Crucially, with CAPS, domains can take simple steps to protect themselves from MITM attacks in the presence of one or more misbehaving CAs, and yet the interaction between domains and CAs remains fundamentally the same. We implement CAPS and show that it adds at most 5% to connection establishment latency.

CCS CONCEPTS

• **Security and privacy** → **Web protocol security**.

KEYWORDS

public-key infrastructure, HTTPS, TLS, deployment

ACM Reference Format:
Stephanos Matsumoto, Jay Bosamiya, Yucheng Dai, Paul van Oorschot, and Bryan Parno. 2020. CAPS: Smoothly Transitioning to a More Resilient Web PKI . In *Annual Computer Security Applications Conference (ACSAC 2020), December 7–11, 2020, Austin, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3427228.3427284

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ACSAC '20, December 7–11, Austin, TX, USA
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8858-0/20/12…$15.00
https://doi.org/10.1145/3427228.3427284

1 INTRODUCTION

HTTPS is fundamental for secure Web communication. When a user Alice wishes to securely access Bob's site `bob.com`, HTTPS allows Bob to serve his site over a secure communication channel that provides secrecy and integrity. To establish this channel, Alice and Bob perform the Transport Layer Security (TLS) handshake protocol [33], which allows Bob to use his public key K_B to establish a shared secret key with Alice, which they can subsequently use for encrypted communication.

To convince Alice that K_B should be associated with `bob.com`, however, HTTPS relies on the Web public-key infrastructure (PKI). A trusted third party called a *certificate authority (CA)* checks that Bob owns both `bob.com` and the private key corresponding to K_B, and issues a digitally signed certificate that vouches for this binding. CAs thus play a crucial role in secure Web communication: the failure of any CA due to error, compromise, or coercion can lead to a certificate that binds `bob.com` to a different public key K_M. If for example this key belongs to an adversary Mallory, she can impersonate Bob to Alice in a *man-in-the-middle (MITM) attack*, one of the main problems that a PKI aims to solve.

Unfortunately, the current Web PKI is demonstrably fragile. Existing certificate databases [11, 26] indicate that Web browsers and operating systems provided by Mozilla, Apple, and Microsoft directly or indirectly trust more than 1,500 CA signing keys across more than 600 organizations worldwide. Little prevents any of these CAs from issuing an unauthorized certificate for any site, resulting in *weakest-link security* for most sites: the compromise of any CA can threaten the security of all Web domains, and by extension, all clients visiting sites on those domains. Recent years have seen a plethora of incidents where misbehaving CAs issued unauthorized certificates, both accidentally [37] and intentionally [42].

As we describe in Section 2, previous work has made progress towards protecting clients and domains against the misbehavior of trusted entities such as CAs, and some of this work has seen increasingly widespread deployment. Unfortunately, no proposed solution offers both preemptive, robust protection against misbehaving CAs and a feasible deployment strategy. In particular, while systems like Google's CT project [25, 26] enjoy relatively widespread deployment, they only enable detection, not prevention, of unauthorized certificate issuance. Systems that do prevent unauthorized issuance (1) require drastic changes to certificate issuance [22], (2) require domains to deploy complex new infrastructure [20, 41], (3) increase latency and communication [45], (4) require all domains to increase their security level at once [3], or (5) render a domain inaccessible if misconfigured [32].

To address these shortcomings, we propose Certificates with Automated Policies and Signaling (CAPS), a system that provides a practical roadmap for transitioning to a more resilient Web PKI. In CAPS, domains can take simple steps to protect themselves from MITM attacks in the presence of one or more misbehaving CAs. CAPS also provides an incremental deployment strategy. Interactions between domains and CAs remain unchanged in the vast majority of cases, and simple misconfigurations do not cause domains to become inaccessible. CAPS provides immediate benefits to domains and clients who deploy it, but does so in a way that does not penalize non-deploying domains. CAPS also protects clients from downgrade attacks such as *TLS stripping* [28]. Finally, the bulk of the deployment effort of CAPS occurs at a small handful of participants, namely browser vendors and public logs, who are better equipped to make these changes than domains or CAs that have historically been reluctant to deploy major changes.

The design of CAPS relies on four observations.

B1. *Our near-complete view of the modern Web PKI can serve as a low-effort channel for domains to communicate information.* Fast, Internet-wide scanning [12] and browser-based Web PKI standards [38] have produced large databases of TLS certificates [11]. This view provides domain information, including CA information and public keys, without domains taking any additional action beyond obtaining certificates (which they already do).

B2. *A viable deployment strategy must signal both deployment of HTTPS and deployment of PKI enhancements to clients.* The Web PKI encompasses a vast set of domains with diverse security needs, and it is highly unlikely that any change to the PKI will be universally adopted overnight. Therefore CAPS provides a *signaling mechanism* that conveys (1) whether a domain supports HTTPS, and (2) if so, whether it supports CAPS. Without the first bit of information, TLS stripping can occur, causing the client to ignore even the existing PKI. With this bit, the remaining signaling information can be communicated as an extension to the standard TLS handshake. Previous approaches have required operational changes to domains [2, 20] or significant storage overhead (Section 2.2). CAPS uses a global view of the Web PKI along with data compression techniques and compact data structures to locally store at each client a succinct summary of nearly the *entire* set of domains that deploy HTTPS. In our prototype, we summarize over 64 million domain names with <150 MB of storage (and Section 6.2 discusses tradeoffs that reduce this overhead to <35 MB).

B3. *Any domain that supports HTTPS has obtained a certificate using an existing issuance process, and can use this process to obtain certificates from further CAs.* Domains in CAPS use multiple independent certificate chains for the same public key to establish one or more "authoritative" public keys (Section 4.2). By communicating only the *number* of independent chains that denote an authoritative key, CAPS makes it easy for a domain to recover from the loss or compromise of a private key or from a misconfiguration. The domain simply obtains an equal or greater number of chains for a new public key to establish it as authoritative. Moreover, the CAPS' global view allows it to *automatically* keep this number up to date with no changes to current certificate issuance processes and no additional action from domains. Even with this flexibility, authoritative public keys in CAPS are hard to forge: if a public key is backed by n independent chains, an adversary must forge *at least* n certificates for the same public key to carry out a MITM attack.

B4. *Authoritative public keys benefit both current and future Web PKIs.* These keys can be used on their own to provide greater confidence in a site's identity, or they can be used to authenticate richer policies proposed in prior work [3, 41]. CAPS can thus simplify the deployment of these proposed systems, whose existing deployment and certificate issuance strategies rely on complex coordination among domains, CAs, and public logs to certify these policies. CAPS also enhances the recoverability of these systems, which, as originally proposed, require waiting for days to replace a policy if the corresponding private key is lost or compromised.

Even without considering wholly new PKIs, CAPS is attractive from a deployment standpoint. In particular, the administrative burden of deployment for domains is limited to acquiring additional certificates, and with the use of free certificate services like Let's Encrypt [14], the financial burden can be minimized as well. Furthermore, CAPS is an opt-in system, meaning that only domains who choose to obtain additional certificates incur a cost, and this cost is mainly for purchasing the certificates. From its initial deployment, CAPS protects all domains from TLS stripping, and it allows non-deploying domains to coexist with deploying domains without enabling downgrade attacks on deploying domains.

In summary, we make the following contributions.

- We present the design of CAPS, which provides an incrementally deployable, backwards compatible path to a more resilient Web PKI.
- We show how a security policy based on the number of independent certificate chains for a domain strikes a good balance between automation, resilience to attack, and robustness to domain errors (including private key loss).
- We combine a global view of Web certificates with techniques from data compression and compact data structures to succinctly signal HTTPS deployment.
- We demonstrate via an evaluation of a prototype that the client-side overhead for CAPS in terms of storage, memory, and connection-establishment latency is modest.

2 BACKGROUND AND RELATED WORK

We provide a brief overview of the work related to the three problems we address in this paper: tracking Web certificates, enforcing the use of HTTPS, and improving certificate authentication.

2.1 Tracking Web Certificates

Censys is a service that uses ZMap [12] to provide a search engine of network devices and infrastructure in the IPv4 address space [11]. In the context of the Web PKI, Censys keeps records of periodic TLS handshake attempts to the entirety of the public IPv4 address space on port 443 (the standard port for HTTPS) dating back to 2015. Censys provides a database of information on certificates received in these handshakes such as validity in different operating systems.

Certificate Transparency (CT) publicizes certificate issuance [26]. CT introduces the role of *certificate logs* to the Web PKI, entities that use Merkle hash trees [30] to maintain an auditable, append-only, and tamper-proof database [7] of certificates in the Web PKI. A domain obtains proofs from logs that its certificate has been

logged, and sends these proofs to clients during the TLS handshake inside a TLS extension. (Two extensions may be used: a CT-specific extension or the Certificate Status Request extension, also known as OCSP stapling [13].) CT-enabled clients reject certificates that are not accompanied by such a proof.

Censys and CT can be used in conjunction to provide a reasonably complete view of the Web's HTTPS ecosystem, particularly the fully-qualified domain names (FQDNs) of sites deploying HTTPS [43]. Previous work has used this view to efficiently represent the set of all revoked certificates [24].

2.2 Enforcing HTTPS

Recent work to enforce the use of HTTPS has aimed to address and prevent MITM attacks that use *TLS stripping*. In a TLS stripping attack, an adversary causes a client and server to establish a connection over plain HTTP, even when they could have established a connection over HTTPS. Since HTTP is unencrypted, this makes a MITM attack possible by *any* entity between the client and server. Below, we describe several efforts in this space.

HTTP Strict Transport Security (HSTS) is a mechanism that allows sites to tell a client connecting over HTTP that all future connections to the site should take place over HTTPS [19]. A site typically does this by redirecting to HTTPS and providing a *strict transport security policy* in its response specifying the time period and subdomains for which the client should use HTTPS. HSTS is *trust-on-first-use (TOFU)*, meaning that the adversary can only mount a MITM attack when a client connects to a site without having a strict transport security policy. Web browsers (mainly Chrome) mitigate this vulnerability with an *HSTS preload list* of domains that are always treated as if they have such a policy in place. This approach cannot scale as-is to all HTTPS sites and thus only protects a limited set of sites.

HTTPS Everywhere is a Web browser extension that rewrites HTTP requests to HTTPS requests for certain sites confirmed to serve over HTTPS [17]. Because additions to the list of confirmed HTTPS sites are typically suggested to the developers by members of the public, there is a delay between when a site newly deploys HTTPS and when HTTPS Everywhere enforces HTTPS connections for that site. Similarly to HSTS preload lists, this approach cannot scale to the entire Web and thus protects a limited set of sites (though larger than does HSTS preloading).

Smart HTTPS is a Web browser extension that simply rewrites *all* HTTP requests to HTTPS requests, falling back to HTTP if it encounters an error during the HTTPS connection attempt [21]. Thus, it adds significant latency when connecting to HTTP-only sites, as it must first fall back to HTTP. Smart HTTPS mitigates this weakness by using the first response to cache the domain as HTTP or HTTPS, but this makes a MITM attack trivial: an adversary can simply intercept and block the HTTPS request to cause Smart HTTPS to *permanently* log the site as HTTP-only.

2.3 Rethinking Certificate Authentication

Recent work on improving certificate issuance and validation has focused on providing alternate or additional mechanisms by which domains can authenticate their public keys to clients. We focus on two approaches: public-key pinning, and log-based authentication.

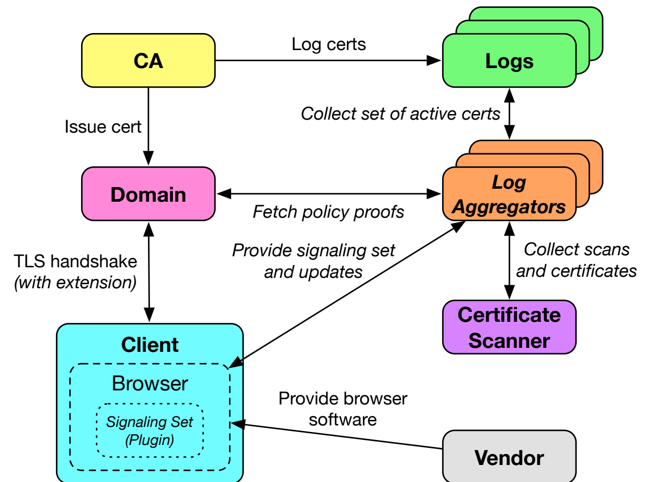

Figure 1: Overview of CAPS architecture (log auditors and monitors not shown). Dotted lines denote the browser and its components, and italic text denotes new entities or actions in CAPS. As shown in the diagram, CAPS is currently implemented as a browser plug-in, but we envision it would become a standard browser component.

Public-key pinning is a TOFU approach that, rather than only enforce the use of HTTPS, enforces the use of specific public keys for a site. HTTP Public Key Pinning (HPKP) allows a domain to specify hashes of public key information for one or more certificates in the domain's certificate chain [15], while Trust Assertions for Certificate Keys (TACK) allows a domain to specify a public key that the client should treat as a trust anchor for the domain [29]. For security, both approaches deliberately make it difficult to prematurely remove or update a pin for a domain, but this approach can easily make a domain inaccessible due to misconfiguration or malicious behavior. In 2017, HPKP was removed from Google Chrome due in part to these pitfalls [32], and TACK was never widely adopted.

Log-based authentication extends the functionality of CT to enable logs to provide or authenticate public-key information for a domain. AKI [22], its successor ARPKI [3], and CIRT [34] have logs directly provide public-key information for domains and use log proofs to efficiently prove this information. PoliCert [41] allows domains to specify a rich set of policies that can emulate systems such as public-key pinning or AKI. While these systems can provide strong security guarantees (ARPKI, for example, formally proves protection against MITM attacks despite the failure of n CAs), they require significant changes to the certificate issuance process. These systems can require multiple CAs to coordinate to issue a certificate or easily make a misconfigured domain inaccessible. Moreover, the deployment of these systems may require a "flag day" when issuers switch to the new system or allow adversaries to downgrade handshakes to the current, vulnerable PKI.

3 CAPS OVERVIEW

Goals. CAPS primarily aims to enable a smooth transition from the Web's existing PKI to an *improved PKI* (which can range from an

extension of the existing PKI to a new PKI altogether). We assume that during this transition, both the existing and improved PKIs will coexist, and that the improved PKI will make MITM attacks more difficult to carry out. Hence, CAPS must prevent downgrades to the old PKI as well as TLS stripping. More precisely, if a client and server both support the improved PKI, then when they perform a handshake, they should negotiate a session key based on the domain's public key as certified in the improved PKI. As secondary objectives, we also seek to prevent domains from becoming inaccessible due to misconfiguration, private key loss, or private key compromise, and to minimize the changes to existing interactions between clients, domains, and CAs.

Adversary Model. We consider a MITM adversary, who has full control of the network during the TLS handshake; that is, the adversary can intercept, drop, or modify all messages sent among all entities described below. We assume the adversary cannot break standard cryptographic primitives.

Architecture. Fig. 1 illustrates the CAPS architecture and how CAPS achieves our goals. Since CAPS transitions from the current Web PKI, it necessarily includes the entities in the current PKI:

- *Domains* serve webpages to clients. Each domain has a name such as `example.com`.
- *CAs* issue certificates to domains. Each certificate binds a set of names to a single public key.
- *Clients* connect to domains over HTTP or HTTPS, and in the latter case, verify the binding between a domain's name and public key.
- *Browser/OS vendors* (hereafter simply *vendors*) provide the software by which clients connect to domains and verify domains' certificates.
- *Public logs* maintain a publicly auditable, append-only database of certificates, such as those used in CT.

CAPS introduces a new entity, the *log aggregator*, a high-availability entity that uses publicly available data to maintain a database of domains that have deployed HTTPS and/or the improved PKI. As our figure shows, there may be multiple independent log aggregators. While throughout this section we assume that the log aggregator is separate from other entities and that there is a globally accepted set of known and trusted log aggregators, in Section 7 we discuss how we can relax these assumptions, namely, by arguing that browser vendors should take on this responsibility.

As shown in Fig. 1, most interactions between entities in the current PKI remain the same in CAPS. CAs remain entirely unchanged; they issue certificates to domains and log newly-issued certificates as they do in the current PKI (with CT). Clients and domains establish an encrypted communication channel through the standard TLS handshake, and vendors provide clients with browser software.

To prevent an adversary from manipulating an attempted HTTPS connection into an HTTP connection (and thus bypassing TLS completely), the log aggregators use data from public logs to construct a *signaling set*, which succinctly represents the set of all domains that support HTTPS. The log aggregators build this set by downloading the set of all currently valid (i.e., non-expired) certificates from the logs and extracting all domains named in these certificates. The log aggregators then make this set, as well as updates to this set over time, available to client browsers. When connecting to a server, the browser first checks whether the server is in the signaling set; if it is, then the browser will refuse to engage in an HTTP connection.

To give domains more control over their public keys than in the current PKI, log aggregators use data from public logs to construct a set of *CAPS policies*, which allows each domain to establish one or more *authoritative public keys* in the *current* Web PKI. CAPS policies take a simple and intuitive approach: *treat any public key backed by a maximal number of independent certificate chains*[1] *in the current PKI as authoritative*. A domain wanting to increase client confidence in one of its public keys can obtain additional independent certificates for that key, and the log aggregators will automatically update their CAPS policies for that domain.

Intuitively, log aggregators simply provide signed pairs of the form $(name, c)$ where $name$ is a domain name and c is the *CAPS policy value*, i.e., the number of independent certificate chains backing an authoritative key for $name$. In case of a tie, $name$ may have more than one authoritative key: if public keys K_1 and K_2 are both backed by three independent chains then both public keys are treated as authoritative for $name$. To prevent an adversary from downgrading handshakes in the improved PKI to a handshake in the existing Web PKI, each log aggregator indicates which domains in its signaling set have a CAPS policy value (c) greater than 1.

When establishing an HTTPS connection with a server, clients use the signaling set to check if the domain has a CAPS policy value that is exactly one or greater than one. In the former case, the client connects to the domain using the standard TLS handshake, but in the latter case, the client requests policy information from the domain using a TLS extension similar to ones under consideration for IETF standardization [4, 40]. Domains obtain the information from log aggregators and forward it to the client; if a client receives and verifies a policy of $(name, c)$, it will expect to receive c independent certificate chains from the server. If it receives fewer chains, or if any fail to validate, the client aborts the connection.

The current PKI actually supports three classes of certificates: standard domain-validated (DV) certificates, organization-validated (OV) certificates, and extended validation (EV) certificates [16]. EV certificates require domains to undergo more rigorous screening than the other two. Hence, the actual CAPS policy value is a pair $(c_{EV}, c_{!EV})$. For a given domain, CAPS will treat as authoritative a public key with the largest observed c_{EV}, with ties broken based on the largest value of $c_{!EV}$, which represents the policy value of all non-EV certificates for the domain.

4 CAPS DETAILED DESIGN

In this section, we describe CAPS in detail, including how it signals which domains have deployed HTTPS and improvements to the Web PKI (Section 4.1), how it provides stronger public-key authentication over the existing Web PKI (Section 4.2), and how clients establish secure end-to-end connections with servers (Section 4.3). We conclude by describing how CAPS thus enables the bootstrapping of more advanced policies (Section 4.4).

[1] This term means that the certificate chains share no public keys except at the leaf.

4.1 Building the Signaling Set

The signaling set represents (1) a set of FQDNs (hereafter names) known to have deployed HTTPS (by virtue of having a valid public-key certificate appear in a public log), and (2) the subset of names that have adopted CAPS (by virtue of having multiple independent certificate chains (MC) for the same public key). Formally, the signaling set is a pair $(S_{\mathsf{HTTPS}}, S_{\mathsf{MC}})$ where S_{HTTPS} and S_{MC} are unordered sets of valid names in ASCII[2] and $S_{\mathsf{MC}} \subseteq S_{\mathsf{HTTPS}}$. The set supports a query operation, formally defined as query : $\Sigma^* \rightarrow \{\mathsf{no_https}, \mathsf{one_cert}, \mathsf{multi_cert}\}$ where Σ^* is the set of all ASCII strings and no_https, one_cert, and multi_cert are values indicating whether a string is a name known to have no HTTPS certificate, a public key backed by one certificate chain, or a public key backed by multiple independent certificate chains, respectively.

To build its signaling set, a log aggregator must first determine S_{HTTPS} and S_{MC}, which it can achieve using the set of all certificates in the Web PKI. The log aggregator maintains a database of current certificates by using information from public sources, namely, (1) public logs that collect Web certificates (e.g., CT and Censys – see Section 2.1), (2) CRLs such as those published by CAs [6] or browser vendors [18, 23], and (3) revocation information retrieved from OCSP responders [36]. The log aggregator updates this database regularly (e.g., each day), thus maintaining a list of certificates valid on a given day.[3]

The log aggregator extracts the names from the currently valid certificates in its database; the resulting set of distinct names is S_{HTTPS}. The log aggregator also analyzes the certificate chains in this set to determine S_{MC}, a procedure we describe in Section 4.2. The log aggregator then creates a representation of the signaling set and makes it available to client browsers.

Because the signaling set must be available to each client that supports CAPS, the log aggregator must succinctly represent this set to minimize the bandwidth and storage burdens on each client. However, simply minimizing the storage burden is insufficient; clients must store the signaling set in memory to minimize connection latency overhead. Thus the log aggregator must also represent this set in such a way that clients can query the set with acceptably small latency and memory.

We considered several approaches when determining how to represent the signaling set. A Bloom filter [5] supports efficient set membership queries, but has false positives (which for this application would result in incorrectly flagging a site as under attack) and for the scale of data we consider, results in too large a storage burden (Section 6.1). A filter cascade [35] would eliminate false positives, but has the virtually impossible prerequisite of knowing all names in the DNS namespace, which requires the cooperation of all TLD operators (including many national governments). Finally, using an existing data compression utility (e.g., zpaq) with aggressive compression parameters could minimize the required storage space, but would also require decompression and lookup each time a client tried to connect to a site whose HTTPS deployment status was not known, resulting in significant latency overhead (Section 6.4).

[2]A valid name is a Unicode or ASCII string up to 253 bytes in length overall, with no label longer than 63 bytes [31]. We further add the requirement that the name has a top-level domain (TLD) that is a current global TLD according to ICANN. We use ASCII here because we can encode names in Punycode.

[3]A certificate is valid on a given day if the signature on it is valid, it chains to a recognized root certificate store, and that day falls between the certificate's `notBefore` and `notAfter` fields [6].

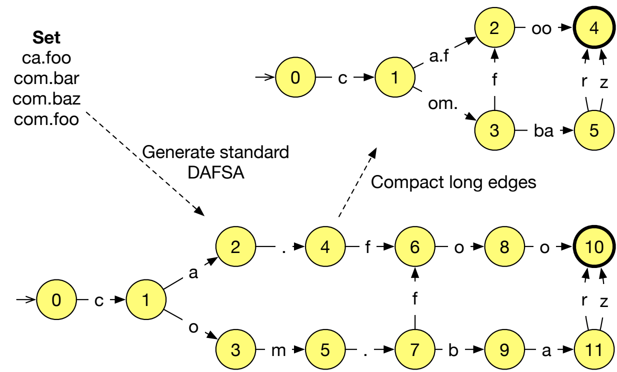

Figure 2: The first stages of our DAFSA construction. Each stage modifies the DAFSA to ultimately minimize the size of its binary representation.

In our log aggregator prototype, we ultimately chose to represent the signaling set as a data structure known as a *deterministic acyclic finite-state automaton*, which succinctly stores a set of strings, supports efficient membership queries in this set, and supports efficient, compact construction [8]. As shown in Fig. 2, a DAFSA takes advantage of both common prefixes and suffixes that appear in a set of strings; since such patterns are frequent in a large set of domain names, much of the redundancy in S_{HTTPS} can be removed with this approach. Additionally, DAFSAs can also be represented succinctly, using an approach we summarize below [9]. Given the characteristics of our input set as described in Section 6.2, we make an additional design change, *path compaction*, to our DAFSA representation that further reduces its size.

Formal DAFSA Definition. To precisely describe these changes, we begin by presenting a formal framework to describe DAFSAs. Formally, a DAFSA is a tuple $(\Sigma, S, s_0, \delta, F)$ where (1) Σ is a set of possible *input symbols*, (2) S is a set of *states*, (3) s_0 is an *initial state* where $s_0 \in S$, (4) $\delta : S \times \Sigma \rightarrow S$ is a partial function called the *state transition function* that maps a state-symbol pair to a new state, and (5) F is a set of *accept states* where $F \subseteq S$. The DAFSA also has the restriction that the state transition function is *acyclic*, that is, there is no sequence of states and symbols $(s_1, \ldots, s_n), (\sigma_1, \ldots, \sigma_n)$ where $\delta(s_i, \sigma_i) = s_{i+1}$ for $1 \leq i < n$ and $\delta(s_n, \sigma_n) = s_1$.

We represent queries to the signaling set within the DAFSA as follows: let Σ contain a unique symbol μ that is not present in any name in S_{HTTPS}. Then, if there exists a sequence of states and symbols $(s_0, \ldots, s_n), (\sigma_1, \ldots, \sigma_n)$ that satisfies (1) $\delta(s_i, \sigma_{i+1}) = s_{i+1}$ for each i where $0 \leq i < n$, (2) $\sigma_1 \| \ldots \| \sigma_{n-1} = name$, (3) $\sigma_n = \mu$, and (4) $s_n \in F$, we say that query($name$) = multi_cert. Otherwise, if (1) $\delta(s_i, \sigma_{i+1}) = s_{i+1}$ for each i where $0 \leq i < n$, (2) $\sigma_1 \| \ldots \| \sigma_n = name$, and (3) $s_n \in F$, then query($name$) = one_cert. Otherwise, query($name$) = no_https.

DAFSA Representation. We begin by building the DAFSA as described in previous work [8] (Fig. 2). Though this previous work assumes transitions based on a single character, we consider the possibility of multi-character symbols and thus our symbol set

consists of strings of length up to 253 characters:[4] $\Sigma^{\leq 253} = \bigcup_{i=0}^{253} \Sigma^i$, where Σ is the set of all ASCII characters allowed in domain names, and Σ^i indicates strings of length i.

We succinctly represent this DAFSA as a bitvector by following the high-level approach of previous work [9]. Intuitively, we represent the DAFSA as a sequence of state encodings, which mostly consist of outgoing transition encodings. By our definition of δ, for a state s, if $\delta(s, \sigma) = t$, then each outgoing transition must be represented by an encoding of the label σ and the destination state t. We observe that in this construction, the overall number of outgoing transitions, as well as the size of the representation of each transition's label and destination state, strongly influence the size of the final bitvector. We thus leverage patterns in the underlying data to minimize the overall size of the DAFSA representation.

Path Compaction. We extend the design of prior work with *path compaction*, which minimizes the DAFSA representation by reducing the overall number of transitions. Intuitively, path compaction removes a connected set of states from the DAFSA and replaces transitions into or out of this set with transitions equivalent to paths through the set. As we formalize below, we can model this process as the transformation of one DAFSA into another and use this model to determine how we should select a set of nodes to ensure a minimal DAFSA representation.

Given a DAFSA $(\Sigma, S, s_0, \delta, F)$, we define a *path* between s_1 and s_m to be a sequence of alternating states in S and symbols in Σ, written $(s_1, \sigma_1, \ldots, s_m)$, where for all i where $1 \leq i < m$, $\delta(s_i, \sigma_i) = s_{i+1}$. We say that a set of states $T \subseteq S$ is a *connected component* if either $T \subseteq F$ or $T \cap F = \emptyset$, and for any two states $t, u \in T$, any path between t and u contains only states in T. The *upstream states* of a connected component T, written $U(T)$, is the set of all states $s \in S \setminus T$ for which there exists a state $t \in T$ and a symbol σ where $\delta(s, \sigma) = t$. The *downstream states* of T, written $D(T)$, is the set of all states $s \in S \setminus T$ for which there exists a state $t \in T$ and a symbol σ where $\delta(t, \sigma) = s$. A path *through* a connected component T is a path $(s_1, \sigma_1, \ldots, s_m)$ where $s_1 \in U(T)$, for all i where $1 < i < m$, $s_i \in T$, and $s_m \in D(T)$.

Path compaction consists of repeatedly (1) selecting a connected component T within the DAFSA, (2) calculating the estimated reduction in representation size from compacting the paths through T, and (3) if the change reduces the size of the DAFSA representation, removing these states from S and replacing the paths through T with an equivalent set of transitions. Specifically, when removing T, we transform the DAFSA $(\Sigma, S, s_0, \delta, F)$ to $(\Sigma, S \setminus T, s_0, \delta', F)$, where for all paths $(s_1, \sigma_1, \ldots, s_m)$ through T, $\delta'(s_1, \sigma_1 \| \ldots \| \sigma_{m-1}) = s_m$.

Our goal is to select components that result in the greatest reduction in size. To determine the impact of removing a connected component, we consider both the reduction in the number of edges in the DAFSA and any changes to the Shannon entropy of the distributions of symbols and destination states in the DAFSA, as removing a component cause these distributions to change. To quantify this change, we define several helpful variables.

For a set $X \subseteq S$, let $\tau(X)$ denote the set of transitions that start or end in X, that is, the number of triples (s, t, σ) where $\delta(s, \sigma) = t$ and s or t (or both) is in X. For a connected component C, let $\pi(C)$ denote the set of paths through C. Then the change in the number of edges by removing C through path compaction is $|\pi(C)| - |\tau(C)|$. By collecting the distribution of symbols in $\tau(S)$ and $\tau(C)$, as well as the concatenated symbols in $\pi(C)$ (which we can find via depth-first search from $U(C)$), we can compute the change in entropy in symbols and similarly for destination states, which we write as ΔH_σ and ΔH_δ, respectively. Suppose that we know the original entropies H_σ and H_δ, and that we define $\Delta H = \Delta H_\sigma + \Delta H_\delta$ and $H = H_\sigma + H_\delta$. Then we can compute the difference in size between the two DAFSAs as

$$|\tau(S)|\Delta H + (|\pi(C)| - |\tau(C)|)(H + \Delta H) \quad (1)$$

and only remove C if this quantity is negative.

We found several classes of components that, for our underlying set of domain names, provided substantial reductions in the size of the DAFSA representation. The first was to select what we call *isolated paths*, that is, paths of the form $(s_1, \sigma_1, \ldots, s_m)$ where for all i such that $1 < i < m$, s_i only had one incoming and one outgoing transition. Using techniques from prior work [8] to build the DAFSA results in a significant number of isolated paths. Hence performing path compaction on all such paths results in a size savings of nearly 10% of the original DAFSA size. We also found that selecting a constant α and then selecting components consisting entirely of states that had one incoming transition and α outgoing transitions (or vice versa) yielded more modest but still nontrivial size reductions for $\alpha = 2$ and $\alpha = 3$.

[4]Recall that DNS names can be a maximum of 253 characters.

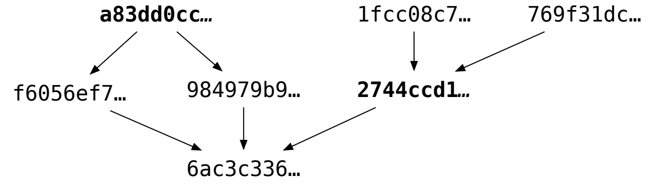

Figure 3: Sample certificate fingerprint graph. $A \rightarrow B$ indicates that a certificate with fingerprint A is the authority public key for a certificate with fingerprint B. Though the leaf certificate `6ac3c336`... has four chains, the bolded fingerprints show that only two of the chains are independent.

4.2 Building the CAPS Policy Database

The CAPS *policy database* represents a binding between a name and a policy, that is, the number of independent certificate chains a client should expect during a handshake with a server corresponding to the name. To construct and maintain this database, each log aggregator tracks the certificates and chains active for a domain at any given time, using the data collected for the signaling set.

A log aggregator uses this data to maintain an internal database with (many-to-many) maps of (1) certificates to names, (2) certificates to public keys, and (3) certificates to chains. Regular updates to this database (described in Section 4.1) ensure that the log aggregator has a list of currently valid certificates. The log aggregator can then use the database to construct a mapping of names to policies. Specifically, the aggregator creates a graph of certificate fingerprints as shown in Fig. 3, and computes the *policy value* (c), the minimum number of CA public keys that must be compromised for a CAPS-enabled browser to accept a fraudulent certificate for

this site. The policy value can be computed using a straightforward approach (e.g., computing a minimal vertex separator), allowing the log aggregator to easily construct a mapping of certificate fingerprints to policy values. The log aggregator can then perform a series of simple join operations to map each name to the maximum policy number associated with a certificate containing the name.

The log aggregator constructs this name-to-policy mapping each time it receives data from public sources, which occurs at regular, scheduled intervals (recall Section 4.1). Once it has created the mapping, the log aggregator certifies the mapping's name–policy-value pairs by timestamping and signing them. For auditability, the log aggregators can additionally take an approach similar to that of logs in CT, using a Merkle hash tree to store their policy proofs over time in a cryptographically verifiable, append-only fashion.

A domain can now provide the information necessary for clients to establish the domain's authoritative public key. The domain periodically downloads the latest *policy proofs*, which are signed and timestamped name-policy pairs, from each of the log aggregators. For a name *name*, a policy value c, a timestamp ts, and a log aggregator A, the policy proof is

$$\mathcal{P}(A, \mathit{name}, c, \mathrm{ts}) = \left\{data, \mathsf{Sign}_A(data)\right\} \text{ where} \\ data = A \| K_A \| \mathit{name} \| \mathrm{ts} \| c \tag{2}$$

where K_A and K_A^{-1} denote respectively the public and private keys of A, and $\mathsf{Sign}_A(m)$ denotes a signature on m with K_A^{-1}. The domain caches these proofs and serves them to clients in the handshake protocol described below.

4.3 Connection Establishment

To establish a connection to a domain, a client (e.g., browser) first queries the signaling set for the domain's name. If this query returns one_cert then the client performs a standard TLS handshake (and refuses any attempts to downgrade to plain HTTP). If the query returns multi_cert, indicating the domain has more than one certificate chain, then the client performs the CAPS-extended TLS handshake to establish a connection with the domain. The result of the query to the signaling set is cached until the next signaling set update, which eliminates the need for this query in future connections. Caching is particularly effective at minimizing overhead for operations such as session resumption [33].

The CAPS-extended TLS handshake protocol allows a client to verify a domain's authoritative public key. The TLS handshake protocol [10] provides support for open-ended extensions (implemented in cryptographic libraries), and thus we designed our protocol as an extension within the existing TLS handshake.

During the initial ClientHello message, the client includes a CAPS extension message, which consists of a single integer k, indicating the number of policy proofs that the domain should send back. In an initial deployment, we expect that typically $k = 1$, but allowing the use of larger values for k provides resilience against compromised log aggregators. If the client asks for more proofs than the domain is willing to provide, then the connection simply fails. See Section 5.2 for further discussion.

The domain then sends back a ServerHello message that contains cached policy proofs from k distinct log aggregators, as well as c certificate chains. The client can then verify the domain's policy value and the certificate chains supporting it. Formally, suppose a domain has hostname *name* and policy value c, and selects a set of log aggregators $\{A_1, \ldots, A_k\}$. Let $\Pi_i = \mathcal{P}(A_1, \mathit{name}, c, \mathrm{ts}_i)$ for $1 \le i \le k$, where ts_i is the timestamp of the policy proof from A_i. The domain sends the following back in the ServerHello extension message:

$$\{\Pi_1, \ldots, \Pi_k, C_1, \ldots, C_{c-1}\} \tag{3}$$

where C_j is a certificate chain for *name*. In the extension message, the domain only sends $c-1$ certificate chains because the remaining chain is sent in the ServerCertificate message of the TLS handshake.

The client then checks that (1) the signature on each policy proof is valid, (2) the timestamp for each policy proof is sufficiently recent, (3) the name in each policy proof matches the domain name to which the browser is connecting, (4) the policy value for each policy is one more than the number of certificate chains sent in the domain's extension message, and (5) each certificate chain is valid as specified in the X.509v3 standard [6]. If the above checks pass, the client continues with the standard TLS handshake, which requires the client to verify an additional certificate chain in the ServerCertificate message and perform all other checks required by the TLS handshake protocol.

While our current design uses a custom TLS extension, in the future, CAPS may instead leverage recently proposed TLS extensions [4, 40] designed to allow the transmission of additional certificates (primarily to facilitate content hosting by CDNs).

4.4 Bootstrapping Advanced Policies

Once an authoritative public key for a domain has been established through the CAPS handshake, signatures made by the corresponding private key are useful beyond simply improving the domain's security in the current PKI. In particular, a signature from the authoritative public key can be used to verify the binding between a domain and a richer set of policies. For example, in systems such as ARPKI [3] and PoliCert [41], these policies can specify a set of CAs that are authorized to issue certificates for the domain, or pin specific public keys to the domain. In this way, the authoritative public key established in CAPS can be used to bootstrap confidence in these advanced policies while preventing downgrades to the old PKI.

This bootstrapping approach obviates the need for logs to directly store the policies, which can be quite large in these previously proposed systems. Moreover, in CAPS, this bootstrapping can take place at any time during deployment. This means that in the case of a lost private key, a domain can simply obtain c new independent certificate chains, and in the case of a compromised private key, $c+1$ new independent chains. This contrasts with previous proposals, which often rely on heavyweight manual processes.

5 SECURITY ANALYSIS

We analyze the security of CAPS, beginning with our main security claim: an adversary must compromise a threshold number of CAs or log aggregators to mount a successful MITM attack. We then describe potential weaknesses not covered by our security claim.

5.1 Main Security Claim

In making our main security claim, we assume that the client requests k policy proofs and that the domain has domain name D. We assume that the domain has a policy value of c and thus at least c independent certificate chains, along with at least k policy proofs for this policy value. We also assume that our adversary controls $k-1$ private keys of log aggregators and the private keys of $c-1$ CAs. The adversary can also, of course, create its own public key pairs, e.g., (K_M, K_M^{-1}). Recall that controlling a log aggregator's private key allows the adversary to forge policy proofs for any domain with any policy value, and controlling a CA's private key allows the adversary to forge certificates binding any domain name to any public key. We assume that the adversary can intercept, suppress, replay, and modify any handshake message sent between the client and domain, and the adversary can use its private keys to sign any message it can construct with the information it obtains. We also assume that the adversary cannot obtain a certificate binding D to K_M from any CA besides the ones it controls. Under these assumptions, we claim that a CAPS-enabled client will abort the handshake protocol if it receives any certificate chain containing the adversary's K_M as the leaf public key for D. This claim supports the security of CAPS, because to mount a successful MITM attack, the adversary must convince the client to complete the CAPS handshake based on a key that the adversary controls.

We first show that the adversary cannot convince the client that that the domain has a policy value other than c. Because the client requests k policy proofs for D, the client will abort the handshake if the ServerHello message does not contain k independent policy proofs. Because we assume that the adversary only has access to $k-1$ log aggregator signing keys, the adversary cannot use those keys to generate k independent proofs. Specifically, if the adversary sends a set of proofs and there are fewer than k valid proofs, or if any of them fails to prove that c is the policy value for D, then the client will abort the handshake.

From our assumption that the adversary can access the signing keys of $c-1$ CAs, we know that the number of *independent* certificate chains that the adversary can generate is *at most* $c-1$. It is straightforward to show this by induction on c, with $c=2$ as the base case.[5] If the adversary generates $c-1$ independent certificate chains for K_M and sends these chains to the client, the client will validate the chains, but abort the handshake when the cth independent chain fails to arrive.

5.2 Potential Weaknesses

Our main security claim shows that under our assumptions, the adversary cannot mount a successful MITM attack on a client and domain. However, we did not yet address the ways in which CAs, log aggregators, or other parties shown in Fig. 1 may fail. We now discuss possible failures for each of these parties and how they may affect the security of CAPS, along with potential mitigations.

CAs. Historically, all publicly known CA failures have been singletons (i.e., separated in time and causation from other failures), but a systemic flaw may allow an adversary to compromise many CAs at once. Such a widespread compromise would be quickly detected by public logs and certificate scanning services. Subsequently, the browsers or CAs could issue revocations of the affected certificates or CA keys using existing methods.

Public Logs. A misbehaving public log may record a fraudulently-issued certificate, refuse to include a certificate in its database, or attempt to change details of previously logged certificates. CT expects auditors (a role anyone can adopt) to monitor logs for any such misbehavior [26]. Recording fraudulent certificates may cause a log aggregator to compute a policy value that is too high, making a domain inaccessible in CAPS. However, this also requires the failure of c CAs to mount a MITM attack, and would be quickly detected by an auditor. Once detected, browsers and log aggregators can simply ignore the misbehaving log.

Log Aggregators. A log aggregator's private key may be lost or stolen, causing it to provide incorrect policy information or signaling sets. A successful MITM attack would require k aggregators to send incorrect policy information. Clients can choose a value of k to minimize this risk; in practice, we expect a value of 2 or 3 will suffice. Indeed, major browsers already expect to see at least two CT log proofs during a normal TLS handshake [1]. Clients can take a similar approach to verify signaling sets and their updates.

A log aggregator may not provide updates in a timely manner. Since domains provide cached policy proofs to clients, domains who change their certificates or obtain additional certificates can ensure that their policy proofs are up to date by fetching and caching proofs with the correct values before proceeding to establish connections with the new/additional certificates. The delay may also affect signaling sets, but only if the policy value changes between 0, 1, and ≥ 2 independent chains. Finally, logs are held to a standard of 99% availability [39], and doing the same for log aggregators would minimize the risk of these failures.

Browser Vendors. A misbehaving browser vendor can ship a malicious version of a browser to clients, and we consider this failure mode outside of the scope of CAPS. A malicious browser can arbitrarily deviate from TLS or CAPS, insert its own root certificates, or display arbitrary pages.

6 EVALUATION

Below, we evaluate the performance of CAPS in our prototype. All of the code for this section, including our prototype implementation of the core CAPS components and the scripts we used for testing, are available at https://github.com/syclops/caps.

6.1 Signaling Set Domains

We can obtain a view of the Web PKI using data from public logs (Section 4.1). Specifically, we obtain public-key certificates from Censys [11] and logs in CT [26]. From Censys, we collected 1,026 scans of the IPv4 address space from September 12, 2015 to July 3, 2018. From CT, we collected all entries from known CT logs that were not disqualified or unreachable as of July 3, 2018,[6] which totaled approximately 1.74B certificates from 26 logs from March 26, 2013 to July 3, 2018.

[5] Note that the $c=1$ case is equivalent to the current Web PKI (i.e., the domain only provides a single certificate chain).

[6] https://www.certificate-transparency.org/known-logs

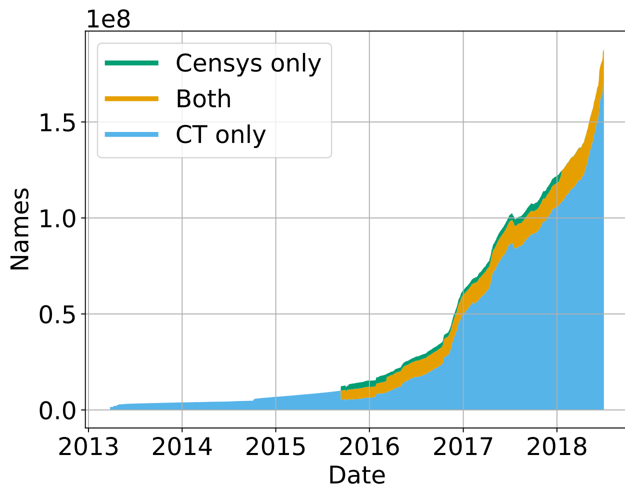

Figure 4: Number of unique names (including hostnames and wildcard names) seen by Censys and CT over time.

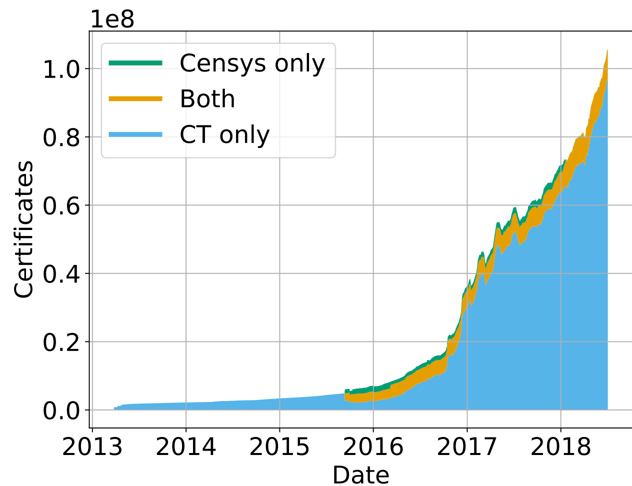

Figure 5: Number of unique certificates seen by Censys and CT over time.

On each of these days, we consider an "active set" of certificates consisting of all certificates that were valid on that day[7] and chained to one of the three major root certificate stores, determined by Apple, Microsoft, or Mozilla, respectively. In the Censys dataset, because we observed a great deal of churn (i.e., certificates disappearing and appearing in consecutive scans), we included a certificate in the active set from the time it was first observed in our data until its expiration. We then consider the number of unique, valid domain names to build the signaling set.

Fig. 4 shows the number of domain names observed by Censys, CT, and both over time. We found that CT observes vastly more certificates (and consequently names) than Censys. It is unclear what causes this large discrepancy. One possibility is that many certificates are simply never deployed in public-facing HTTPS sites. Another likely contributing factor is the increasing use of Server Name Indication (SNI) [13], which cause Censys' probes to be rejected when they do not include the correct server name.

Fig. 5 shows the size of the "active set" of certificates as observed by our two data sources. For the most part, the trend in the number of active certificates on a given day matches the trend in the number of active domain names as seen in Fig. 4. There does, however, seem to be slightly more churn in the number of active certificates over time, particularly during 2017, in which the usage of Let's Encrypt was becoming increasingly widespread. We point out that the number of names is almost consistently greater than the number of certificates due to the fact that certificates can contain multiple names (and many popular sites such as Google have certificates with all of their domain names).

From Censys and CT, we obtained a total of 156,289,973 valid domain names for which a certificate had been issued. To address the possibility that many of these names may not be used for public-facing sites, we performed a scan of port 443 (the default for HTTPS) using ZGrab [11] for all of these domain names, and discarded any domain names that consistently failed to respond. This resulted in 64,050,329 names that we used for testing, as described below.

[7]Recall our definition in Footnote 3.

6.2 Signaling Set Representation

As described in Section 4.1, our motivation for using a DAFSA-based representation of the signaling set was twofold: first, the representation has no false positives or negatives, and second, it can be searched in its compressed state, reducing client memory usage. To evaluate the effectiveness of these design decisions, we measured the space requirements for the signaling set in various representations. We measured both the fully compressed size (used when transmitting the set to the client and when stored on disk) and the size in memory (when being used during certificate verification).

In particular, we compared the plaintext representation of the signaling set (as of July 3, 2018) with a compressed representation using Bloom filters, the generic compression utility zpaq [27],[8] and various configurations of our DAFSA-based representation. We also compressed the DAFSA-based representation using zpaq to find its size on disk and in transit.

We specifically tested a Bloom filter with false positive rates of 0.001%, 0.01%, and 0.1%. Since the number of domain names is on the order of 100M [44], we expect that the number of false positives will be on the order of 1k, 10k, and 100k, respectively. We estimate that a false positive rate of 0.001% will be sufficient for most users. We tested zpaq using two compression methods, 1 and 5, where method 1 completes in a short amount of time (25 seconds) but compresses the input less while method 5 takes a long time (20 minutes) but yields excellent compression. Furthermore, with zpaq method 5, we tested with 64 MiB and 2048 MiB blocks, where larger blocks typically yield better compression. Finally, with our DAFSA-based representation, we tested a plain encoding as well as an encoding using our path-compressed DAFSA, and compressed each of these encodings with zpaq method 5 using both block sizes.

[8]While we tested compression with other utilities, zpaq had the smallest size.

Table 1: Size (on-disk or in transit) of CAPS signaling set on July 3, 2018 (64,050,329 names) with various compression approaches. The representation size in memory is not shown.

Representation	Size (MB)
Plaintext	1,509
Bloom Filter (0.001% FP, best-case)	192
Bloom Filter (0.01% FP, best-case)	153
Bloom Filter (0.1% FP, best-case)	115
zpaq (method 1, 16 MiB blocks)	286
zpaq (method 5, 64 MiB blocks)	105
zpaq (method 5, 2048 MiB blocks)	104
DAFSA	214
DAFSA w/ path compaction (PC)	190
DAFSA w/ zpaq (method 5, 64 MiB blocks)	162
DAFSA w/ zpaq (method 5, 2048 MiB blocks)	160
DAFSA w/ PC, zpaq (method 5, 64 MiB blocks)	150
DAFSA w/ PC, zpaq (method 5, 2048 MiB blocks)	148

The results are shown in Table 1. Starting from a plaintext corpus of over 1.5 GB, the various options all achieve impressive compression ratios. However, the results also indicate that to achieve a competitive size (i.e., 153 MB or less), Bloom filters require an unacceptably high false positive rate: one in every 10K sites would be falsely signaled as supporting HTTPS and hence would be rendered inaccessible. While zpaq does not have any false positives or false negatives and yields excellent compression when run using method 5, its in-memory representation is simply the uncompressed set of domains, yielding a memory requirement of 1.5 GB. Our DAFSA-based representation captures a "sweet spot" between these two alternatives, suffering no false positives or negatives and, in the best case, an on-disk representation of just 148 MB with an in-memory representation of 190 MB.

For some clients, an initial download size of 148 MB may be too much. One approach that such clients might take to protect themselves would be to only track sites that have more than one certificate (i.e., sites with $c > 1$). This would ensure that such clients still benefit from greater resiliency against CA compromises, particularly for "high-value" domains that take the effort to obtain extra certificates. For these clients, this optimization would reduce disk and memory usage, but CAPS would no longer protect such clients from TLS stripping attacks targeting "normal" domains (those with a single certificate). To estimate clients' memory and disk usage in this case, we subsampled the full set of names and computed the size of the DAFSA with path compaction and the size of the zpaq-compressed DAFSA (which represents the best-case disk usage).

Table 2 shows the results. If the fraction of domains that use multiple independent certificate chains for the same name is small, as we would anticipate, then CAPS clients significantly reduce their memory and disk usage. For example, even if 10% of all HTTPS websites deployed additional certificates, the compressed DAFSA representation would require just 33 MB. Of course, at very low levels of adoption, the advantage of the DAFSA-based approach over a list of names decreases. This makes sense, given that the DAFSA takes advantage of common substrings.

Table 2: Size of the signaling set in various representations when the names are subsampled from the full set of names.

Fraction	0.01	0.05	0.1	0.2	0.5
Names (100K)	6.39	32.0	64.1	128	320
Uncompressed (MB)	15.1	75.5	151	302	755
DAFSA (MB)	5.25	22.9	41.7	73.1	140
Compressed DAFSA (MB)	4.21	18.2	33.0	58.1	112

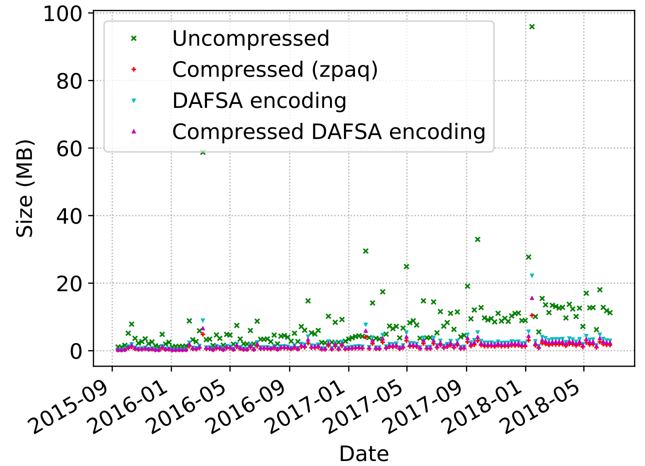

Figure 6: Size of update set (added name set and deleted name set) in different formats over time.

6.3 Signaling Set and Certificate Updates

Because the signaling set will be updated over time, we experimented to determine the size of updates sent to clients. An update to the signaling set consists of the names added to the signaling set since the most recent version, as well the names deleted due to certificate expiration or revocation. We computed the set of added and deleted names for our range of scans, aggregating these sets by week. We then computed the combined sizes of these sets in four different representations: (1) as an uncompressed text file of name strings, (2) as a compressed zpaq archive containing the above file, (3) as a DAFSA of the set of strings, and (4) as a compressed zpaq archive containing the above DAFSA. For each method, we used the variant that produced the smallest representation; e.g., we used the zpaq method that produced the smallest archive (method 5 with 64 MiB blocks) and our DAFSA representation used path compaction. We experimented with the full set of names.

Fig. 6 shows the results. The size of added and deleted names is slowly increasing over time, with the set of added names consistently being larger than the set of deleted names (Appendix A). Given the relatively modest sizes of these sets compared to the full signaling set, the most space-efficient method for representing and transmitting these updates to clients is a zpaq-compressed archive of the raw text file of names rather than a DAFSA-based representation. This method of transmission is also advantageous since clients can simply add the set of added names to their existing DAFSA,

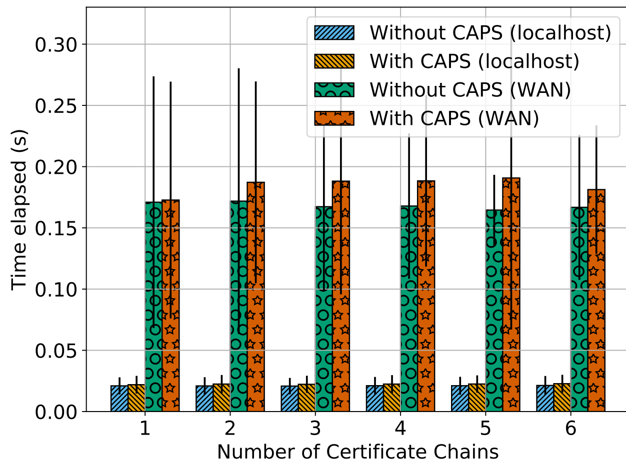

Figure 7: Handshake latency vs. the number of certificate chains sent by the domain. Error bars are standard error.

then build (and subsequently add to) a new DAFSA for the set of deleted names. Our results show that updates are typically less than 3 MB per week or ~439 KB/day; by comparison, downloading the Google homepage requires approximately 400 KB.

Under certain circumstances, changes in a domain's CA may cause an incorrect policy value update. Specifically, if a domain obtains a new independent certificate chain (e.g., by switching CAs) for the same public key without revoking the old one, log aggregators may report a policy value one higher than the domain has, potentially affecting the signaling set and CAPS handshakes. We reviewed our database for these potential cases and found this affected only 0.11% of the overall set. This is a conservative upper bound, containing certificates that are almost certainly from the same issuer with slightly different issuer organization names, such as "Gandi" and "GANDI SAS", indicating that a good portion of this small percentage of failures can be mitigated in a straightforward way.

6.4 Connection Establishment

To measure the performance of connection establishment in CAPS, we implemented the handshake as a custom TLS extension in the OpenSSL library. For concrete evaluation of this extension, we use `nginx` and `curl` with minor modifications to use our TLS extension.

Additionally, we constructed sample sets of domain names based on four parameters: (1) the number of proofs requested during the ClientHello message (k), (2) the number of certificate chains sent with the ServerHello message (c), (3) the average number of certificates per chain, and (4) the average size of each certificate chain. While varying each of these parameters, we measured the amount of extra data sent in the CAPS handshake, and the latency of the handshake both with and without the CAPS TLS extension.

We tested this both over the Internet (by connecting to a virtual private server with latency varying from 30 to 300 ms), as well as over the local loopback interface. The tests over the internet (*WAN*) provide an indication of the effect of the extension on "real world" servers, whereas the *localhost* tests provide a lower bound on time added due to sending/receiving/processing extra data. For all tests, we used a single active client at any given time to isolate the overhead added by CAPS from factors such as load balancing. A total of 15385 TLS connections were established for our testing: 5768 over WAN, 9617 over localhost.

Our results were similar in each case, with a representative example shown in Fig. 7 (others in Appendix B). In comparison to the mean time elapsed, there is an approximately 5% increase in connection establishment time: an average of 11ms longer for WAN and 1.2ms for localhost. Since our TLS extension does not add any extra round-trips to the handshake, the time added is small compared to random measurement fluctuations (i.e., the error bars).

7 DEPLOYMENT CONSIDERATIONS

We now discuss potential issues related to the deployment of CAPS in practice, and propose possible ways to address them.

Candidates for Log Aggregators. Our design of CAPS does not require specific entities to serve as log aggregators. However, from our analysis in Section 5.2, we conclude that log aggregators should have high availability and be widely known. We believe that browser vendors or public logs would be particularly suited for these roles. Both already take an active role in "policing" the Web PKI. Both offer high availability, with logs being held to a minimum of 99% availability, minimizing the risk of synchronization issues. Finally, lists of major browser vendors and logs are already widely known, making it easy to present clients with a list of available log aggregators to provide policy proofs.

Independent Certificate Chains. In Section 4.2 we described how we can use a graph of certificate fingerprints to compute the policy value for a domain. Specifically, we compute the policy value as the minimum number of CA private keys that would need to be compromised to create an independent set of certificate chains for an adversary's public key. However, this value does not necessarily result in truly independent certificate chains, as many CAs control multiple private keys. If a single CA with poor issuance practices possesses multiple private keys, an adversary may gain fraudulent certificates from multiple private keys in a single attack. If these certificates are in multiple independent chains, then an adversary may be able to mount a MITM attack.

One solution to this vulnerability is to use the certificate's issuer organization name to differentiate CAs, but the success of this approach depends on the CA itself. As we describe in Section 6.3, we found certificates that are likely from the same issuer, but have slightly different issue organization names. Furthermore, one CAs may own another and use similar security practices. If these CAs do not certify one another, their certificate chains may be considered independent by log aggregators. While we leave deeper exploration of this area to future work, we can easily configure CAPS to determine the independence of certificate chains by the issuer key or organization name, and this is likely to provide sufficient security for the vast majority of cases.

CAPS in the Modern Web. The use of HTTPS has increased significantly over the past years (Section 6.1), in part due to the advent of services such as Let's Encrypt. Despite this increase, we anticipate the storage and memory overhead of the CAPS signaling set

to remain manageable for three reasons. First, Table 2 suggests a sublinear growth in the size of the signaling set relative to the number of names. Second, despite Fig. 4 showing a large increase in certified names, Fig. 6 suggests that the growth of names that are accessible on the public Web is much slower. Finally, the cost of disk space and memory falls over time, offsetting CAPS's overhead.

Many Web clients are now mobile devices with limited computational, memory, and storage resources, but we can still deploy CAPS on these devices. Smaller forms of the signaling set (Section 6.2) can provide some protection against MITM attacks on resource-constrained devices. Moreover, Fig. 7 shows the added latency of CAPS is small compared to the overall latency, and is mostly due to validating additional certificate chains or policy proofs.

8 CONCLUSION

We described how CAPS provides for a secure, smooth transition to a more resilient Web PKI. Using a global view of Web certificates, CAPS prevents TLS stripping and downgrade attacks with modest overhead, thanks to data compression and existing TLS functionality. CAPS' flexible policy approach simplifies incremental deployment and recovery from errors. While room for improvement remains, CAPS addresses critical challenges and provides an important step towards a resilient Web PKI.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for valuable feedback. Stephanos Matsumoto was supported, in part, by a CyLab Presidential Fellowship. Paul Van Oorschot is supported by an NSERC Discovery Grant. Work at Carnegie Mellon University was supported in part by the NSF under Grant No. CNS 1900996, the Alfred P. Sloan Foundation, and the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

REFERENCES

[1] Apple. 2019. Apple's Certificate Transparency policy. https://support.apple.com/en-us/HT205280.
[2] Roy Arends, Rob Austein, Matt Larson, Dan Massey, and Scott Rose. 2005. DNS Security Introduction and Requirements. RFC 4033.
[3] David Basin, Cas Cremers, Tiffany Hyun-Jin Kim, Adrian Perrig, Ralf Sasse, and Pawel Szalachowski. 2014. ARPKI: Attack Resilient Public-Key Infrastructure. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 382–393.
[4] M. Bishop, Nick Sullivan, and M. Thomson. 2019. Secondary Certificate Authentication in HTTP/2. IETF Internet Draft https://tools.ietf.org/html/draft-ietf-httpbis-http2-secondary-certs-05.
[5] Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (July 1970), 422–426.
[6] David Cooper, Stefan Santesson, Stephen Farrell, Sharon Boeyen, Russell Housley, and Tim Polk. 2008. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280.
[7] Scott A. Crosby and Dan S. Wallach. 2009. Efficient Data Structures For Tamper-Evident Logging. In *USENIX Security Symposium*. 317–334.
[8] Jan Daciuk, Stoyan Mihov, Bruce W. Watson, and Richard E. Watson. 2000. Incremental Construction of Minimal Acyclic Finite-State Automata. *Computational Linguistics* 26, 1 (March 2000), 3–16.
[9] Jan Daciuk and Dawid Weiss. 2012. Smaller Representation of Finite State Automata. *Theoretical Computer Science* 450, 7 (September 2012), 10–21.
[10] Tim Dierks and Eric Rescorla. 2008. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246.
[11] Zakir Durumeric, David Adrian, Ariana Mirian, Michael Bailey, and J. Alex Halderman. 2015. A Search Engine Backed by Internet-Wide Scanning. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[12] Zakir Durumeric, Eric Wustrow, and J Alex Halderman. 2013. ZMap: Fast Internet-wide Scanning and Its Security Applications. In *USENIX Security Symposium*, Vol. 8. 47–53.
[13] Donald Eastlake. 2011. Transport Layer Security (TLS) Extensions: Extension Definitions. RFC 6066.
[14] Let's Encrypt. [n.d.]. http://letsencrypt.org.
[15] Chris Evans, Chris Palmer, and Ryan Sleevi. 2015. Public Key Pinning Extension for HTTP. RFC 7469.
[16] CA/Browser Forum. 2018. Guidelines for the Issuance and Management of Extended Validation Certificates, Version 1.6.8.
[17] Electronic Frontier Foundation. [n.d.]. HTTPS Everywhere. https://www.eff.org/https-everywhere.
[18] Mark Goodwin. 2015. Revoking Intermediate Certificates: Introducing OneCRL. https://blog.mozilla.org/security/2015/03/03/revoking-intermediate-certificates-introducing-onecrl/.
[19] Jeff Hodges, Collin Jackson, and Adam Barth. 2012. HTTP Strict Transport Security (HSTS). RFC 6797.
[20] Paul Hoffman and Jakob Schlyter. 2012. The DNS-based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA. RFC 6698.
[21] Smart HTTPS. [n.d.]. https://mybrowseraddon.com/smart-https.html.
[22] Tiffany Hyun-Jin Kim, Lin-Shung Huang, Adrian Perrig, Collin Jackson, and Virgil Gligor. 2013. Accountable Key Infrastructure (AKI): A Proposal for a Public-Key Validation Infrastructure. In *International World Wide Web Conference (WWW)*. 679–690.
[23] Adam Langley. 2012. Revocation checking and Chrome's CRL. https://www.imperialviolet.org/2012/02/05/crlsets.html.
[24] James Larisch, David Choffnes, Dave Levin, Bruce M. Maggs, Alan Mislove, and Christo Wilson. 2017. CRLite: A Scalable System for Pushing All TLS Revocations to All Browsers. In *IEEE Symposium on Security and Privacy (S&P)*.
[25] Ben Laurie. 2014. Certificate Transparency. *Commun. ACM* 57, 10 (Sept. 2014).
[26] Ben Laurie, Adam Langley, and Emilia Kasper. 2013. Certificate Transparency. RFC 6962.
[27] Matt Mahoney. 2019. ZPAQ: Incremental Journaling Backup Utility and Archiver. http://www.mattmahoney.net/dc/zpaq.html.
[28] Moxie Marlinspike. 2009. New Tricks for Defeating SSL in Practice. http://www.thoughtcrime.org/software/sslstrip/.
[29] Moxie Marlinspike and Trevor Perrin. 2013. Trust Assertions for Certificate Keys. https://tools.ietf.org/html/draft-perrin-tls-tack-02, (work in progress).
[30] Ralph C. Merkle. 1988. A Digital Signature Based on a Conventional Encryption Function. *Advances in Cryptology (CRYPTO)* (1988), 369–378.
[31] Paul Mockapetris. 1987. Domain Names – Implementation and Specification. RFC 1035.
[32] Chris Palmer, Rich Baldry, Andrew Meyer, Jochen Eisinger, Ryan Sleevi, Rick Byers, Phillip Hallam-Baker, Ryan Lester, and Joe Medley. 2017. Intent to Deprecate and Remove: Public-Key Pinning. Chromium mailing list, https://groups.google.com/a/chromium.org/forum/#!topic/blink-dev/he9tr7p3rZ8.
[33] Eric Rescorla. 2018. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446.
[34] Mark D Ryan. 2014. Enhanced Certificate Transparency and End-to-End Encrypted Mail. In *Network and Distributed System Security Symposium (NDSS)*.
[35] Kamil Salikhov, Gustavo Sacomoto, and Gregory Kucherov. 2014. Using cascading Bloom filters to improve the memory usage for de Brujin graphs. *Algorithms for Molecular Biology* 9, 2 (February 2014).
[36] Stefan Santesson, Michael Myers, Rich Ankney, Ambarish Malpani, Slava Galperin, and Carlisle Adams. 2013. X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP. RFC 6960.
[37] Ryan Sleevi. 2015. Sustaining Digital Certificate Security. https://googleonlinesecurity.blogspot.com/2015/10/sustaining-digital-certificate-security.html.
[38] Ryan Sleevi, Doug Beattie, Bruce Morton, and Peter Bowen. 2016. Announcement: Requiring Certificate Transparency in 2017. https://groups.google.com/a/chromium.org/forum/#!msg/ct-policy/78N3SMcqUGw/ykIwHXuqAQAJ.
[39] Ryan Sleevi and Devon O'Brien. 2017. Certificate Transparency Log Policy. https://github.com/chromium/ct-policy/blob/master/log_policy.md.
[40] Nick Sullivan. 2019. Exported Authenticators in TLS. IETF Internet Draft https://tools.ietf.org/html/draft-ietf-tls-exported-authenticator-10.
[41] Pawel Szalachowski, Stephanos Matsumoto, and Adrian Perrig. 2014. PoliCert: Secure and Flexible TLS Certificate Management. In *ACM SIGSAC Conference on Computer and Communications Security*.
[42] Filippo Valsorda. 2015. Komodia/Superfish SSL Validation is Broken. https://blog.filippo.io/komodia-superfish-ssl-validation-is-broken/.
[43] Benjamin VanderSloot, Johanna Amann, Matthew Bernhard, Zakir Durumeric, Michael Bailey, and J Alex Halderman. 2016. Towards a Complete View of the Certificate Ecosystem. In *ACM Internet Measurement Conference (IMC)*. 543–549.
[44] Verisign. 2017. The Verisign Domain Name Industry Brief. https://www.verisign.com/assets/domain-name-report-Q42016.pdf.
[45] Jiangshan Yu, Vincent Cheval, and Mark Ryan. 2016. DTKI: A New Formalized PKI with Verifiable Trusted Parties. *Comput. J.* 59, 11 (November 2016), 1695–1713.

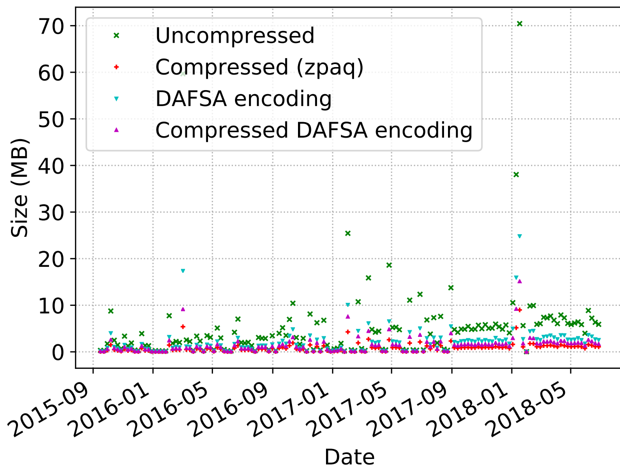

Figure 8: Sizes of the set of added names over time in different representations.

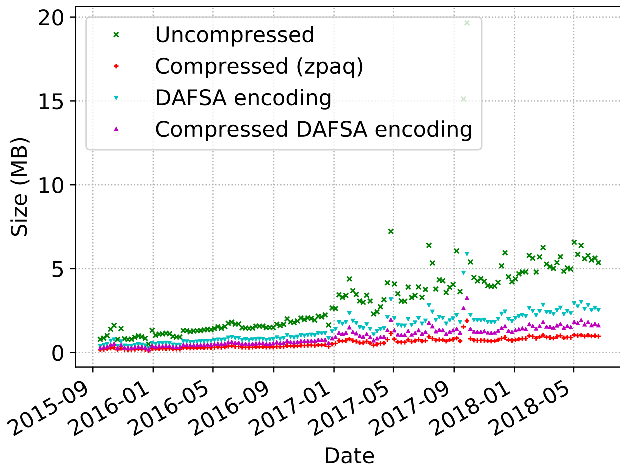

Figure 9: Sizes of the set of deleted names over time in different representations.

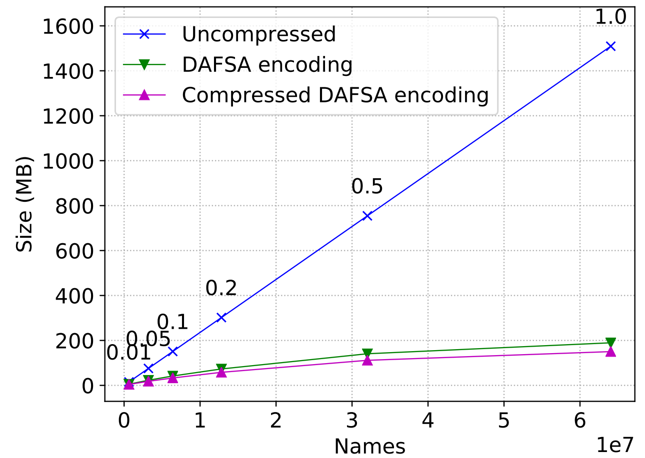

Figure 10: Size of the signaling set given subsampling from the full set of names. The labels above each distinct value on the x-axis denote the fraction of the full set that was sampled.

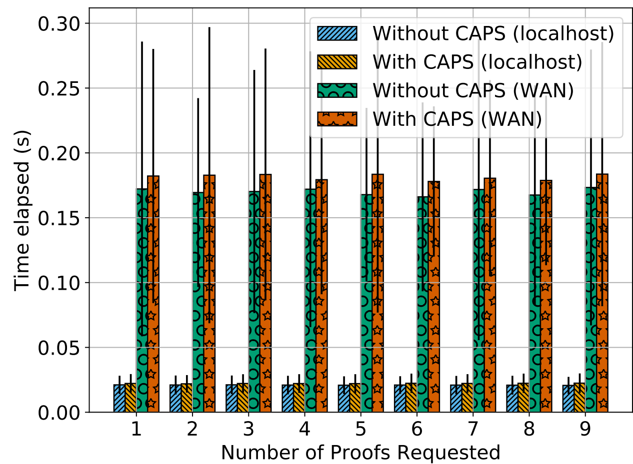

Figure 11: Handshake latency versus the number of policy proofs sent by the domain. Error bars represent standard error.

A UPDATES TO THE SIGNALING SET

In Section 6.3, we showed that the size of updates to the signaling set were slowly increasing over time. To better understand how the size of the signaling set may change over time, we also considered the additions and deletions to the signaling set separately.

Figs. 8 and 9 show the sizes of these sets over time. We see that the set of added names is almost consistently larger than the set of deleted names, which matches the upward trend in the number of domain names in the Web PKI over time. We also note that beginning in 2017, in which Let's Encrypt issuance became more widespread, that the number of deleted names begins to trend upwards at a greater rate. This is likely due to the fact that Let's Encrypt certificates are issued for only 90 days at a time, and the increased frequency in certificate expiration causes names to be purged from the active set at a greater rate.

Despite the increase in the number of names (and hence in the size of the signaling set) over time, we argued in Section 7 that this increase would be outweighed by the sublinear growth of the signaling set size and falling cost of storage and memory. We can see the trend in the signaling set size especially well in Fig. 10, a graphical representation of Table 2.

B DETAILED PERFORMANCE MEASUREMENTS

In Section 6.4, we showed that the connection latency overhead in using CAPS is approximately 5%. Figs. 11, 12, and 13 show this overhead given the number of policy proofs request, the number

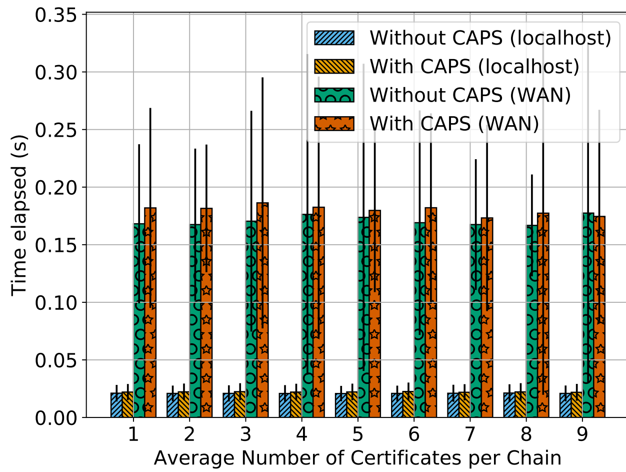

Figure 12: Handshake latency versus the number of certificates per chain sent by the domain. Error bars represent standard error.

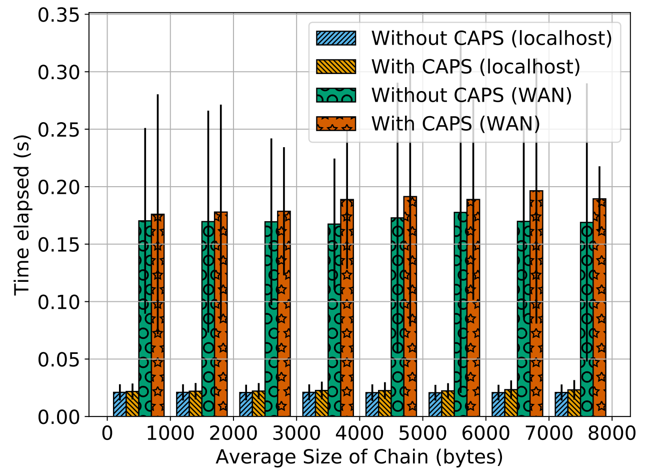

Figure 13: Handshake latency versus the average certificate chain size sent by the domain. Error bars represent standard error.

of certificates per chain, and the average size of each certificate chain, respectively. In almost every case, the variance in network latency resulted in error bars that far exceed the difference in latency between using the existing Web PKI and using CAPS.

Given the way we structured the messages in our CAPS TLS extension, the extra data sent in the CAPS handshake is directly dependent on the size and number of certificate chains, as well as the number of proofs sent. In particular, the extra data sent from client to server is 1 byte, and the extra data sent from server to client is $(2 + (292 \times \#\text{proofs}) + (\sum_{\text{chain}} \texttt{sizeof}(\text{chain})))$ bytes.