Hammurabi: A Framework for Pluggable, Logic-Based X.509 Certificate Validation Policies

James Larisch
Harvard University

Waqar Aqeel
Duke University

Michael Lum
University of Maryland

Yaelle Goldschlag
University of Maryland

Kasra Torshizi
University of Maryland

Leah Kannan
University of Maryland

Yujie Wang
University of Maryland

Taejoong Chung
Virginia Tech

Dave Levin
University of Maryland

Bruce M. Maggs
Duke University and Emerald Innovations

Alan Mislove
Northeastern University

Bryan Parno
Carnegie Mellon University

Christo Wilson
Northeastern University

ABSTRACT

This paper proposes using a logic programming language to disentangle X.509 certificate validation *policy* from *mechanism*. Expressing validation policies in a logic programming language provides multiple benefits. First, policy and mechanism can be more independently written, augmented, and analyzed compared to the current practice of interweaving them within a C or C++ implementation. Once written, these policies can be easily shared and modified for use in different TLS clients. Further, logic programming allows us to determine when clients differ in their policies and use the power of imputation to automatically generate interesting certificates, e.g., a certificate that will be accepted by one browser but not by another.

We present a new framework called Hammurabi for expressing validation policies, and we demonstrate that we can express the complex policies of the Google Chrome and Mozilla Firefox web browsers in this framework. We confirm the fidelity of the Hammurabi policies by comparing the validation decisions they make with those made by the browsers themselves on over ten million certificate chains derived from Certificate Transparency logs, as well as 100K synthetic chains. We also use imputation to discover nine validation differences between the two browsers' policies. Finally, we demonstrate the feasibility of integrating Hammurabi into Firefox and the Go language in less than 100 lines of code each.

CCS CONCEPTS

• **Security and privacy → Web protocol security**; *Logic and verification.*

KEYWORDS

TLS, Web PKI, X.509 certificate validation, logic programming

ACM Reference Format:
James Larisch, Waqar Aqeel, Michael Lum, Yaelle Goldschlag, Kasra Torshizi, Leah Kannan, Yujie Wang, Taejoong Chung, Dave Levin, Bruce M. Maggs, Alan Mislove, Bryan Parno, and Christo Wilson. 2022. Hammurabi: A Framework for Pluggable, Logic-Based X.509 Certificate Validation Policies. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3548606.3560594

CCS '22, November 7–11, 2022, Los Angeles, CA, USA

ACM ISBN 978-1-4503-9450-5/22/11…$15.00
https://doi.org/10.1145/3548606.3560594

1 INTRODUCTION

The Transport Layer Security (TLS) Public-Key Infrastructure (PKI) is critical for establishing authenticated communication between entities on the Internet. All manner of servers (and many clients) authenticate themselves during the TLS handshake by sending a public key housed in an X.509 certificate signed by a trusted Certificate Authority (CA). TLS user-agents[1] verify that the received certificates are valid according to various RFCs [39, 103] and, for browsers, CA/B Baseline Requirements (BRs) [28] that prescribe or suggest, among other things, validating cryptographic signatures, checking extensions, and checking revocation status.

However, even when two user-agents are standards compliant, they may still make different *policy* decisions about what constitutes a valid certificate. For instance, while the CA/B BRs do not require browsers to implement revocation checks, Firefox performs OCSP revocation checking for all certificates while Chrome only uses OCSP for EV certificates [81]. Furthermore, there are often defensible policy reasons (e.g., backwards-compatibility) for violating standards. For example, Firefox allows leaf certificates that include the `keyCertSign` key usage (§7.3), despite RFC 5280 forbidding it.

[1]We define "user-agent" as software responsible for validating TLS certificates, e.g., web browsers or TLS libraries like OpenSSL.

1.1 The Problem of Entanglement

Unfortunately, today's user-agents currently entangle certificate validation *policy*—the high-level rules by which a certificate is deemed valid or authorized to communicate on behalf of an entity—with validation *mechanism*—the low-level implementation code responsible for parsing certificates, verifying cryptographic signatures, and executing policy rules. The separation of mechanism and policy is a core security design principle of many research and software systems [21, 44, 74, 79, 110, 112, 121, 123, 124], and its absence in the TLS PKI has particularly problematic consequences.

First, it is difficult to determine *what* policy a user-agent implements, since doing so often requires examining the source code of a large and complex implementation where validation policy and implementation details are intertwined. This has obvious negative consequences for analyzing the intended validation behavior. For example, the only way to check that Firefox obeys Mozilla's Root Store Policy [90], which describes Firefox's validation criteria in English, is to manually examine Firefox's C++ source code.

Second, entanglement between mechanism and policy makes it difficult for new TLS libraries to *reuse* existing—well-developed but complex—validation policies. As a result, they must implement their own policy decisions about when to accept a certificate based on the sometimes ambiguous [125] standards, which can lead to security-critical, validation-related bugs [3–11, 15, 56]. Furthermore, software depending on TLS libraries inherit their entangled policy decisions, which are difficult to understand or change.

Finally, entanglement makes it difficult to determine *how* and *why* two user-agents differ in the rules they use for validation. For instance, the best way to determine whether a certificate is valid in Chrome versus Firefox is to visit a website serving that certificate in both browsers and compare the results (although even this may not provide complete certainty, as the results may differ based on the OS used [81]). If the browsers return different results, is the difference due to differing interpretations of an RFC [125], due to an unintentional bug, or due to an intentional policy choice?

1.2 Disentanglement with Hammurabi

In this work, we present *Hammurabi*, a framework for executing "pluggable" certificate validation policies expressed in a standardized, high-level, logic-programming language. Hammurabi separates the mechanisms required to process X.509 certificates (e.g., parsing, signature validation, performing OCSP and CRL requests, etc.) from the validation policy (e.g., enforcing a maximum certificate lifetime, blocking particular CAs, enforcing name constraints, determining which revocation checking mechanisms to use, etc.).

Hammurabi replaces the X.509 certificate validation code found in TLS user-agents. Hammurabi-compatible user-agents pass certificates (a leaf and intermediates), their choice of high-level validation policy, and other parameters to the Hammurabi *engine*, which executes the policy over the given certificates and returns the validation result. We present a feature-complete prototype of Hammurabi implemented as a system service (§6), but it could also be implemented as a library, as we describe in §9.1.

The disentanglement of policy and mechanism enabled by Hammurabi generates several benefits. First, it facilitates separation of required expertise. TLS user-agents often have legitimate reasons for deviating from standard validation policy requirements, as we discuss in §2, and validation policy changes impact both end-users and CAs. Using Hammurabi, policy experts (e.g., the managers of root-store programs [34, 89]) can write, extend, and analyze the policies of multiple user-agents. Likewise, implementation engineers need only add the plumbing required to support certificate validation, rather than interweaving the high-level validation policy with low-level mechanisms.

Second, Hammurabi makes it easier to *reuse* validation policies. For instance, Chrome and Firefox share much of their validation logic, which could be standardized and reused, as we show in §7.1. Reusable policy components may lead to more consistency across the vast TLS ecosystem (e.g., web browsers, command-line tools, and IoT devices). Hammurabi also makes it easier to *extend* existing TLS validation policies: for example, a new TLS user-agent can import (and modify) Chrome or Firefox's existing policy, rather than writing its own, which often leads to omissions and bugs. Additionally, user-agents can adopt sophisticated validation policies with little TLS-specific implementation work.

Third, Hammurabi improves TLS policy agility for user-agents. Changing policy today typically means modifying low-level code or switching TLS libraries entirely, both of which are complex tasks that require recompilation and redistribution. With Hammurabi, applications like `curl` could provide runtime interfaces for developers to choose validation policies as they see fit, e.g., to comply with new CA/B BRs or RFCs, or in response to security incidents like the deprecation of a CA.

Finally, by leveraging techniques available to high-level languages (e.g., *imputation* using our chosen language, Prolog), developers and testers can automatically generate example certificates that are valid according to one policy but not another, making it easier to automatically discover *how* two policies differ. For example, although Chrome and Firefox share validation logic, they disagree in subtle and complex ways, as we show in §7.3. These differences, once highlighted automatically, can then be further classified as validation bugs or differences of institutional opinion.

1.3 Contributions and Roadmap

This paper makes the following major contributions:

- We separate TLS certificate validation mechanism and policy in the design of Hammurabi, a drop-in replacement for the validation logic of existing user-agents. The Hammurabi *engine* (§4) performs the validation mechanisms (e.g., parsing, signature checking, and revocation checking) before executing Hammurabi policies (§5) expressed in a logic-programming language.
- We implement and open source[2] the validation logic of Mozilla Firefox and Google Chrome in our high-level policy language (Prolog, §5) and verify that, using our Hammurabi engine prototype (§6), our policies make the same validation decisions as the corresponding browsers for 10M certificates sampled from nine Certificate Transparency logs and for 100K fuzzed certificates (§7.1).

[2] https://github.com/semaj/hammurabi

- We leverage our use of logic programs to automatically discover differences between our two Hammurabi policies (and thus the corresponding browsers) using *imputation*. We find and discuss nine subtle differences between Firefox and Chrome in the context of the RFCs and CA/B BRs (§7.3).

We begin in §2 by discussing how certificate validation policy and mechanism are entangled in modern TLS user-agents and why this is a problem. We then briefly discuss our threat model in §3, before presenting the design of the Hammurabi engine in §4 and the choice of Prolog as a policy language in §5. We describe the preliminary Hammurabi prototype implementation in §6 and evaluate it and our Hammurabi policies in §7. In §8 and §9 we discuss related work, deployment options, and our future outlook.

2 BACKGROUND

In this work we focus on the *Web PKI*, whose goal is to authenticate websites to web clients (e.g., web browsers, mobile apps, and command line tools), which enables confidentiality and integrity for data sent over an untrusted network (e.g., the Internet) [99]. The Web PKI focuses on X.509 certificates, which are signed attestations that bind a subject (i.e., a domain or CA name) to a public key.

Over the nearly 30 years since its inception [66], X.509 has evolved to meet new threats and deployment considerations. In 1996, RFC 5280 [39] standardized X.509 version 3, which introduced certificate *extensions*. Each extension consists of a unique object identifier (OID), a boolean `Critical`, and arbitrary data specific to that extension. Most changes to certificates have been made via new extensions, such as Subject Alternative Names (SAN) and Certificate Revocation List (CRL) distribution points [39] in 1996, certificate transparency log timestamps (SCT) [77] in 2013, and Online Certificate Status Protocol (OCSP) Must-Staple [59] in 2015.

Certificates used by websites are referred to as leaf certificates.[3] TLS user-agents accept leaf certificates that are issued and signed by CAs, whose certificates are in turn signed by other CAs, terminating with a self-signed CA-owned root certificate.[4] The certificates in-between the leaf and root are referred to as intermediate certificates. Typically, the server sends the leaf and intermediate certificates to the user-agent during the TLS handshake.

2.1 Path Building

To be more specific, a user-agent accepts a leaf certificate if and only if the user-agent can build a "valid path" from it to a trusted, self-signed CA root certificate. For every certificate in a valid path excluding the self-signed root: (1) the certificate's Issuer Distinguished Name (DN) matches its parent's Subject DN, (2) the certificate's `authorityKeyIdentifier` field matches its parent's `subjectKeyIdentifier` field, and (3) the certificate is signed by the parent's private key (corresponding to the parent's public key).

There may be multiple paths for a given leaf certificate [82] because there may be multiple intermediate certificates with the same Subject DN and public keys, each signed by different CA certificates (i.e., cross-signing). As a result, the path building algorithm is a depth-first search (DFS) problem in which the user-agent attempts to build and validate *any* path from the leaf to a root, abandoning invalid paths along the way. Correctly implementing this logic is subtle, and failure to properly fall back to alternative paths led to an outage in May of 2020 [108].

2.2 Certificate Validation Ambiguity

Unfortunately, the list of criteria for what constitutes a valid path described in §2.1 is incomplete. Indeed, **there is no complete, universal, canonical definition of what constitutes a valid certificate path**. For instance, web browsers only construct an edge between a child and parent certificate if the parent issued the child, the public keys match, the signature is valid, *and* the browser's own criteria (based on RFCs, BRs, and unique decisions) are met. Such criteria may include disallowing particular signature algorithms, disallowing particular extensions, and more.

Path validation requirements are specified across many different and occasionally ambiguous RFCs [45, 125], including but not limited to RFCs 3647, 5280, 6066, 6125, and 6960. For example, RFC 5280 states that the `signatureAlgorithm` field of the certificate and the `signature` field of the inner `tbsCertificate`[5] must be "the same", but does not specify what "the same" means. One user-agent may require that the encodings must be "byte-for-byte" equivalent, while another may require equivalency after being parsed (see §10.1).

Adding to the complexity, web browsers—arguably the most ubiquitous TLS user-agents deployed today—and CAs agree upon their own set of 123 requirements for TLS certificates as part of the Certificate Authority/Browser (CA/B) Forum. These requirements, dubbed the CA/B Baseline Requirements (BRs) [28], both augment and *supersede* (when in contradiction to) the RFC requirements.

Nonetheless, many user-agents ignore RFC and BR requirements due to lack of resources, performance concerns, business-driven decisions, or mistakes. For instance, the standard HTTP libraries in Python performed virtually no certificate validation until 2014 [55].

Additional discrepancies arise because some user-agents implement RFC or BR recommendations (e.g., "MAY" and "MAY NOT"), while others do not. For example, Firefox implements revocation-checking for almost all leaf certificates using OCSP, OCSP Stapling, OCSP-Must-Staple, and/or more recently CRLite [76]. Chrome, on the other hand, only supports CRLSet, OCSP Stapling, and leaf OCSP checking for EV certificates, and has no plans to implement support for OCSP-Must-Staple [13, 14].

We present detailed descriptions of nine areas where Chrome and Firefox's certificate validation policies deviate from one other based on our imputation-driven analysis (§7.3).

2.3 Other Challenges

Library Design. Georgiev et al. found that existing TLS libraries make it easy for applications to perform validation incorrectly, due to poor API design [56]. Fahl et al. showed that significant misuse of the TLS APIs provided to Android applications (via Java) resulted in potential MITM attacks for 8% of examined applications [53], among other issues with TLS on Android [54, 119]. Similar issues have been documented in the IoT ecosystem, where devices sometimes fail to properly validate certificates [93].

[3] Standards often refer to these certificates as "end-entity" certificates.
[4] Technically the path terminates with a trust anchor [39], whose information is often distributed as a self-signed certificate.
[5] The signature field redundancy was meant to defend against a now-obsolete attack [58].

Implementation. Extensive research examines certificate validation code for bugs and omissions. Brubaker et al. [27] used a fuzzing technique called "Frankencerts" to determine which TLS libraries would correctly (or incorrectly) reject certificates with randomly generated fields and extensions. Fuzzing has also been used to identify TLS-protocol level bugs in libraries [109]. Chau et al. [32] used a combination of symbolic and concrete execution of TLS validation code to find discrepancies across libraries. CERES [45] is an RFC-based, formally specified reference implementation designed for the differential testing of existing non-browser TLS libraries.

2.4 Summary

While prior work has found bugs and inconsistencies in existing certificate validation implementations, it fails to address a fundamental issue: certificate validation *mechanism* and *policy* are two separate components, and should be treated as such. This entanglement makes it difficult to discover and reason about validation bugs and errors. Over-inclusive validation logic within an application may not be a *mechanism* bug at all, but rather an insecure *policy* choice. Complicating the issue further, there is no "one, true" validation policy—there are multiple, sometimes ambiguous RFCs with optional requirements, additional requirements for certain user-agents such as web browsers, and good reasons to break the rules from time to time (e.g., for backward compatibility, or to voluntarily choose a more secure policy than what the standards prescribe).

In other words, even if on-going efforts to produce a bug-free TLS library [25] and formally specified certificate parsing routines [45, 96] are successful, a key question remains: *which certificate validation policy should such libraries execute?*

3 THREAT MODEL

Our goal is to replace the certificate validation code used by existing TLS user-agents with a system that provides identical external functionality but that properly factors validation policy from validation mechanism. Since our system runs on the client, it inherits the threats that target existing TLS user-agent code. In particular, we assume that adversaries may send arbitrary data to the client, including bogus certificates.

Since our focus is on certificate validation, we assume that the X.509 parsing routines, the cryptographic libraries, and the TLS protocol itself are free from bugs and implemented correctly. Other work focuses on improving these components [25, 45, 96], but this is not a goal of Hammurabi. We also assume that the user-agent's certificate validation policy (detailed in §4 and §5) is legitimate and has not been tampered with.

4 THE HAMMURABI ENGINE

Our system, *Hammurabi*, decouples certificate processing *mechanism* from certificate validation *policy*. In Hammurabi, validation mechanisms (e.g., parsing the fields of an X.509 certificate, parsing an OCSP response, etc.) produce information that we call *facts*. Validation policies use these facts to make validity determinations about certificates. We call these decision procedures policy *rules*. Hammurabi policies are composed of one or more rules. Crucially, user-agents specify the validation policy that Hammurabi must use when validating a given leaf certificate.

Figure 1 presents the conceptual design of the Hammurabi engine. User-agents delegate leaf certificate validation to the engine subject to a given validation policy, and the engine returns a decision (valid or invalid) to the user-agent.

Guided by Figure 1, in the following sections we describe the operation of Hammurabi in detail, starting with how user-agents interface with Hammurabi, critical initial policy choices, and validation mechanisms like X.509 parsing and cryptographic signature validation. In §5 we discuss Hammurabi's policy language in detail. We present details of our prototype implementation in §6.

4.1 Interface and Initial Policy Choices

Starting from the left of Figure 1, the Hammurabi engine defines an interface for interacting with user-agents. When invoking the engine, user-agents are required to supply: (1) the subject name in question (e.g., a DNS name), (2) the leaf certificate (and any other certificates provided during the TLS handshake), and (3) a validation policy written in Hammurabi's policy language. User-agents may also pass optional arguments that further refine their desired validation policy, as we describe below.

Because the engine must validate paths from a given leaf certificate to root certificates, it must be told which roots to trust. Many existing user-agents use the OS's certificate root store, while major browsers often use their own root store [37, 90]. Hammurabi offers three ways for user-agents to choose their root store:

(1) Validation policies may include a directive specifying which root store to use.
(2) A user-agent may override the root store directive specified in the validation policy by supplying its own root store as an argument.
(3) If (1) and (2) above are left unspecified, Hammurabi will use the OS's root store as a fallback.

Similarly, existing user-agents differ in terms of which certificate revocation checking protocols they support [43, 81]. Revocation decisions must be made before policies are executed because policies may rely on revocation information (e.g., the result of an OCSP request), and the policies themselves cannot perform I/O. As a result, policies may include directives specifying which revocation checking protocols the engine should use, e.g., CRL, OCSP, OCSP stapling, etc. Additionally, user-agents may override a policy's revocation directives by passing in optional arguments.

4.2 Validation Mechanisms

Given the root store and revocation policies, the Hammurabi engine executes the validation mechanisms that ultimately produce facts for the validation policy. The engine includes two major classes of mechanisms: certificate tree construction and revocation checking.

4.2.1 Certificate Tree Construction. The engine performs the initial depth-first-search path building and cryptographic signature validation process (§2.1, §2.2). The engine starts at the leaf certificate supplied by the user-agent and constructs *all possible* paths to a trusted root by *only* checking the signature, public keys, and issuer match of potential child-parent pairs. It discards all certificates that do not belong to such a path, backtracking as required. During path building the engine may check intermediate certificates supplied by

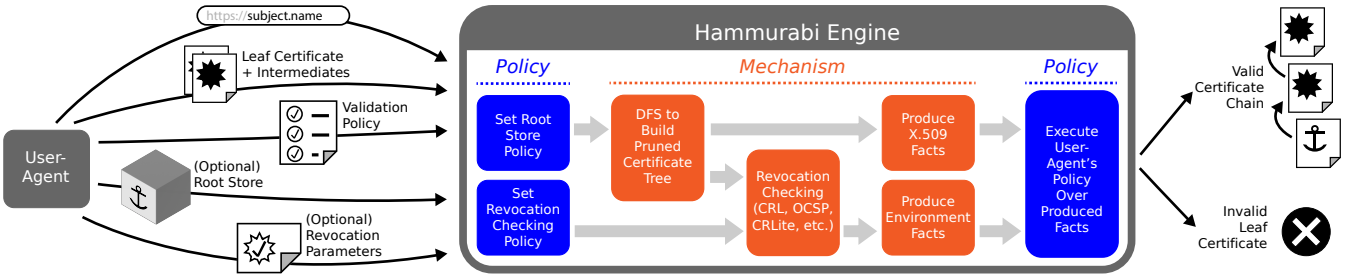

Figure 1: Overview of how a user-agent uses Hammurabi to validate a certificate. Starting at the left, the user-agent supplies arguments to Hammurabi: a subject name, leaf and intermediate certificates, a validation policy, (optionally) a root store, and (optionally) revocation checking parameters. Hammurabi picks root store and revocation checking policies based on the supplied validation policy and any optional arguments. Next, it builds a candidate tree of certificates by performing a DFS starting at the leaf certificate, traversing edges to intermediate and root certificates, and verifying cryptographic signatures along the way. It then performs revocation checking of certificates in the candidate tree. Next, it translates the candidate certificates and revocation information into Prolog facts, gathers additional environmental facts, and passes them all as input to the user-agent's policy. Finally, Hammurabi executes the policy and returns the result to the user-agent: either a valid chain of certificates from the supplied leaf to one or more trusted roots, or a message indicating that the certificate failed validation.

the user-agent, and it may fetch new intermediates by examining the Authority Information Access (AIA) field of certificates.

The end result of path building is a tree of certificates that are cryptographically valid. However, at this point, no other certificate validity checks are applied. Rather, the engine forwards this "partially-valid" certificate tree to the validation policy supplied by the user-agent. As we show in Figure 1 and discuss in §5, the validation policy supplied by the user-agent performs the full path validation, e.g., checking the certificate lifetimes, signature algorithms, blocklist checking, and more.

4.2.2 Revocation Checking. The engine is responsible for retrieving the revocation information decided upon in the first stage for all certificates in the tree produced during path building. Based on a user-agent's policy choices (see §4.1), the engine will attempt to retrieve revocation information from various sources such as CRL, OCSP, or OCSP responses that were stapled to certificates supplied by the user-agent. Advanced versions of the engine may also incorporate additional sources of revocation information such as OneCRL [57], CRLSet [12], and CRLite [76]. The engine implements all necessary network protocols and cryptographic signature checks necessary to retrieve and verify revocation information.

4.2.3 Fact Production. The engine takes the results of certificate tree construction and revocation checking and translates them into facts for the validation policy. It first parses all relevant X.509 fields and extensions from certificates in the tree and converts them into a policy-compatible format. We call these *X.509 facts*, and they must be encoded in a format specified by Hammurabi.

The engine also produces *environment facts*, i.e., information about the validation context—such as the current time and subject name supplied by the user-agent—and passes them to the validation policy. These environment facts include all fields parsed from revocation information (e.g., OCSP responses). Similar to X.509 facts, these facts must be encoded in Hammurabi format.

We discuss both X.509 and environment facts in §5.3.

X.509 Facts
`fingerprint(Cert, Fingerprint)`
`commonName(Cert, CommonName)`
`subjectAlternativeName(Cert, SAN)`
`notBefore(Cert, NotBefore)`
`notAfter(Cert, NotAfter)`
`maxPathLength(Length)`
`signatureAlgorithm(Cert, Algo)`
`extensionExists(Cert, Extension, Exists)`
`signsAndIssuer(Cert, Parent)`

Table 1: A non-exhaustive list of *X.509 Facts*—Prolog facts containing existing X.509 data for the concrete certificate `Cert`. Capitalized terms are placeholders for actual values. A certificate may have multiple instances of a fact, e.g., `subjectAlternativeName`.

5 HAMMURABI POLICIES

The Hammurabi engine's final step is executing a validation policy. In Hammurabi, policies are implemented as logic programs: they contain a set of *rules* that operate over engine-produced *facts*. For example, a policy author might specify the rule "if the certificate lifetime is greater than one year and the SAN list has more than 12 entries, the certificate is invalid". The Hammurabi policy interpreter evaluates these rules over the engine-produced facts such as "the certificate lifetime is 370 days" to determine whether a particular *query* ("is the certificate invalid?") can be derived from the given facts and rules. The interpreter derives new facts until either the query is satisfied or it reaches a fixed point.

In this section, we describe the responsibilities of Hammurabi validation policies and how these responsibilities can be expressed as rules in a logic programming language, and provide more details about the facts that the rules operate over. For the Hammurabi prototype we chose Prolog as the language for expressing validation policies; we justify this choice in §5.2.

Environment Facts	Description
`hostname(Hostname)`	The DNS name being accessed by the user in the user-agent.
`now(Timestamp)`	The current system time in seconds since UNIX epoch.
`serverIPAddr(Addr)`	The current certificate chain was received from a server with IPv4 or IPv6 address `Addr`.
`tlsVersion(Ver)`	Connection with the remote server was established using TLS version `Ver`.
`sctReceivedTLS(Cert)`	Valid SCT for `Cert` received via TLS extension `signed_certificate_timestamp`.
`sctReceivedOCSP(Cert)`	Valid SCT for `Cert` received via OCSP extension OID 1.3.6.1.4.1.11129.2.4.5.
`publicSuffix(Suffix)`	`Suffix` is a public suffix such as `"co.uk"`. There is one such fact for each public suffix.
Revocation Facts	**Description**
`ocspResponse(Cert, [IsValid, IsSigned, IsExpired, Status])`	Indicates whether the OCSP response is valid, signed, and expired, plus the revocation status.
`stapledResponse(Cert, [IsValid, IsSigned, IsExpired, Status])`	Same as above for stapled OCSP responses.
`crl(Cert, Issuer, [IsValid, IsSigned, IsExpired, Status])`	Indicates that a certificate appears on CRL signed by `Issuer`.
`crlite(Cert)`	`Cert`'s presence in CRLite [76].
`crlSet(Cert)`	`Cert`'s presence in CRLSet [12].
`oneCRL(Cert)`	`Cert`'s presence in OneCRL [57].

Table 2: Environment facts made available to Hammurabi policies.

5.1 Policy Responsibilities

The engine extracts necessary syntactical information from sources like certificate fields and revocation status; policies make decisions based on this information. We include a non-exhaustive list of ways in which user-agent policies (in particular, those of browsers) may diverge from standards and one another:

Extension and field processing. X.509 extensions and fields must be processed according to the RFCs and BRs, but browsers may diverge from these standards (e.g., for backwards compatibility).

Time checking. All certificates in the candidate chain must have valid issuance and expiration dates (relative to the current time), and their lifetime must be within a certain range (which varies on a per-browser basis).

Name matching. The leaf certificate must be authorized to speak on behalf of the subject (e.g., DNS name) in question. The `commonName` field used to specify this information, but was deprecated in favor of the `subjectAltName` extension. Browsers may differ in how they handle this deprecation.

Name constraints. The name constraints extension [39] was implemented relatively recently [65] and is used by certificate authorities to restrict the names (among other things) that descendant certificates may be used for. Browsers have varying degrees of support for name constraints.

Revocation decisions. The BRs do not require user-agents to check for revocation, but some browsers prioritize it.

Blocklist consultation. Certain certificates, such as those issued by Symantec roots, are blocked by default by most modern browsers [30, 48, 86]. Browsers may differ, however, on exactly which Symantec certificates to block, when to block them, and whether exceptions should be made for certain child intermediates.

SCTs. Chrome rejects certificates that do not include Signed Certificate Timestamp attestations from public Certificate Transparency logs that the certificate was issued and recorded in one of these logs—this helps detect certificate misissuance [31]. Firefox does not currently require leaves to include SCTs.

5.2 Policies Are Logic Programs

We adopt Prolog in our prototype implementation as the language for expressing Hammurabi policies for several reasons:

- **Expressive power.** Prolog is sufficiently powerful to express a wide range of certificate validation constraints, as we demonstrate in §7.1. Logic-programming languages such as Prolog and its syntactic subset Datalog [16, 117] have been used extensively for expressing security policies [17, 52, 67, 72, 92, 118].
- **Declarative style.** Prolog code mirrors the declarative nature of certificate validation (constraints over "facts"). In contrast, the (often C or C++) source code of existing TLS clients and libraries obfuscates the policies they implement. Policies cannot use non-declarative operators such as `cut`.
- **Imputation.** Specifying certificate validation policies in Prolog enables us to programmatically compare different policies using Prolog's unification algorithm. As we show in §7.3, this allows us to automatically identify differences in real-world certificate validation policies.

5.3 Facts

As shown in Figure 1 and introduced in §4.2.3, the Hammurabi engine passes two types of facts to policies: *X.509 facts* and *environment facts*. X.509 facts are simply the fields of every X.509 certificate included in the candidate tree, encoded as Prolog facts. For instance, `notBefore(root45, 1650901161)` would be the `notBefore` time (seconds since Epoch) of the 45^{th} root certificate in the root store encoded as a Prolog fact. The X.509 facts also include signature information produced by the engine. For instance, `signsAndIssuer(root45, cert1)` indicates that the 45^{th} root signed and issued `cert1` (and the signature was valid). Table 1 provides a subset of X.509 facts.

Hammurabi policies may also need to reference information that is not found in certificates but is still relevant for validation, such as the current time, revocation information, the subject DNS name, or the TLS server's IP address. These are called *environment facts*. For instance, the current time (seconds since Epoch) may be encoded as `now(1650901161)`. Table 2 lists the environment facts produced by the engine and made available to policies. Policies may only rely on environment facts specified by Hammurabi.

5.4 Imputation

With policies specified as logic programs, policy authors can, when given a partially specified certificate, *impute* the missing values that

would make the certificate satisfy a particular set of constraints. As a consequence, Hammurabi can automatically examine the differences between two validation policies, e.g., by generating a certificate that one policy accepts while the other rejects.

To illustrate imputation, consider the simple rule `foo` that is true if the Key Usage extension is `keyCertSign` and the certificate's `notBefore` time is before its `notAfter` time:

```
foo(KeyUsage, NotBefore, NotAfter) :-
  isKeyUsage(KeyUsage),
  KeyUsage == "keyCertSign",
  NotBefore < NotAfter.
```

If we query Prolog with concrete values for `NotBefore` and `NotAfter`, it will impute a concrete value for `KeyUsage` that makes the rule hold (or report that it cannot find such a value) using unification [70]. Critically though, the values must be drawn from an enumerable domain—Prolog cannot enumerate all possible strings, hence the `isKeyUsage` clause, which ensures that `KeyUsage` is from a finite set of valid key usage strings. Despite `NotBefore` and `NotAfter` being integers, we can also impute their values (by providing a concrete value for `KeyUsage`) using the Constraint Logic Programming over Finite Domains (CLPFD) library [116].

Although most X.509 and environment fact values are easily enumerable—booleans, enumerable strings, or OIDs—some policy-critical values are relatively unstructured strings, such as DNS names, distinguished names, and email addresses. DNS names, in particular, will be used by almost every policy, since the given subject name must match the Common Name or a Subject Alternative Name of the leaf certificate.

Because Prolog can not search all possible strings, we must restrict each set of strings to an enumerable set. Consider a DNS name, which one can think of as a list of labels separated by dots. Often the actual concrete value of each label is irrelevant from the policy's perspective—instead, we know that policies usually check whether the labels of two DNS names match (with special consideration taken for the wildcard * character). Using this insight, we restrict all DNS names (subject name, Common Name, SAN) to the following `DNSName` structure:

```
DNSName = Prefix + "." + Suffix
Suffix = "com" | "co.uk" | "org" | ...
Prefix = Wildcard | Wildcard + "." + Constants | Constants
Constants = Constant | Constant + "." + Constants
Constant = "a" | "b" |"c" | "d" | "e"
```

The values for `Suffix` are drawn from the Public Suffix List [2] and we limit the total length of DNS names to seven labels.[6] Because this set of strings is enumerable (and relatively small), Prolog can impute operands of "type" `DNSName`. Note that this approach is a heuristic, it is not complete: if a policy depends on a particular label's concrete value, this approach will not work. However, this approach adds enough power to allow imputation to discover DNS name-related differences in the Firefox and Chrome policies (§7.3).

This approach can be extended to other relatively unstructured string values. Further, more sophisticated imputation may be possible using a constraint solver that includes a theory of strings such as Z3 [87] or using a logic programming language integrated with a constraint solver like Formulog [23]. We look forward to exploring such options in future work.

[6] The choice of seven labels is not fundamental, but represents a reasonable tradeoff between coverage and imputation time. This limit may be increased later if desired, but seven labels covers the vast majority (>99.5%) of domain names [68].

5.5 Example Policy: Mozilla Firefox

As we discuss in §7, we manually extracted the certificate validation logic from the Mozilla Firefox C++ code and converted it into a Hammurabi policy. We show an abridged slice of this policy in Listing 1.

The policy is split into four major rules: `validChain`, `validLeaf`, `validNonLeaf`, and `validRoot`. The `validChain` rule is the entry-point and its definition is required by Hammurabi. In the Firefox policy, it first extracts properties of interest from the leaf (62–70); note that `x509` and `env` denote X.509 and environment facts, respectively. The `validChain` rule then checks that the leaf is valid (71) before checking that there exists an issuer of `Leaf` that is a valid non-leaf certificate (75).

The `validNonLeaf` rule has two critical branches for validating the non-leaf as an intermediate (32–39) and for validating it as a root (41–44).[7] The `validNonLeaf` rule is called recursively until a root is reached. The potential existence of multiple paths need not be expressed in Prolog, since rules such as `x509:signsAndIssuer` hold if there *exist* operands that satisfy the relation.

We also extracted and converted Chrome's certificate validation logic into a Hammurabi policy (§7). The high-level structure of the Chrome policy is exactly the same as the Firefox policy shown in Listing 1—the differences lie in the semantics of some of the component rules whose definitions are not shown here. For example, both the Chrome and Firefox policies define `isTimeValid` policies, but their semantics differ (Firefox allows certificates with longer lifetimes than Chrome).

6 IMPLEMENTATION

We implemented a feature-complete Hammurabi prototype that we use for testing and evaluation. Our prototype runs as a daemon and existing user-agents interact with it using Inter-process Communication (IPC). It is written in roughly 2.5K lines of Rust. Our prototype implements the following functionality.

Certificate parsing. Our prototype uses the Rust X.509 and DER parsing libraries from the Rusticata project [100, 101] to parse each certificate into a Rust object. Syntactic validation related to ASN.1 DER encoding and X.509 requirements happens at this stage. For example, `notBefore` and `notAfter` are syntactically required to be in either the `UTCTime` or `GeneralizedTime` ASN.1 types. If these fields are not correctly encoded then validation is terminated.

Cryptographic validation. We use the Rust OpenSSL bindings for certificate signature validation. As described in §4, our prototype only checks signatures for potentially valid paths where, for each child–parent pair, the parent issued the child.

X.509 fact production. Our prototype converts each X.509 field into a corresponding Prolog X.509 fact.

Environment fact production. Our prototype produces environment facts such as the current time, subject DNS name, and public suffix list. It also performs any necessary revocation checking steps

[7] The ; operator denotes disjunction while , denotes conjunction.

```
validRoot(LeafSANList, Fingerprint, Lower, Upper,
          BasicConstraints, KeyUsage, ChildFingerprint) :-
  isCA(BasicConstraints),
  keyUsageValidRoot(KeyUsage),
  isTimeValid(Lower, Upper),
  symantecValid(Fingerprint),
  trustedRoot(Fingerprint).

validIntermediate(Fingerprint, Lower, Upper, Algorithm,
                  BasicConstraints, KeyUsage, ExtKeyUsage,
                  LeafEVStatus, StapledResponse, OcspResponse) :-
  extKeyUsageValid(BasicConstraints, ExtKeyUsage),
  isCA(BasicConstraints),
  isTimeValid(Lower, Upper),
  keyUsageValid(BasicConstraints, KeyUsage),
  strongSignature(Algorithm),
  \+env:oneCRL(Fingerprint),
  notOCSPRevoked(Lower, Upper, LeafEVStatus,
                 StapledResponse, OcspResponse).

validNonLeaf(Cert, CertsSoFar, Leaf) :-
  x509:commonName(Leaf, LeafCommonName), x509:sanList(Leaf, LeafSANList),
  x509:extKeyUsage(Cert, KeyUsage), x509:keyUsage(Cert, KeyUsage),
  x509:fingerprint(Cert, Fingerprint),
  x509:notBefore(Cert, Lower), x509:notAfter(Cert, Upper),
  x509:signature(Cert, Algorithm, Params),
  env:stapledResponse(Cert, StapledResponse),
  env:ocspResponse(Cert, OcspResponse),
  getEVStatus(Leaf, LeafEVStatus),
  getBasicConstraints(Cert, BasicConstraints),
  ((
    validIntermediate(Fingerprint, Lower, Upper, Algorithm,
                      BasicConstraints, KeyUsage, ExtKeyUsage,
                      LeafEVStatus, StapledResponse, OcspResponse),
    x509:signsAndIssuer(Cert, Parent),
    Cert \= Parent,
    validNonLeaf(Parent, LeafCommonName, LeafSANList,
                 LeafEVStatus, CertsSoFar + 1, Leaf),
    nameConstrained(Cert, Leaf)
  );(
    x509:signsAndIssuer(Child, Cert),
    x509:fingerprint(Child, ChildFingerprint),
    validroot(LeafSANList, Fingerprint, Lower, Upper, BasicConstraints,
              KeyUsage, ChildFingerprint)
  )).

validLeaf(Fingerprint, SANList, CommonName, Lower, Upper,
          Algorithm, BasicConstraints, KeyUsage, ExtKeyUsage,
          EVStatus, StapledResponse, OcspResponse) :-
  extKeyUsageValid(BasicConstraints, ExtKeyUsage),
  \+isCA(BasicConstraints),
  isTimeValid(Lower, Upper),
  keyUsageValidLeaf(KeyUsage),
  leafDurationValid(EVStatus, Lower, Upper),
  nameValid(SANList, CommonName),
  nameMatchesHost(SANList, CommonName),
  strongSignature(Algorithm),
  \+env:oneCRL(Fingerprint),
  notOCSPRevoked(Lower, Upper, EVStatus, StapledResponse, OcspResponse).

validChain(Leaf) :-
  x509:commonName(Leaf, CommonName), x509:sanList(Leaf, SANList),
  x509:extendedKeyUsage(Leaf, ExtKeyUsage), x509:keyUsage(Leaf, KeyUsage),
  x509:fingerprint(Leaf, Fingerprint),
  x509:notBefore(Leaf, Lower), x509:notAfter(Leaf, Upper),
  x509:signature(Leaf, Signature, Params),
  env:stapledResponse(Leaf, StapledResponse),
  env:ocspResponse(Leaf, OcspResponse),
  getEVStatus(Leaf, EVStatus),
  getBasicConstraints(Leaf, BasicConstraints),
  validLeaf(Fingerprint, SANList, CommonName, Lower, Upper,
            Signature, BasicConstraints, KeyUsage, ExtKeyUsage,
            EVStatus, StapledResponse, OcspResponse).
  x509:signsAndIssuer(Leaf, Parent),
  validNonLeaf(Parent, 0, Leaf).
```

Listing 1: A highly abridged section of the Hammurabi-Firefox Prolog policy. `validChain` is the entrypoint.

(e.g., HTTP requests, signature checking) to produce required revocation facts, as specified by the user-agents' policy. Our prototype currently supports OCSP and OCSP stapling revocation checking.

Policy Execution. Our prototype executes Prolog policies using the SWI-Prolog (`swipl`) interpreter [122]. After loading the aggregated facts and the policy itself, Hammurabi executes the query `validChain(cert_0)?`. Every policy must provide a `validChain` rule which serves as the entry point. By convention, `cert_0` is the leaf certificate. If the query `validChain(cert_0)?` succeeds, the target leaf certificate is valid.

`swipl` allows us to "compile" the Hammurabi policy, root store, and some environment facts by checkpointing the program state. We do this out-of-band to speed up policy execution. At certificate validation time, the program resumes from the saved state, loads the certificate facts, and executes the entry point query. We expect a production-grade Hammurabi engine to use additional techniques for efficiency.[8]

7 EVALUATION

In this section, we evaluate Hammurabi from four perspectives: (1) can Hammurabi correctly capture existing certificate validation policies, (2) can Hammurabi identify meaningful differences between certificate validation policies using imputation, (3) how difficult is it to integrate Hammurabi into real user-agents, and (4) what performance impact would our (proof-of-concept, unoptimized) Hammurabi prototype have on the validation process relative to existing implementations, and is there room for optimization?

7.1 Correctness

The Hammurabi policy language must be (1) powerful enough to express the policy logic found in state-of-the-art TLS certificate validation implementations while (2) remaining concise and easy to reason about. To that end, we manually translated the certificate validation logic found in Mozilla Firefox and Google Chrome (both written in C++) into Prolog. In this section, we evaluate the correctness and coverage of our Hammurabi policy implementations.

Setup. We evaluate the correctness of our Prolog browser implementations by checking whether *Hammurabi-Firefox* (our Prolog version of Firefox's policy) and Firefox itself produce the same validation results for a large test set of real-world TLS certificate chains. We repeat the process for *Hammurabi-Chrome* and Chrome.[9]

Hammurabi-Firefox and Hammurabi-Chrome are each less than 650 lines of code. Comparing the number of lines of C++ browser validation logic to the number of lines of Prolog is a challenge since the code in browsers to validate a certificate is deeply entangled with parsing, revocation network requests, caching, cryptographic signature checking, etc. Just one of the relevant files in Firefox (`NSSCertDBTrustDomain.cpp`) is 1.8K lines of C++, and this does not include parsing, revocation checking, and more. An analogous file in Chrome (`cert_verify_proc.cc`) is 1K lines of C++, with an additional 500 lines each for iOS, Mac, and Windows platforms. We estimate that the amount of code *directly related* to certificate validation in Chrome and Firefox respectively is ~5K lines.

We do not claim to have completely emulated *all* of Firefox and Chrome's certificate validation behavior. For instance, Firefox and

[8] For instance, while our prototype aggregates facts and rules into files (as input to the Prolog interpreter), a production-grade implementation would integrate its engine and interpreter to keep facts and rules in memory.

[9] We use Chromium's verification logic for Linux at git commit `0590dcf7b03` and Firefox's at mercurial changeset `610414:dbd5ee74c531`.

Chrome also have special rules for imported root and intermediate certificates, which we omitted for brevity in our Prolog implementations, though we have no reason to believe it would be any harder to implement such policies in Hammurabi.

Instead, we intend to demonstrate that it is possible to express the complex validation logic found in Firefox and Chrome using our policy language. As our results in this section show, our Prolog implementations agree with the canonical browser implementations on all certificates "found in the wild" and all synthetic certificates except four rejected by Chrome due to parsing errors.

7.1.1 In-the-Wild Validation. To validate the correctness of our browser Prolog implementations with respect to actual browsers, we examine the differences in validation responses across certificates found in the wild. We first collect ~10M certificates from nine public CT logs [105] (Pilot, Rocketeer, Skydiver, Argon2022, Argon2021, Argon2020, Xenon2022, Xenon2021, and Xenon2020). We consider only non-expired certificates and exclude pre-certificates. 97% of the certificates we tested with were issued between August 1st and October 1st, 2020. We validated all certificates on October 2nd, 2020 UTC.[10] Since the CT logs often contain only the leaves of certificate chains, we use metadata (the AIA field) found in the leaves to fetch and construct about 10M candidate certificate chains. We disable OCSP revocation checking in all experiments to (1) avoid overwhelming OCSP servers, (2) avoid mismatched OCSP responses as a result of soft-failures (e.g., Firefox is unable to contact the OCSP responder and accepts the certificate, while Hammurabi-Firefox receives a `REVOKED` response), and (3) speed up the experiments.

We use a binary validation result for each chain: each validator deems the certificate chain either valid or invalid. For all 10,603,456 certificates, both Hammurabi-Chrome and Hammurabi-Firefox provide the same validity results as Chrome and Firefox, respectively.[11]

7.1.2 Synthetic Validation. A limitation of using certificates from CT logs for testing is that the vast majority are valid (99.998% in our case—all four validators agree that only ~17K of the 10M CT certificates are invalid). To test our implementations against a more diverse set of (often invalid) certificates, we apply the "Frankencerts" fuzzing technique of Brubaker et al. [27]. We randomly sampled a set of 1K leaves and 613 intermediates from the same CT logs as seeds to generate 100K synthetic certificate chains. The certificates in these chains are assembled using elements randomly taken from seed certificates and randomly generated elements (e.g., extensions, field values, and critical bits). As expected, some of these certificates do not even parse due to malformed extensions and fields.

Of the 100K synthetic chains, Hammurabi-Chrome and Chrome disagree on the validity of four. These four are considered valid by Hammurabi-Chrome and invalid by Chrome due to a parsing error (Chrome returned `net::ERR_CERT_INVALID`). Hammurabi-Firefox and Firefox agree on the validity of all synthetic certificates. Notably, Firefox and Chrome disagree on the validity of 9,544 of the 100K synthetic certificates, underscoring how different their validation implementations are (this degree of disagreement is expected, as these certificates were generated by fuzzing and likely do not resemble real certificates). Despite these promising results, we do not claim that our Hammurabi-Firefox and Hammurabi-Chrome implementations are complete. Rather, we believe the CT logs and these Frankencert results demonstrate the feasibility of implementing complex TLS validation logic correctly in Prolog.

[10] We hard-coded this date as the current time in all of our experiments.

[11] Ignoring our Hammurabi implementations, the canonical Firefox and Chrome browsers disagree on the validity of 48 certificates.

Validator	Mean	Median	Max	StdDev
Chrome	2.73	2.0	39	1.11
Firefox	1.81	2.0	423	2.47
Hammurabi-Chrome	2.05	2.0	21	0.31
Hammurabi-Firefox	2.07	2.0	19	0.35

Table 3: Certificate validation times (ms) for browsers versus their corresponding Hammurabi Prolog implementations.

7.2 Validation Performance

Although performance is not our primary concern, we nonetheless investigate the performance of executing our Hammurabi policies using our unoptimized prototype. We compare the time taken by Hammurabi-Firefox, Hammurabi-Chrome, Firefox, and Chrome to validate a subset of our 10M certificate-chain corpus. Before running the experiment, as mentioned in §6, we compile both the Hammurabi-Firefox and Hammurabi-Chrome policies into what SWI-Prolog calls "saved state". In a realistic deployment, we similarly expect developers to compile and load their policies before starting validation to reduce the latency. For this analysis we disable OCSP and CRL checking for all four validators.

In Table 3 we show the results from timing the validation of 105K certificate chains sampled from the 10M dataset described in §7.1. We ran each benchmark on a Dell XPS 9560 laptop with an Intel i7-7700HQ 2.8 GHz 16-thread CPU, 32 GB of memory, and a 2TB NVMe SSD, running Ubuntu 20.04, kernel v5.13. We measure the wall-clock running time of the `net/cert/cert_verify_proc.cc#Verify` method in Chrome, the `nsNSSCertificateDB.cpp#AsyncVerifyCertAtTime`[12] function in Firefox, and the `validChain` query in Hammurabi-Chrome and Hammurabi-Firefox. Mean validation times range from 1.81–2.73ms across browser implementations and 2.05–2.07ms across Prolog implementations, while the median validation time is 2ms for all validators. The Hammurabi times include the time taken to load certificate facts and run the query, and do not include the times taken to (1) parse certificates (which all validators must do), (2) translate certificates into Prolog facts (which can be cached for intermediate and root certificates), (3) validate signatures of candidate subject-issuer pairs (which can be cached), and (4) run the Prolog binary (which can be amortized across all validations). Steps (2) and (3) take an average of 24ms in our unoptimized prototype.

Considering that our Hammurabi prototype is unoptimized, we find these results promising. We believe a deployment-caliber Hammurabi engine would be able to minimize performance gaps by implementing the optimizations above and by using a Prolog engine designed for low-latency queries. To put these benchmark times in context of overall HTTP request latency, Google found that the average website time-to-first-byte (which includes certificate validation) ranged between 1.8–2.7 seconds in 2017 [19].

[12] The full path is `mozilla-unified/security/manager/ssl/nsNSSCertificateDB.cpp`.

7.3 Imputation

As a demonstration of how imputation can be used to programmatically find differences between certificate validation policies, we compare our Hammurabi-Firefox and Hammurabi-Chrome policies. We limit our imputation experiments to leaf certificates, for now. Both Hammurabi-Firefox and Hammurabi-Chrome include `validLeaf` rules (shown in §5.5), which hold if the leaf certificate passes preliminary checks. `validLeaf` takes the following operands, each listed with its possible values:

- **OCSP Response.** An OCSP response has four boolean values that indicate the following: syntactic validity, expiration, signature verification, and status (revoked or not).
- **Stapled OCSP Response.** A stapled OCSP response has the four OCSP Response booleans plus one more to indicate whether a stapled OCSP response was received.
- **Key Usage** and **Extended Key Usage (EKU).** These extensions each contain a list of enumerable key usage values.
- **CA status.** The Basic Constraints extension contains a boolean indicating whether the certificate can be used as a CA certificate and a path length. We enumerate over the boolean only.
- **Signature Algorithm.** The signature algorithm is drawn from an enumerable list of OIDs.
- **EV status.** A certificate is EV if it contains an EV OID, which can vary by intermediate.
- **Not Before.** A Unix timestamp denoting when the certificate's validity period starts.
- **Not After.** A Unix timestamp denoting when the certificate's validity period ends.
- **Common Name.** The Common Name is (for our purposes) a DNS name, whose possible values are described §5.4.
- **SAN List.** The Subject Alternative Names extension is a list of DNS names. We only consider SAN Lists of length zero and one.
- **Hostname.** The hostname is an environment fact and a DNS name denoting the subject name.

We then construct a new rule `firefoxOnly` (and `chromeOnly`), which takes the above operands, restricts them to their enumerable values, and holds if the values are valid in Hammurabi-Firefox but not Hammurabi-Chrome:

```
firefoxOnly(SANList, CommonName, Lower, Upper,
            Algorithm, BasicConstraints, KeyUsage, ExtKeyUsage,
            EVStatus, StapledResponse, OcspResponse, Hostname) :-
  restrict(...), % Limits each field to enumerable values.
  firefox:validLeaf(...), % "..." denotes all rule operands, for brevity.
  \+chrome:validLeaf(...).
```

We can then impute over subsets of operands as follows. First, we pick an operand or operands to impute over—consider `SANList`, `CommonName`, and `Hostname`, which should illuminate differences in name checking between the two policies. We then construct a query `firefoxOnly(...).`, using concrete values for all *other* operands, while keeping `SANList`, `CommonName`, and `Hostname` as Prolog variables. Prolog will automatically then produce all possible values for our three chosen operands that make the rule hold, i.e., that are valid in Hammurabi-Firefox but not in Hammurabi-Chrome. We then repeat the process for different subsets of concrete and variable operands to reveal further differences.

Our imputation analysis uncovered the following differences between Chrome and Firefox. We confirmed the differences using a combination of differential testing, manual analysis of the original browser source code, and querying the Chrome and Firefox teams directly. We confirmed that these policy differences are intentional (yet obscure) rather than errors or bugs.

TLD Wildcard. The wildcard character (`*`) is allowed in the Common Name or Subject Alternative Names as the entirety of the leftmost label (labels are separated by dots) as long as the remaining label is not a registry-controlled TLD. For instance, `*.example.com` is a valid name, but `*.com` is not. The CA/B BRs recommend consulting the Public Suffix List [2] to identify registry-controlled TLDs, which Chrome does, despite the maintainer advising against it [1]. Firefox instead checks whether there are at least two labels to the right of the wildcard. The result is that names like `*.co.uk` are valid in Firefox but invalid in Chrome.

Common Name Fallback. Per RFC 5280, TLS user-agents should process targeted domains in the Common Name or Subject Alternative Names fields. RFC 2818 and the CA/B BRs deprecate the usage of the Common Name field, and discourage its use. Chrome *only* processes domains in the Subject Alternative Names and ignores the Common Name. Firefox will process domains in the Common Name if and only if the Subject Alternative Name extension is omitted or empty.[13]

Leaf `keyCertSign` Key Usage. According to RFC 5280: "[i]f the `cA` boolean is not asserted, then the `keyCertSign` bit in the key usage extension MUST NOT be asserted." Chrome adheres to this rule, but Firefox allows certificates with `cA` set to `false` to include the `keyCertSign` Key Usage bit, for compatibility reasons.

Leaves with CA enabled. The CA/B BRs specify that end-entity or subscriber certificates MUST NOT set the CA bit to true. Firefox correctly rejects leaf certificates with the CA bit set to true, while Chrome accepts them. Interestingly, Chrome will only accept such a leaf if it *also* sets a maximum path length and has the `keyCertSign` key usage.

Revocation. While the CA/B BRs do not require browsers to perform revocation checking of any kind, almost all do, but to varying degrees. Firefox checks its small local database of revoked intermediates (OneCRL) and performs OCSP revocation checking for all certificates by default. Firefox is also currently testing CRLite, an efficient, private, and complete revocation mechanism [76]. Firefox also supports OCSP Must-Staple. Chrome checks its small local database (CRLSet) and only performs OCSP checking for leaf EV certificates. Chrome does not support OCSP Must-Staple.

Maximum Certificate Lifetime. RFC 5280 does not specify a maximum certificate lifetime, but the CA/B BRs specify a maximum leaf certificate lifetime of 398 days for certificates issued after September 1st, 2020. A certificate's lifetime is the time between its `notBefore` and `notAfter` fields. Chrome enforces this restriction, while Firefox does not (yet) [90, 98]—the maximum certificate

[13] This behavior is configurable using the `security.pki.name_matching` configuration preference. Note that by default, Firefox will not fall back to the Common Name, but we modeled and evaluated Firefox as if this preference was set to "fallback".

lifetime for EV leaf certificates is 27 months, and for non-EV leaf certificates is unbounded.

RSA PSS Algorithms. The version of Firefox we used to evaluate Hammurabi does not allow the `id-RSASSA-PSS` algorithm identifier (OID `1.2.840.113549.1.1.10`), while Chrome does.

`anyExtendedKeyUsage`. The Extended Key Usage extension allows CAs to further restrict the purposes for which certificates can be used. For example, the `serverAuth` EKU indicates that the certificate can be used to authenticate a web server during a TLS handshake. One possible EKU is `anyExtendedKeyUsage`, which was intended to denote that the certificate can be used for any EKU purpose. However, the CA/B BRs state that the `anyExtendedKeyUsage` EKU "MUST NOT be present". Neither Firefox nor Chrome reject certificates that include the `anyExtendedKeyUsage` EKU. But if it is present, Firefox skips over it entirely—in other words, Firefox will still require `serverAuth`, regardless of whether `anyExtendedKeyUsage` is present or not. Chrome, on the other hand, does not require `serverAuth` if `anyExtendedKeyUsage` is present—it treats `anyExtendedKeyUsage` as a catch-all for all EKUs.

`ocspSigning` Extended Key Usage. Firefox rejects leaf certificates that include the EKU `ocspSigning`, while Chrome accepts them (as long as the EKU is otherwise valid). The BRs (7.1.2.3) specify that the value "SHOULD NOT" be present. From the Firefox source:

> When validating anything other than an [sic] delegated OCSP signing cert, reject any cert that also claims to be an OCSP responder, because such a cert does not make sense. For example, if an SSL certificate were to assert `id-kp-OCSPSigning` then it could sign OCSP responses for itself, if not for this check.

7.4 Integration

Finally, to show Hammurabi can be integrated into real-world TLS user-agents, we modified Mozilla Firefox and the Go programming language (which uses its own native TLS and X.509 libraries) to use our Hammurabi daemon. Rather than executing their own certificate validation routines during the TLS handshake, the two user-agents send the certificates and domain name to the Hammurabi daemon, which executes a Prolog policy and returns the result and validated chain (all over a local TCP socket). One interesting consequence of this integration is that we can instruct Firefox to validate certificates according to Chrome's policy. Integrating Hammurabi into Firefox required approximately 100 lines of additional code; integrating into Go required approximately 50.

8 RELATED WORK

We now provide an overview of recent work on the PKI.

Measurements. There is a long thread of studies examining the web's certificate ecosystem from various vantage points [50, 51, 62, 63, 71, 94, 107, 120]. Studies have specifically examined the cost of HTTPS [91]; how certificates are managed from the web administrator side [29, 36, 127], the CA side [73], within root stores [84, 119], and by CDNs [29, 80]; and the response to specific security vulnerabilities [126] like Heartbleed [49, 127]. Other studies have measured the deployment of new security features such as CAA [104], SCSV, and others [18]. Huang et al. studied the usage of forged TLS certificates [64]. As the reach of the HTTPS protocol has moved beyond web browsers, studies are now examining the adoption of web PKI primitives across a variety of application domains [20, 61], including mobile apps [97], IoT devices [93], and DNS [26, 33, 83].

These measurement studies reveal that the TLS ecosystem is constantly in flux: protocols change, certificate features get adopted and deprecated, and applications evolve. We believe that these observations motivate the need for systems like Hammurabi, to help manage this complexity across time and application-domains.

Formalization. Given the critical importance of TLS to the internet, there have been several efforts to guarantee its correctness and security properties through the application of formal and symbolic methods. Recent efforts have focused on proving the properties of the TLS 1.3 protocol in general, and of specific implementations of the protocol [24, 41, 42, 47]. Other studies have used symbolic execution and formal specification to examine the correctness of X.509 certificate parsing and validation code [32, 45].

Hammurabi complements these techniques by bringing more rigor to the specification and evaluation of X.509 certificate validation logic. It does not attempt to formally specify X.509 certificate parsing or TLS protocol implementations, but could easily be combined with systems that do.

Alternative Designs. Alternate architectures to the current CA-based systems have been proposed [22, 38]. For example, DNS-based Authentication of Named Entities (DANE) replaces the CA-based system by mirroring the existing DNS hierarchy [60]. Delignat-Lavaud et al. [46] propose replacing the existing X.509 format with a zero-knowledge verifiable computation scheme [40]. Lee et al. proposes a new framework that uses the cloud to enable evolution of PKI enhancements [78]. Techniques have also been suggested to improve the existing TLS ecosystem [69, 85, 95, 102, 113, 114]. For example, Ryan proposes extending certificate transparency to support end-to-end encrypted mail [102].

Revocation. The number of revoked certificates continues to grow significantly, making distributing revocation information to user-agents a challenging task. Several studies attempt to measure OCSP lookup latency and suggest improvements [75, 111, 128]. Topalovic et al. proposes using short-lived certificates to reduce the size of revocation data [115]. CRLite uses a filter cascade to compress the set of revoked certificates [76]. Schulman et al. proposes a method for disseminating revocation data over FM radio [106]. The efforts of CAs and website administrators are, however, pointless if user-agents do not perform OCSP lookups or utilize the response, as has been observed in prior work [81]. Chuat et al. present a detailed framework for thinking through the costs and benefits of various delegation and certificate revocation schemes [35].

Hammurabi may make it easier for developers to choose revocation checking policies that match their use case, by separating these policy choices from low-level certificate parsing mechanisms.

9 CONCLUSION

X.509 certificate validation is a complicated ecosystem, with constantly evolving best practices and policies hidden inside complex

codebases. In this paper, we aimed to address these issues by separating low-level mechanism (which rarely evolves) from the high-level policy (which must adapt to changing circumstances). We presented a framework, Hammurabi, that cleanly separates the two.

The more sophisticated, interesting, and evolvable portions of Hammurabi are the policies themselves, which we express in a high-level logic-based programming language, Prolog. To demonstrate the expressiveness of logic-based languages in this domain, we reimplemented Chrome and Firefox's validation policies. In addition to being able to faithfully and concisely represent these implementations, Prolog gives us a valuable feature for free: the ability to leave some features of certificates unspecified, and to have the language automatically impute valid values for them. This makes it straightforward to interrogate the differences between policies ("what are certificate values such that Firefox validates but Chrome does not?") and to assist administrators in choosing compliant values ("what is a value such that Chrome and Firefox both validate?").

Although it is not surprising that a high-level language like Prolog yields more concise code than C, C++, or even Go, what is surprising is how amenable logic-based languages are to certificate validation. Indeed, in retrospect it seems obvious that certificate validation is a set of logical constraints: what better way to represent them than with a language made up of logical constraints?

9.1 Deployment Considerations

Crucially, Hammurabi is incrementally deployable. Different TLS user-agents already adopt a variety of certificate validation policies—Hammurabi does not change this status quo, and user-agents that choose to adopt it will have no impact on those that do not. Furthermore, deploying Hammurabi does not necessitate changes to the TLS protocol, certificates, CAs, or root stores.

The interface we described for Hammurabi in §4.1 is sufficiently abstract that there are a variety of concrete ways it could be implemented. The particular way in which Hammurabi is integrated into user-agents will undoubtedly have tradeoffs, thus we present several options for concrete implementations.

System service. Hammurabi can be implemented as a system-level service. Any user-agent wishing to validate a certificate simply forwards the arguments to the Hammurabi daemon using IPC. Although this design may result in performance overhead, it is compatible with all programming languages and allows the service to cache data (e.g., intermediate certificates and revocation information) at the OS, rather than user-agent, level. As we describe in §6, our Hammurabi prototype is implemented as a daemon.

Shared library. Hammurabi could be implemented as a library that user-agents would bind to (e.g., like OpenSSL or NSS). This would eliminate IPC overhead, but may not be compatible with all platforms and programming languages.

Bespoke implementations. Finally, each user-agent could implement its own Hammurabi-standard engine. In this model, the user-agent must produces the exact same certificate facts and execute policies in the exact same way as other Hammurabi implementations (described in §5), to ensure policy portability. Some high-profile user-agents (e.g., web browsers) may choose this approach for performance reasons, though it requires careful interface standardization to ensure consistency across implementations.

9.2 Future Outlook

We conclude this paper by considering what a future web PKI might look like were it to adopt Hammurabi.

Hammurabi could make it easier for standards bodies to publish certificate validation policies, and easier for developers to adopt them. For example, the CA/B could also publish supplementary Prolog rules that precisely capture their proposed text-based policies. Developers could simply include the new Prolog in their Hammurabi validation policies rather than translating (often ambiguous) text into code. Similarly, the IETF could publish executable policies in Prolog alongside new RFCs. Having a canonical validation policy from the IETF would facilitate detection (via imputation) of cases where user-agents' policies deviate from the RFC standard (as we showed in §10.1, such deviations do occur).

As a proof-of-concept, we translated ZLint [129] into a Hammurabi validation policy. The Mozilla root program recommends that CAs perform pre-issuance linting of their certificates with ZLint (or a similar tool) to make sure they comply with RFC and BR requirements [88]. *Hammurabi-ZLint* demonstrates that Prolog can be used to express baseline X.509 certificate validation standards. As an added bonus, user-agents could directly incorporate Hammurabi-ZLint into their validation policy if they were concerned about encountering misissued certificates in-the-wild.

This discussion highlights the flexibility that Hammurabi offers to developers and power users. Today, if a developer wants to change validation policy in their user-agent, they must either modify validation logic that is enmeshed with validation mechanisms, or switch to an entirely different tool (e.g., from OpenSSL to NSS) that offers the desired policy. In contrast, once a developer adopts Hammurabi, they are free to choose any available validation policy that meets their needs or change policies at any time, without switching tools (or even recompiling their software, depending on the Hammurabi implementation). Likewise, if a power user wished to take on a stricter posture than their user-agent, they could simply append additional Prolog rules to the policy.

Finally, the existence of well-specified, executable validation policies may make the adoption of these strong policies more ubiquitous across all TLS user-agents rather than just browsers.

These use cases point to a web PKI in which certificate validation policy is easier to articulate, easier to reason about, easier to adopt, more transparent, and easier to update in the face of new risks.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful comments. We also thank Zachary Hanif, Olamide Omolola, and Aaron Bembenek for their valuable help. This research was supported in part by the NSF under grants CNS-1901047, CNS-1901090, CNS-1901325, CNS-1900879, CNS-1900996, and CNS-2053363. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funders.

REFERENCES

[1] [n. d.]. Proper handling for wildcard certificates for all tlds. https://bugzilla.mozilla.org/show_bug.cgi?id=1196364.
[2] [n. d.]. Public Suffix List. https://publicsuffix.org/list/public_suffix_list.dat.
[3] 2002. CVE-2002-0862. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2002-0862.
[4] 2003. CVE-2003-1229. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2003-1229.
[5] 2005. CVE-2005-3170. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-3170.
[6] 2008. CVE-2008-4989. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-4989.
[7] 2009. CVE-2009-2408. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-2408.
[8] 2010. CVE-2010-1378. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-1378.
[9] 2011. CVE-2011-0228. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-0228.
[10] 2012. CVE-2012-3446. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-3446.
[11] 2014. CVE-2014-0092. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0092.
[12] 2015. Crlsets. The Chromium Projects. http://bit.ly/1JPsUeC.
[13] 2015. Issue 572734: Support for OCSP Must-staple. https://bugs.chromium.org/p/chromium/issues/detail?id=572734.
[14] 2016. Feature request: OCSP Must Staple (RFC 7633). https://groups.google.com/a/chromium.org/g/security-dev/c/-pB8IFNu5tw.
[15] 2021. CVE-2021-3450. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3450.
[16] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison-Wesley Reading.
[17] Jalal Al-Muhtadi, Anand Ranganathan, Roy Campbell, and M Dennis Mickunas. 2003. Cerberus: a context-aware security scheme for smart spaces. In *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications, 2003.(PerCom 2003)*. IEEE, 489–496.
[18] Johanna Amann, Oliver Gasser, Quirin Scheitle, Lexi Brent, Georg Carle, and Ralph Holz. 2017. Mission Accomplished? HTTPS Security after DigiNotar. In *Proceedings of the ACM Internet Measurement Conference (IMC)*.
[19] Daniel An. 2018. Find out how you stack up to new industry benchmarks for mobile page speed. *" Think with Google-Mobile, Data & Measurement"* (2018).
[20] Blake Anderson and David McGrew. 2019. TLS Beyond the Browser: Combining End Host and Network Data to Understand Application Behavior. In *Proceedings of the ACM Internet Measurement Conference (IMC)*.
[21] Thomas H. Austin, Jean Yang, Cormac Flanagan, and Armando Solar-Lezama. 2013. Faceted execution of policy-agnostic programs. In *Proceedings of the Eighth ACM SIGPLAN workshop on Programming languages and analysis for security*. 15–26.
[22] Adam Bates, Joe Pletcher, Tyler Nichols, Braden Hollembaek, and Kevin R.B. Butler. 2014. Forced Perspectives: Evaluating an SSL Trust Enhancement at Scale. In *Proceedings of the ACM Internet Measurement Conference (IMC)*.
[23] Aaron Bembenek, Michael Greenberg, and Stephen Chong. 2020. Formulog: Datalog for SMT-based static analysis. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–31.
[24] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. 2017. Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate. In *Proceedings of the IEEE Symposium on Security and Privacy*.
[25] Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Catalin Hritcu, Samin Ishtiaq, Markulf Kohlweiss, Rustan Leino, Jay Lorch, Kenji Maillard, Jinyang Pang, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Ashay Rane, Aseem Rastogi, Nikhil Swamy, Laure Thompson, Peng Wang, Santiago Zanella-Beguelin, and Jean-Karim Zinzindohoué. 2017. Everest: Towards a Verified, Drop-In Replacement of HTTPS. In *Summit on Advances in Programming Languages (SNAPL)*.
[26] Timm Böttger, Felix Cuadrado, Gianni Antichi, Eder Leão Fernandes, Gareth Tyson, Ignacio Castro, and Steve Uhlig. 2019. An Empirical Study of the Cost of DNS-over-HTTPS. In *Proceedings of the ACM Internet Measurement Conference (IMC)*.
[27] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. 2014. Using Frankencerts for Automated Adversarial Testing of Certificate Validation In SSL/TLS Implementations. In *Proceedings of the IEEE Symposium on Security and Privacy*.
[28] CA/Browser Forum. 2016. Baseline Requirements: Certificate Policy for the Issuance and Management of Publicly-Trusted Certificates. Version 1.4.1. https://cabforum.org/wp-content/uploads/CA-Browser-Forum-BR-1.4.1-redlined.pdf
[29] Frank Cangialosi, Taejoong Chung, David Choffnes, Dave Levin, Bruce M. Maggs, Alan Mislove, and Christo Wilson. 2016. Measurement and Analysis of Private Key Sharing in the HTTPS Ecosystem. In *Proceedings of ACM CCS*.
[30] CA:Symantec Issues [n. d.]. CA:Symantec Issues. https://wiki.mozilla.org/CA:Symantec_Issues.
[31] Certificate Transparency Enforcement in Google Chrome [n. d.]. Certificate Transparency Enforcement in Google Chrome. https://groups.google.com/a/chromium.org/forum/#!topic/ct-policy/wHILiYf31DE.
[32] Sze Yiu Chau, Omar Chowdhury, Endadul Hoque, Huangyi Ge, Aniket Kate, Cristina Nita-Rotaru, and Ninghui Li. 2017. SymCerts: Practical Symbolic Execution For Exposing Noncompliance in X.509 Certificate Validation Implementations. In *Proceedings of the IEEE Symposium on Security and Privacy*.
[33] Rishabh Chhabra, Paul Murley, Deepak Kumar, Michael Bailey, and Gang Wang. 2021. Measuring DNS-over-HTTPS Performance Around the World. In *Proceedings of the ACM Internet Measurement Conference (IMC)*.
[34] Chrome Root Program 2020. Chrome Root Program. The Chromium Projects. https://www.chromium.org/Home/chromium-security/root-ca-policy/.
[35] Laurent Chuat, AbdelRahman Abdou, Ralf Sasse, Christoph Sprenger, David Basin, and Adrian Perrig. 2020. SoK: Delegation and Revocation, the Missing Links in the Web's Chain of Trust. In *Proceedings of the IEEE European Symposium on Security and Privacy*.
[36] Taejoong Chung, Yabing Liu, Dave Choffnes, Dave Levin, Bruce Maggs, Alan Mislove, and Christo Wilson. 2016. Measuring and Applying Invalid SSL Certificates: The Silent Majority. In *Proceedings of the ACM Internet Measurement Conference (IMC)*.
[37] Catalin Cimpanu. 2020. Chrome will soon have its own dedicated certificate root store. https://www.zdnet.com/article/chrome-will-soon-have-its-own-dedicated-certificate-root-store/.
[38] Convergence [n. d.]. Convergence. http://convergence.io.
[39] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. 2008. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280. http://www.ietf.org/rfc/rfc5280.txt
[40] Craig Costello, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur and. 2015. Geppetto: Versatile Verifiable Computation. In *Proceedings of the IEEE Symposium on Security and Privacy*.
[41] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. 2017. A Comprehensive Symbolic Analysis of TLS 1.3.
[42] Cas Cremers, Marko Horvat, Sam Scott, and Thyla van der Merwe. 2016. Automated Analysis and Verification of TLS 1.3: 0-RTT, Resumption and Delayed Authentication. In *Proceedings of the IEEE Symposium on Security and Privacy*.
[43] CRLite - CA/Browser Forum [n. d.]. CRLite - CA/Browser Forum. https://cabforum.org/wp-content/uploads/CABF_F2Fpreso_030518_vmf.pdf.
[44] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. 2001. The Ponder Policy Specification Language. In *Proceedings of the International Workshop on Policies for Distributed Systems and Networks*. Springer, 18–38.
[45] Joyanta Debnath, Sze Yiu Chau, and Omar Chowdhury. 2021. On Re-engineering the X.509 PKI with Executable Specification for Better Implementation Guarantees. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 1388–1404.
[46] Antoine Delignat-Lavaud, Cedric Fournet, Markulf Kohlweiss, and Bryan Parno. 2016. Cinderella: Turning Shabby X.509 Certificates into Elegant Anonymous Credentials with the Magic of Verifiable Computation. In *Proceedings of the IEEE Symposium on Security and Privacy*.
[47] Antoine Delignat-Lavaud, Cedric Fournet, Markulf Kohlweiss, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy, Santiago Zanella-Beguelin, Karthikeyan Bhargavan, Jianyang Pan, and Jean Karim Zinzindohoue. 2017. Implementing and Proving the TLS 1.3 Record Layer. In *Proceedings of the IEEE Symposium on Security and Privacy*.
[48] Distrust of the Symantec PKI: Immediate action needed by site operators [n. d.]. Distrust of the Symantec PKI: Immediate action needed by site operators. https://security.googleblog.com/2018/03/distrust-of-symantec-pki-immediate.html.
[49] Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael Bailey, Frank Li, Nicolas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, and Vern Paxson. 2014. The Matter Of Heartbleed. In *Proceedings of the ACM Internet Measurement Conference (IMC)*.
[50] Zakir Durumeric, James Kasten, Michael Bailey, and J. Alex Halderman. 2013. Analysis of the HTTPS Certificate Ecosystem. In *Proceedings of the ACM Internet Measurement Conference (IMC)*.
[51] EFF SSL Observatory [n. d.]. EFF SSL Observatory. https://www.eff.org/observatory.
[52] Eslam Elnikety, Aastha Mehta, Anjo Vahldiek-Oberwagner, Deepak Garg, and Peter Druschel. 2016. Thoth: Comprehensive Policy Compliance in Data Retrieval Systems. In *25th USENIX Security Symposium (USENIX Security 16)*. 637–654.
[53] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. 2012. Why Eve and Mallory Love Android: An Analysis of Android SSL (In)Security. In *Proceedings of ACM CCS*. Raleigh, North Carolina, USA.
[54] Sascha Fahl, Marian Harbach, Henning Perl, Markus Koetter, and Matthew Smith. 2013. Rethinking SSL Development in an Appified World. In *Proceedings of ACM CCS*. Berlin, Germany.
[55] Alex Gaynor. 2014. *Enabling certificate verification by default for stdlib http clients*. PEP 476. https://peps.python.org/pep-0476
[56] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. 2012. The Most Dangerous Code In The World: Validating SSL Certificates In Non-browser Software. In *Proceedings of ACM CCS*.
[57] Mark Goodwin. 2015. Revoking Intermediate Certificates: Introducing OneCRL. Mozilla Security Blog. `http://mzl.la/1zLFp7M`.
[58] Peter Gutmann. 2000. X.509 Style Guide.
[59] P. Hallam-Baker. 2015. X.509v3 Transport Layer Security (TLS) Feature Extension. RFC 7633. http://www.ietf.org/rfc/rfc7633.txt
[60] P. Hoffman and J. Schlyter. 2012. The DNS-based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA. RFC 6698. https://www.ietf.org/rfc/rfc6962.txt
[61] Ralph Holz, Johanna Amann, Olivier Mehani, Matthias Wachs, and Mohamed Ali Kaafar. 2016. TLS in the Wild: An Internet-wide Analysis of TLS-based Protocols for Electronic Communication. In *Proceedings of NDSS*.
[62] Ralph Holz, Lothar Braun, Nils Kammenhuber, and Georg Carle. 2011. The SSL Landscape – a Thorough Analysis of the X.509 PKI Using Active and Passive Measurements. In *Proceedings of the ACM Internet Measurement Conference (IMC)*.
[63] Ralph Holz, Jens Hiller, Johanna Amann, Abbas Razaghpanah, Thomas Jost, Narseo Vallina-Rodriguez, and Oliver Hohlfeld. 2020. Tracking the Deployment of TLS 1.3 on the Web: A Story of Experimentation and Centralization. *SIGCOMM Comput. Commun. Rev.* 50, 3 (July 2020), 3–15.
[64] Lin-Shung Huang, Alex Rice, Erling Ellingsen, and Collin Jackson. 2014. Analyzing Forged SSL Certificates In The Wild. In *Proceedings of the IEEE Symposium on Security and Privacy*. San Jose, California, USA.
[65] Ian Haken. [n. d.]. BetterTLS. https://netflixtechblog.com/bettertls-c9915cd255c0.
[66] International Telecommunications Union. 1988. ITU-T Recommendation X.509: The Directory: Authentication Framework. Technical Report X.509.
[67] Trevor Jim. 2000. SD3: A Trust Management System with Certified Evaluation. In *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*. IEEE, 106–115.
[68] Joe St Sauver. [n. d.]. How Many "Parts" (or "Labels") Does A Domain Name Typically Have? https://www.farsightsecurity.com/blog/txt-record/rrlabel-20171013/.
[69] 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 *Proceedings of WWW*.
[70] William A Kornfeld. 1983. Equality for Prolog. In *IJCAI*, Vol. 83. 514–519.
[71] Platon Kotzias, Abbas Razaghpanah, Johanna Amann, Kenneth G. Paterson, Narseo Vallina-Rodriguez, and Juan Caballero. 2018. Coming of Age: A Longitudinal Study of TLS Deployment. In *Proceedings of the ACM Internet Measurement Conference (IMC)*.
[72] Robert Krahn, Bohdan Trach, Anjo Vahldiek-Oberwagner, Thomas Knauth, Pramod Bhatotia, and Christof Fetzer. 2018. Pesos: Policy Enhanced Secure Object Store. In *Proceedings of the Thirteenth EuroSys Conference*. 1–17.
[73] Deepak Kumar, Zhengping Wang, Matthew Hyder, Joseph Dickinson, Gabrielle Beck, David Adrian, Joshua Mason, Zakir Durumeric, J. Alex Halderman, and Michael Bailey. 2018. Tracking Certificate Misissuance in the Wild. In *Proceedings of the IEEE Symposium on Security and Privacy*.

[74] Butler W. Lampson and Howard E. Sturgis. 1976. Reflections on an Operating System Design. *Commun. ACM* 19, 5 (1976), 251–265.

[75] Adam Langley. 2012. Revocation Checking and Chrome's CRL. https://www.imperialviolet.org/2012/02/05/crlsets.html.

[76] 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 *Proceedings of the IEEE Symposium on Security and Privacy*.

[77] Ben Laurie, Adam Langley, and Emilia Kasper. 2013. Certificate Transparency. RFC 6962. https://www.ietf.org/rfc/rfc6962.txt

[78] Taeho Lee, Christos Pappas, Pawel Szalachowski, and Adrian Perrig. 2018. Towards Sustainable Evolution for the TLS Public-Key Infrastructure. In *Proceedings of AsiaCCS*.

[79] Roy Levin, Ellis Cohen, William Corwin, Fred Pollack, and William Wulf. 1975. Policy/Mechanism Separation in Hydra. In *Proceedings of the Fifth ACM symposium on Operating Systems Principles*. 132–140.

[80] Jinjin Liang, Jian Jiang, Haixin Duan, Kang Li, Tao Wan, and Jianping Wu. 2014. When HTTPS Meets CDN: A Case of Authentication in Delegated Service. In *Proceedings of the IEEE Symposium on Security and Privacy*.

[81] Yabing Liu, Will Tome, Liang Zhang, David Choffnes, Dave Levin, Bruce Maggs, Alan Mislove, Aaron Schulman, and Christo Wilson. 2015. An End-to-end Measurement of Certificate Revocation in the Web's PKI. In *Proceedings of the ACM Internet Measurement Conference (IMC)*.

[82] Steve Lloyd. 2002. Understanding Certification Path Construction. In *PKI Forum White Paper*. http://www.oasis-pki.org/pdfs/Understanding_Path_construction-DS2.pdf.

[83] Chaoyi Lu, Baojun Liu, Zhou Li, Shuang Hao, Haixin Duan, Mingming Zhang, Chunying Leng, Ying Liu, Zaifeng Zhang, and Jianping Wu. 2019. An End-to-End, Large-Scale Measurement of DNS-over-Encryption: How Far Have We Come?. In *Proceedings of the ACM Internet Measurement Conference (IMC)*.

[84] Zane Ma, James Austgen, Joshua Mason, Zakir Durumeric, and Michael Bailey. 2021. Tracing Your Roots: Exploring the TLS Trust Anchor Ecosystem. In *Proceedings of the ACM Internet Measurement Conference (IMC)*.

[85] Stephanos Matsumoto, Pawel Szalachowski, and Adrian Perrig. 2015. Deployment Challenges in Log-based PKI Enhancements. In *European Workshop on Systems Security*. Bordeaux, France.

[86] Misissued/Suspicious Symantec Certificates [n. d.]. Misissued/Suspicious Symantec Certificates. https://groups.google.com/forum/#!msg/mozilla.dev.security.policy/fyJ3EK2YOP8/yvjS5leYCAAJ.

[87] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[88] Mozilla CA Recommendations 2017. CA/Required or Recommended Practices. Mozilla Wiki. https://wiki.allizom.org/CA/Required_or_Recommended_Practices.

[89] Mozilla Root Store Policy 2022. Mozilla Root Store Policy. Mozilla. https://www.mozilla.org/en-US/about/governance/policies/security-group/certs/policy/.

[90] Mozilla Root Store Policy Archive [n. d.]. Mozilla Root Store Policy Archive. https://wiki.mozilla.org/CA/Root_Store_Policy_Archive.

[91] David Naylor, Alessandro Finamore, Ilias Leontiadis, Yan Grunenberger, Marco Mellia, Maurizio Munafò, Konstantina Papagiannaki, and Peter Steenkiste. 2014. The Cost of the "S" in HTTPS. In *Proceedings of ACM CoNEXT*.

[92] Xinming Ou, Sudhakar Govindavajhala, Andrew W. Appel, et al. 2005. MulVAL: A Logic-based Network Security Analyzer.. In *Proceedings of the USENIX Security Symposium*. Baltimore, MD, 113–128.

[93] Muhammad Talha Paracha, Daniel J. Dubois, Narseo Vallina-Rodriguez, and David Choffnes. 2021. IoTLS: Understanding TLS Usage in Consumer IoT Devices. In *Proceedings of the ACM Internet Measurement Conference (IMC)*.

[94] Henning Perl, Sascha Fahl, and Matthew Smith. 2014. You Won't Be Needing These Any More: On Removing Unused Certificates From Trust Stores. In *Financial Cryptography and Data Security*.

[95] Matthew Prince. 2018. Encrypting SNI: Fixing One of the Core Internet Bugs. https://blog.cloudflare.com/esni/.

[96] Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej C hajed, Nadim Kobeissi, and Jonathan Protzenko. 2019. EverParse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats. In *Proceedings of the USENIX Security Symposium*.

[97] Abbas Razaghpanah, Arian Akhavan Niaki, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Johanna Amann, and Phillipa Gill. 2017. Studying TLS Usage in Android Apps. In *Proceedings of ACM CoNEXT*.

[98] Reducing TLS Certificate Lifespans to 398 Days 2020. Reducing TLS Certificate Lifespans to 398 Days . https://blog.mozilla.org/security/2020/07/09/reducing-tls-certificate-lifespans-to-398-days/.

[99] Ivan Ristic. 2013. *Bulletproof SSL and TLS: Understanding and Deploying SSL/TLS and PKI to Secure Servers and Web Applications* (2 ed.). Feisty Duck. 97 pages.

[100] Rusticata DER Parser [n. d.]. Rusticata DER Parser. https://github.com/rusticata/der-parser.

[101] Rusticata X.509 Parser [n. d.]. Rusticata X.509 Parser. https://github.com/rusticata/x509-parser.

[102] Mark Dermot Ryan. 2014. Enhanced Certificate Transparency and End-to-end Encrypted Mail. In *Proceedings of NDSS*. San Diego, California, USA.

[103] Peter Saint-Andre and Jeff Hodges. 2011. Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS). RFC 6125. https://doi.org/10.17487/RFC6125

[104] Quirin Scheitle, Taejoong Chung, Jens Hiller, Oliver Gasser, Johannes Naab, Roland van Rijswijk-Deij, Oliver Hohlfeld, Ralph Holz, Dave Choffnes, Alan Mislove, and Georg Carle. 2018. A First Look at Certification Authority Authorization (CAA). *ACM Computer Communication Review* 48, 2 (April 2018).

[105] Quirin Scheitle, Oliver Gasser, Theodor Nolte, Johanna Amann, Lexi Brent, Georg Carle, Ralph Holz, Thomas C. Schmidt, and Matthias Wählisch. 2018. The Rise of Certificate Transparency and Its Implications on the Internet Ecosystem. In *Proceedings of the ACM Internet Measurement Conference (IMC)* (Boston, MA, USA).

[106] Aaron Schulman, Dave Levin, and Neil Spring. 2014. RevCast: Fast, Private Certificate Revocation Over FM Radio. In *Proceedings of ACM CCS*.

[107] Sudheesh Singanamalla, Esther Han Beol Jang, Richard Anderson, Tadayoshi Kohno, and Kurtis Heimerl. 2020. Accept the Risk and Continue: Measuring the Long Tail of Government https Adoption. In *Proceedings of the ACM Internet Measurement Conference (IMC)*.

[108] Ryan Sleevi. 2020. Path Building vs Path Verifying: The Chain of Pain. Medium. https://medium.com/@sleevi_/path-building-vs-path-verifying-the-chain-of-pain-9fbab861d7d6.

[109] Juraj Somorovsky. 2016. Systematic Fuzzing and Testing of TLS Libraries. In *Proceedings of ACM CCS*.

[110] Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, Dave Andersen, and Jay Lepreau. 1999. The Flask Security Architecture: System Support for Diverse Security Policies. In *8th USENIX Security Symposium (USENIX Security 99)*.

[111] Emily Stark, Lin-Shung Huang, Dinesh Israni, Collin Jackson, and Dan Boneh. 2012. The Case for Prefetching and Prevalidating TLS Server Certificates. In *Proceedings of NDSS*. San Diego, California, USA.

[112] Samuel Steffen, Benjamin Bichsel, Mario Gersbach, Noa Melchior, Petar Tsankov, and Martin Vechev. 2019. zkay: Specifying and Enforcing Data Privacy in Smart Contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1759–1776.

[113] Nick Sullivan. 2017. High-reliability OCSP stapling and why it matters. CloudFlare. https://blog.cloudflare.com/high-reliability-ocsp-stapling/.

[114] Pawel Szalachowski, Stephanos Matsumoto, and Adrian Perrig. 2014. PoliCert: Secure and Flexible TLS Certificate Management.

[115] Emin Topalovic, Brennan Saeta, Lin-Shung Huang, Collin Jackson, and Dan Boneh. 2012. Towards Short-lived Certificates. In *IEEE Web 2.0 Security and Privacy*.

[116] Markus Triska. 2012. The Finite Domain Constraint Solver of SWI-Prolog. In *FLOPS (LNCS, Vol. 7294)*. 307–316.

[117] Jeffrey D. Ullman. 1988. Principles of Database and Knowledge-Base Systems.

[118] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Aastha Mehta, Deepak Garg, Peter Druschel, Rodrigo Rodrigues, Johannes Gehrke, and Ansley Post. 2015. Guardat: Enforcing data policies at the storage layer. In *Proceedings of the Tenth European Conference on Computer Systems*. 1–16.

[119] Narseo Vallina-Rodriguez, Johanna Amann, Christian Kreibich, Nicholas Weaver, and Vern Paxson. 2014. A Tangled Mass: The Android Root Certificate Stores. In *Proceedings of ACM CoNEXT*.

[120] Benjamin VanderSloot, Johanna Amann, Matthew Bernhard, Zakir Durumeric, Michael Bailey, and J. Alex Halderman. 2016. Towards a Complete View of the Certificate Ecosystem. In *Proceedings of the ACM Internet Measurement Conference (IMC)*.

[121] Frank Wang, Ronny Ko, and James Mickens. 2019. Riverbed: Enforcing User-defined Privacy Constraints in Distributed Web Services. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 615–630.

[122] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. 2012. SWI-Prolog. *Theory and Practice of Logic Programming - Prolog Systems archive* 12, 1 (2012), 67–96.

[123] Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. 2016. Precise, Dynamic Information Flow for Database-Backed Applications. *ACM SIGPLAN Notices* 51, 6 (2016), 631–647.

[124] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. 2012. A Language for Automatically Enforcing Privacy Policies. *ACM SIGPLAN Notices* 47, 1 (2012), 85–96.

[125] Jane Yen, Ramesh Govindan, and Barath Raghavan. 2021. Tools for Disambiguating RFCs. In *Proceedings of the Applied Networking Research Workshop*. 85–91.

[126] Scott Yilek, Eric Rescorla, Hovav Shacham, Brandon Enright, and Stefan Savage. 2009. When Private Keys are Public: Results from the 2008 Debian OpenSSL Vulnerability. In *Proceedings of the ACM Internet Measurement Conference (IMC)*.

[127] Liang Zhang, David Choffnes, Tudor Dumitras, Dave Levin, Alan Mislove, Aaron Schulman, and Christo Wilson. 2014. Analysis of SSL Certificate Reissues and Revocations in the Wake of Heartbleed. In *Proceedings of the ACM Internet Measurement Conference (IMC)*.

[128] Liang Zhu, Johanna Amann, and John Heidemann. 2016. Measuring the Latency and Pervasiveness of TLS Certificate Revocation. In *Proceedings of PAM*.

[129] ZLint [n. d.]. ZLint. https://github.com/zmap/zlint.

10 APPENDIX

10.1 List of Browser Edge Cases

While translating the Chrome and Firefox certificate validation logic to Prolog, we noticed several cases where the browsers' behavior deviates from the RFCs in identical ways. Currently we cannot detect these differences with imputation because both browsers' implementations are identical. However, if there was a RFC standard compliant Hammurabi validation policy, then these deviations would be automatically detectable by using imputation to compare Hammurabi-Chrome or Hammurabi-Firefox against the RFC standard policy.

Empty Extended Key Usage. The Extended Key Usage extension (EKU) signifies (along with the Key Usage extension) how a certificate is to be used. RFC 5280 states that "[i]f the extension is present, then the certificate MUST be only be used for one the purposes indicated." However, it does not specify what to do when the EKU extension is present but empty. Both Chrome and Firefox interpret an empty EKU extension by permitting all required EKUs for the given certificate.

Signature Algorithm Equality. For historical reasons (defense against the now irrelevant algorithm substitution attack), an X.509 consists of two signature algorithm fields: one in the "outer" `certificate` structure, called the `signatureAlgorithm`, and one in the "inner" `tbsCertificate`, called the `signature`. RFC 5280 states that these two identifiers MUST be equivalent, but does not specify what equivalent means—either byte-for-byte encoded equality or decoded semantic equality. Chrome and Firefox prefer byte-for-byte equality but accept the same signature algorithms with different encodings.