Towards Practical, End-to-End Formally Verified X.509 Certificate Validators with Verdict

Zhengyao Lin[†] Michael McLoughlin[†] Pratap Singh[†] Rory Brennan-Jones[‡]
Paul Hitchcox[†] Joshua Gancher[§] Bryan Parno[†]

[†]*Carnegie Mellon University* [‡]*University of Rochester* [§]*Northeastern University*

Abstract

Validating X.509 certificates is a critical part of Internet security, but the relevant standards are complex and ambiguous. Most X.509 validators also intentionally deviate from these standards in idiosyncratic ways, often for security or backward compatibility. Unsurprisingly, the result is a long history of security vulnerabilities.

In this work, we present Verdict, the first end-to-end formally verified X.509 certificate validator with customizable validation policies. Verdict's formal guarantees cover certificate parsing, path building, and path validation. To make Verdict practical to both verify and to use, we specify its correctness generically in terms of a user-supplied validation policy written concisely in first-order logic, with a proof-producing compiler to efficient Rust code.

To demonstrate Verdict's expressiveness, we use Verdict's policy framework to implement the X.509 validation policies found in Google Chrome, Mozilla Firefox, and OpenSSL, and formally prove that they conform to a subset of RFC requirements. We instantiate Verdict with each policy and show that Verdict matches the corresponding baseline's behavior and state-of-the-art performance on over ten million certificates from Certificate Transparency logs.

1 Introduction

The X.509 standard [11] underpins the security of many ubiquitous PKIs, including those used for HTTPS, Wi-Fi, IPSec [29], code signing [31], and secure boot [58]. X.509 validators have been described as "the most dangerous code in the world" [23], since mistakes can lead to far-reaching security breaches.

Unfortunately, X.509 validation is very complex to implement. The core English-level specifications span at least five different, occasionally ambiguous [17] RFCs [8,11,35,55,56]. On top of the RFCs, a web browser's certificate validator (arguably one of the most critical users of the X.509 PKI) must also comply with 159 pages of CA/B Baseline Requirements (BRs) [20] which extend and in some cases overrule the RFCs. As a final complication, each X.509 validator can and does make its own individual policy decisions, both in terms of which optional features to support, and also when to deliberately violate the standards for security or backward compatibility. For instance, to improve compatibility, Firefox violates RFC 5280 by allowing leaf certificates with a `keyCertSign` key usage [32]. Making matters worse, most modern X.509 validators deeply entangle the code that implements policy logic with code implementing parsing and cryptographic operations [32].

With all of this complexity, it is unsurprising that the world of X.509 validation has a long history of security vulnerabilities [4,12,62] and outages [2]. Some of these bugs have been found via fuzzing [4,7,45] and symbolic execution [6]. While useful, these ad hoc approaches do not rule out the presence of additional bugs.

In principle, formal software verification can prevent large classes of bugs by mechanically proving that an X.509 validator's code correctly implements an abstract specification of X.509 certificate path validation. In practice, formally verifying X.509 validation poses unique challenges, and prior attempts have been both incomplete and impractical. First, for formal verification results to be meaningful, we typically expect the spec to be smaller and simpler than the implementation; otherwise, the spec may be just as buggy (or buggier!). As discussed above, however, the "obvious" spec found in the RFCs and BRs is large and complex, and there is effectively no one true spec. As a result, prior work [17,18], which tried to formalize X.509 validation based only on a subset of one RFC, is both overly strict (ruling out legitimate policy choices) and incomplete (missing requirements from the other RFCs and all of the BRs). Second, prior work made a number of design decisions that result in impractical performance, running one [17] to two [18] orders of magnitude slower than unverified baselines and consuming three orders of magnitude more memory [18].

With Verdict, we show how to design a practical, end-to-end formally verified X.509 validator. Our key idea is to design Verdict as a formally verified *policy framework* that decouples the verification of a user-supplied policy from the rest of the validator. We allow users to specify their own policies in a high-level, readable domain-specific language (DSL), which can then be **(1)** automatically compiled into provably correct executable code (§4.3), and **(2)** used for formal RFC/BR conformance checking (§5.1).

Combined with the formal verification of the rest of the validator (§4), we are able to specify and prove the *end-to-end* correctness of the entire validator generically in terms of a user-provided policy. This makes our top-level theorem clean and succinct, abstracting away the complexities of the RFCs, BRs, and individual policy choices. At the same time, it acknowledges the reality that there is no one true spec by allowing different users to choose their own policies and providing formal guarantees about the policies.

While Verdict provides a clean, end-to-end verification result, internally it includes a number of interesting components. In addition to the policy framework, it includes an X.509 certificate parser and serializer, with proofs of soundness, completeness, and non-malleability. It also includes a verified path builder, which constructs candidate paths of issuing certificates between the offered leaf and a trusted root, with proofs of soundness and completeness, ruling out issues that historically caused large-scale outages [2].

As another step towards practicality, we explicitly design Verdict with performance in mind. Verdict is implemented in performant Rust code and verified using Verus [33, 34], an automated deductive program verifier. From user-defined policies, we automatically derive efficient Rust code with generated proofs of correctness in Verus. Our verified X.509 and ASN.1 parser (§4.1) is designed to be mostly zero-copy and achieves much better performance than prior work [18, 41]. Through careful design and engineering, Verdict adds little overhead on top of cryptographic providers, achieving state-of-the-art performance when combined with verified libraries such as EverCrypt [47] and AWS-LC [57].

We evaluate Verdict's practicality in §5 and §6. First, to demonstrate the expressiveness of Verdict's policy language, we use it to encode the certificate validation policies used in Google Chromium [9] (Chrome), Mozilla Firefox [19], and OpenSSL [44]. We validate their consistency on over ten million certificates from the wild, as well as an existing test suite designed to probe gnarly edge cases in X.509 validation. Second, we show that Verdict's performance is competitive with Chrome, Firefox, and OpenSSL, and it outperforms prior academic work by *orders of magnitude*. We also integrate Verdict into an existing TLS library, Rustls [54], and show that using Verdict in a practical setting incurs negligible overhead.

In summary, we make the following contributions.

1. Verdict, the first end-to-end formally verified X.509 certificate validator, offering both practical flexibility, and practical performance.

2. A verified policy framework for expressing (in a succinct, readable fashion) user-defined X.509 validation policies, which can be compiled into efficient, provably correct executable code, and analyzed formally, e.g., to verify RFC/BR conformance.

3. Formal models of X.509 validation policies in Chrome, Firefox, and OpenSSL in our policy DSL.

4. An extensive evaluation showing that Verdict's performance compares favorably to unverified real-world implementations, and its policy DSL offers the expressivity to capture some of the most complex validation policies used in the wild.

2 Preliminaries

In this section, we provide preliminary background for X.509 certificate validation and the Rust verification language Verus.

2.1 X.509 Certificate Validation

From its humble origins in 1988 [59], the X.509 standard has grown to span multiple RFCs [11, 35, 55, 56] specifying how to parse and validate certificates. For the web's PKI, validators must also conform to a long list of CA/Browser Forum Baseline Requirements (CA/B BRs) [20]. In this section, we primarily focus on the use of X.509 in the web's PKI, but other settings are similar.

An X.509 certificate is a data structure containing four main pieces of information: **(1)** the issuer's identity, **(2)** the subject's identity, **(3)** the subject's public key, and **(4)** the issuer's signature on the entire certificate. For backward compatibility, RFC 5280 adds new features to X.509 via a list of *extensions* attached to the end of the certificate. Most extensions describe additional information for issuer/subject identities (e.g., hostnames are usually kept in the Subject Alternative Name extension), or constraints on how the certificate should be used (e.g., Key Usage and Name Constraints).

Abstractly, an X.509 certificate is its issuer's attestation (via a cryptographic signature) that the subject's public key binds to the subject's identity. Therefore, if one trusts the issuer, then they can also trust that the subject's public key belongs to the subject of the certificate. This trust relation is transitive: if we have a sequence of certificates $C_1, \ldots, C_n$, such that each C_i is issued by C_{i+1} (i.e., C_i contains a signature that is verifiable with the subject public key of C_{i+1}), then if we trust the issuer of C_n, we can also trust the subject of C_1.

Concretely, consider the example of a web client that wants to determine if it can trust a server. The client locally stores a set of trusted *root certificates*, and the server sends a list of certificates, called a *certificate chain*, that includes the server's *leaf certificate* (which binds the server's public key to the hostname being accessed), and an optional set of *intermediate certificates*. The client then tries to find a path (aka *path building* or *chain building*): a sequence $C_1, \ldots, C_n$ of certificates with each issued by the next, such that C_1 is the leaf certificate, and C_n is a root certificate that the client trusts. Then by the transitivity of trust, the client can trust that the public key of C_1 is legitimate, and further secure communication can happen using that public key.

Although most X.509 validators follow this high-level path validation process, the exact rules and checks differ substantially [18, 32]. This can be both due to compatibility and performance reasons, or ambiguous requirements in the standard. For example, when comparing the issuer name of a certificate against the subject name of the parent certificate, Chrome, Firefox, and OpenSSL use different string comparison procedures. Chrome normalizes strings by removing leading/trailing ASCII spaces, case lowering, and compressing multiple ASCII spaces into one; Firefox does not perform any normalization; and OpenSSL is similar to Chrome, except that it considers more characters as white spaces.

X.509 validators also differ in what fields in an X.509 certificate are considered the subject's identity. For instance, Chrome only checks the accessed hostname against the Subject Alternative Name extension, while Firefox falls back to the Common Names in the Subject Distinguished Name if the Subject Alternative Name is empty.

Correctly conforming to the standard is also challenging. For example, although the specification of path building is straightforward (i.e., there exists a valid issuing path), various incorrect implementations have failed to explore all possible paths (e.g., early versions of OpenSSL [44] and GnuTLS [24]), which resulted in a large-scale outage in 2020 [2].

2.2 Formally Verifying Rust Code with Verus

Verdict is implemented in Verus [33, 34], which is an semi-automated program verifier based on the Rust programming language [50]. Verus extends Rust with formal verification capabilities for Hoare-style reasoning [25], allowing users to prove formal properties about their programs, while utilizing the native performance and extensive ecosystem of Rust.

In this section, we provide a brief overview of Verus. We assume some familiarity with the Rust language, and refer readers to the Rust book [30] for more details.

In Verus, users write their code in (a large subset of) Rust and then add annotations that provide mathematical *specifications* and *proof hints* to verify properties about their Rust program. Figure 1 shows a toy example that verifies the functional correctness of an executable function against a specification. This example has two functions, a *specification function* (spec function) called `spec_min_index` and an *executable function* (exec function) called `min_index`. The spec function takes an immutable sequence `s` of type `Seq<u64>` and an unbounded integer index `i`, and defines when the index stores a minimum element of the sequence `s`. A spec function in Verus cannot be compiled or executed, but it serves a concise, abstract definition for static reasoning. Therefore, one can use mathematical *spec types* in a spec function, such as the mathematical, immutable sequence type `Seq`, and the unbounded integer type `int`.

```
spec fn spec_min_index(s: Seq<u64>, i: int)
-> bool { 0 ≤ i < s.len() ∧
          forall |x| x ∈ s ⇒ s[i] ≤ x }

fn min_index(s: &[u64]) -> (res: usize)
  requires s.len() > 0
  ensures spec_min_index(s.view(), res)
{ let mut min = 0;
  for i in 0..s.len()
    invariant spec_min_index(s[..i+1], min)
  { ... }
  ...
```

Figure 1: Example Verus code.

The exec function `min_index`, on the other hand, is a normal Rust function that can be compiled. In addition to the Rust syntax, it is annotated with a precondition (`requires`) and a postcondition (`ensures`), stating a specification for the exec function: for any non-empty slice `s`, the function should correctly output an index with a minimum element. Verus then semi-automatically verifies that this function conforms to the specification, with the help of some extra proof hints such as loop invariants (`invariant`).

In general, the flavor of verification in Verus is similar to this simple example: we write concise, trusted, and purely mathematical specifications of the intended behavior or properties of a program, and then prove that the executable implementation is either equivalent to, or a refinement of, the specification. Along the way, Verus also verifies basic safety properties of the executable code, such as the absence of integer overflow and panics.

3 Threat Model and TCB

Verdict is a formally verified X.509 certificate validator, which is expected to handle potentially malicious leaf and intermediate certificates sent by the peer, e.g., a web server during a TLS handshake. However, we assume that there is no malicious actor locally, which includes assuming that the root certificates provided by the user are issued by trusted CAs.

We verify Verdict against an end-to-end specification (§4) that can rule out a wide range of vulnerabilities in our parser/serializer, path builder, and the user-provided policy; but we do assume that our verification tooling and specifications are correct. That is, we need to trust the end-to-end specification itself (including the user-provided policy encoded in our policy DSL), the Verus program verifier, the Rust compiler, and our glue code calling external verified cryptographic libraries. We only use the safe fragment of Rust (enforced via the `#![forbid(unsafe_code)]` attribute), except for the trusted glue code.

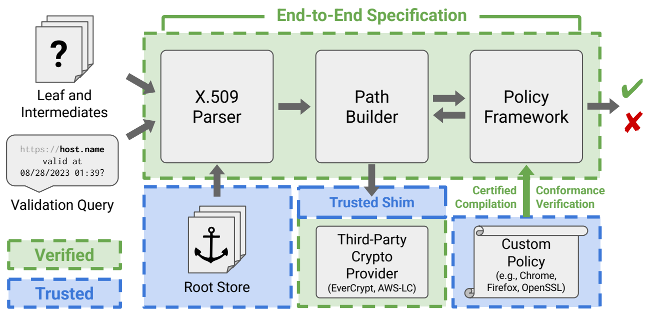

Figure 2: High-level design of Verdict.

```
spec fn spec_valid_x509(query: Query,
  policy: Policy, leaf: Seq<u8>,
  ints: Seq<Seq<u8>>, roots: Seq<Seq<u8>>)
-> bool {
  // Parse input certificates
  let leaf = spec_parse_x509(leaf);
  let ints = ints.map(spec_parse_x509);
  let roots = roots.map(spec_parse_x509);
  // Exists a valid simple path
  exists |path: Seq<Certificate>|
    // The path obeys the issuing relation
    path[0] == leaf ∧ path.last() ∈ roots
    ∧ forall |i| 0 < i < path.len() - 1
        ⇒ path[i] ∈ ints
    ∧ forall |i| 0 ≤ i < path.len() - 1
        ⇒ spec_issued(policy,
            path[i + 1], path[i])
    // No cycles
    ∧ forall |i, j| 0 ≤ i < j < path.len()
        ⇒ path[i] ≠ path[j]
    // Policy evaluation
    ∧ policy.valid_path(...) }
spec fn spec_issued(policy: Policy,
  issuer: Certificate,
  subject: Certificate) -> bool {
  policy.issued(issuer, subject)
  ∧ verify_signature(issuer, subject) }
spec fn spec_parse_x509(Seq<u8>)
  -> Option<Certificate> { ... }
```

Figure 3: End-to-end specification of X.509 validation in Verdict, with some error handling details omitted. Shorthand `xs.map(f)` stands for applying `f` to each element of `xs`.

```
struct Certificate {
  version, not_after, not_before: u64,
  issuer, subject: DistinguishedName,
  subject_key: SubjectKey,
  subject_alt_name: Option<SubjectAltName>,
  key_usage: Option<KeyUsage>, ... }
```

Figure 4: A fragment of the definition of our intermediate representation of X.509 certificates.

4 A Formally Verified X.509 Validator

In this section, we present the design of Verdict, its end-to-end specification, and how we verify its main components.

At a high-level (Figure 2), Verdict takes as input a leaf certificate, intermediate certificates, trusted root certificates, and a validation query (e.g., hostname, validation time, and authentication purpose). These certificates are then passed through a pipeline of three *formally verified* components.

- The *X.509 parser* (§4.1) transforms DER [60]-encoded certificates into an intermediate representation;
- The *path builder* (§4.2) iterates through all candidate paths from the leaf certificate to a trusted root certificate, by verifying signatures of certificates and invoking a policy-specific definition of the issuing relation.
- The *policy framework* (§4.3) evaluates a user-defined validation policy in our policy DSL on candidate paths from the path builder. The policy itself is trusted, but we use certified compilation and conformance verification (§5) for better formal guarantees.

These components are all implemented in Verus, and they are formally verified to conform to a concise, *end-to-end* specification of the entire X.509 validation process.

A simplified version of the end-to-end specification is shown in Figure 3. The main specification (`spec_valid_x509`) takes in a validation query, a user-defined policy, and Base64-encoded leaf, intermediate and root certificates (each certificate is represented as a mathematical sequence of bytes with type `Seq<u8>`). It then states that the query is valid with respect to the given set of certificates, if and only if there exists a valid path from the leaf certificate to one of the root certificates. Namely, a valid path should (1) follow the issuing relation `spec_issued`, which checks the signature and a policy-specific name comparison scheme (`policy.issued`), (2) have no cycles, and (3) be considered valid by the user-defined policy (`policy.valid_path`).

We discuss details of the parser specification `spec_parse_x509` in §4.1. The parsed intermediate representation of certificates (`Certificate`) acts as the main interface throughout the specification and implementation of Verdict. Figure 4 shows a small fragment of its definition, which includes fields in an X.509 certificate (e.g., its issuer, subject, expiration date, etc.), and supported extensions (e.g., Subject Alternative Name, Key Usage, etc.).

In addition to proving that our implementation is equivalent to this end-to-end specification, we verify a variety of other formal properties. First, we ensure basic safety properties including memory safety and the absence of panics and integer overflows. Each component is also verified to satisfy a number of functional-correctness and security properties:

- *Parser non-malleability*: no two different byte sequences can parse to the same intermediate representation.

- *Parser prefix security*: appending bytes to a valid byte sequence would not produce a valid result.
- *Parser soundness and completeness*: the parser succeeds if and only if the input byte sequence is valid according to a formal specification derived from RFC 5280.
- *Path builder soundness and completeness*: the path builder iterates through exactly the simple paths from the leaf certificates to the root certificates.
- *Policy functional correctness*: a policy spec in our DSL is compiled to a provably equivalent executable version.

These properties ensure the correctness of the implementation and give us more confidence in the correctness of the specification. For example, parser non-malleability and prefix security can act as sanity checks for whether we have correctly translated the text of RFC 5280 and ASN.1 specifications to to a formal specification, since they claim to be non-malleable on paper. This formally prevents some actual bugs we found in popular X.509-related libraries (§6.4).

We now discuss the verification of each component.

4.1 X.509 Parser and Serializer

X.509 syntax is a complex beast by itself. In RFC 5280, the encoding of an X.509 certificate is specified using ASN.1 DER [60], which is a standard commonly used in networking and cryptography for the serialization of data structures. RFC 5280 contains over 250 ASN.1 definitions, and the ASN.1 DER specification itself [60] is a 36-page document.

To tackle the complexity of X.509 syntax while achieving good performance, we utilize two main ideas: (1) modular parsing via verified parser combinators; (2) automated generation of verified parsers from concise specification macros.

Verified Parser Combinators. A parser combinator [22], in general, is a function that takes in various parameters (which can be parser combinators themselves), and returns a new parser. In our work, we use a verified parser combinator library called Vest [5], which is also implemented in Verus and provides a common interface for parser combinators.

In Vest, every parser combinator implements the `Combinator` trait shown in Figure 5. It requires definitions of parsing and serializing at the spec level (`spec_parse` and `spec_serialize`) as well as at the executable level (`parse` and `serialize`). At the spec level, we use immutable mathematical objects such as immutable sequences of bytes (`Seq<u8>`), and the definitions are concise and functional; whereas the executable versions are imperative Rust programs using standard data types, such as slices of bytes. The post-conditions of `parse` and `serialize` state that they are equivalent to the spec versions.

In addition to proving the equivalence, one also needs to prove two properties: (1) parsing and serializing are partial inverses of each other (i.e., `prop_inverses`, where the `?` is a shorthand denoting that the property only needs to hold if both parsing and serializing are successful); (2) parsing is *prefix-secure*, that is, appending more bytes to the input would not change a successful parsing result. In particular, (1) implies that the parser is non-malleable.

```
trait Combinator { type R;
  // Spec parser/serializer
  spec fn spec_parse(Seq<u8>) -> Option<R>;
  spec fn spec_serialize(R)
    -> Option<Seq<u8>>;
  // Executable parser/serializer
  fn parse(...)
    ensures parse = spec_parse;
  fn serialize(...)
    ensures serialize = spec_serialize;
  // Properties required for each combinator
  proof fn prop_inverses(r: R, b: Seq<u8>)
    ensures
      spec_parse(spec_serialize(r)?)? = r,
      spec_serialize(spec_parse(b)?)? = b;
  proof fn prop_prefix_secure(b: Seq<u8>)
    requires spec_parse(b)? ≠ None
    ensures forall |b2: Seq<u8>|
      spec_parse(b + b2) = spec_parse(b); }
```

Figure 5: The combinator trait in Vest [5], where the associated type `R` is the type of the parsing result.

Following this parser/serializer specification, we implement and verify 20 parser combinators for ASN.1 DER, including base types (integer, object identifier, various string types, UTC time, etc.), structures (sequence and union), and tagging mechanisms, and prove that they satisfy the partial-inverse and prefix-secure properties. For a more end-to-end guarantee, we also implement a verified combinator for Base64 [27].

These parser combinators are non-trivial as ASN.1 DER uses space quite efficiently. For example, an ASN.1 object identifier is a sequence of integers, and in DER, each such integer is encoded as a variable-length sequence of bytes, with the lower 7 bits from the integer itself, and the highest bit indicating whether it is the last byte. In fact, during differential testing of our verified object identifier parser combinator against a popular Rust library, we found that their object identifier encoding is incorrect in certain ranges (§6.4).

A Combinator DSL for ASN.1. Using the ASN.1 primitives, we designed a domain-specification language (DSL) to conveniently derive more verified ASN.1 combinators. In the DSL, one only needs to define high-level fields and structures, similar to an actual ASN.1 schema, and the verified parser combinators are automatically derived via a Rust macro.

For instance, the main ASN.1 definition in X.509 is `TBSCertificate` (i.e., "to-be-signed certificate"), which contains fields for the version, issuer, subject, subject public key, etc., of a certificate. In RFC 5280, this is defined exactly as:

```
TBSCertificate ::= SEQUENCE {
  version [0] EXPLICIT Integer DEFAULT 0,
  serial  Integer, ...
  sUID [2] IMPLICIT BIT STRING OPTIONAL,
  exts [3] EXPLICIT Extensions OPTIONAL }
```

The keyword SEQUENCE in ASN.1 defines a format similar to a struct in Rust or C. EXPLICIT and IMPLICIT are *tagging* schemes. In ASN.1 DER, all objects are encoded in a tag-length-value (TLV) tuple, where a tag is a variable-length integer that indicates the type of the object. In order to unambiguously parse optional fields, [*n*]EXPLICIT indicates that an extra TLV tuple should be wrapped around the underlying field with tag *n*, and [*n*]IMPLICIT indicates that the tag of the underlying field should be replaced with *n*.

In our DSL, TBSCertificate is specified similarly:

```
asn1!(seq TBSCertificate {
  version: Default(0, Explicit(0, Integer)),
  serial:  Integer, ...
  suid: Optional(Implicit(2, BitString)),
  exts: Optional(Explicit(3, Extensions)) })
```

where the highlighted functions are our verified parser combinators for ASN.1 primitives. We use the Rust macro asn1! to compile our DSL definition into a Verus definition of a new verified combinator TBSCertificate.

Using this DSL, we implemented 26 combinators for X.509 certificates, including 9 commonly used X.509 extensions: Authority Key Identifier, Subject Key Identifier, Basic Constraints, Certificate Policies, Authority Information Access, Key Usage, Extended Key Usage, Subject Alternative Name, and Name Constraints.

Although not required in Verdict, the ASN.1 DER and X.509 combinators also have verified serializers, and they are designed to be reusable in other Rust or Verus projects.

Top-Level Specifications and Properties. Using the manually verified and automatically derived combinators, we implement an executable parser for X.509 certificates from Base64, with formally verified properties:

```
fn parse_x509(b: &[u8])
  -> (r: Option<Certificate>)
ensures r matches Ok(r) ⇒ (
    // Soundness
    spec_parse_x509(b) = Some(r)
    // Non-malleability
    ∧ forall |b2: Seq<u8>|
        spec_parse_x509(b2) = Some(r)
        ⇒ b2 = b
    // Prefix Security
    ∧ forall |b2: Seq<u8>| b2.len() ≠ 0
        ⇒ spec_parse_x509(b + b2) is None),
  // Completeness
  r is Err ⇒ spec_parse_x509(b) is None;
```

Here spec_parse_x509 (Line 28 of Figure 3) is the top-level specification for X.509 parsing, which calls spec_parse of the top-level combinator for X.509 certificates. The implementation parse_x509 calls the corresponding executable parse function of the combinator. Then using the properties of the Combinator trait, we verify that the parser implementation is sound, complete, non-malleable, and prefix-secure.

4.2 Path Builder

Path building is a crucial and error-prone process in any X.509 validator. A naive validator simply checks the chain sent by the server. Unfortunately, such a validator will miss any alternative paths (caused by "cross-signing") from the given leaf to a trusted root, which may be in place to ensure backwards compatibility. Therefore, an X.509 validator should search for all possible paths from the leaf to one trusted root, through any combination of intermediates, until a valid path is found. Failure to do so has led to large-scale outages [2].

In many X.509 validators, path building is usually deeply intertwined with the validation policy [32]. In Verdict, however, we only check for a minimal set of requirements that are common in X.509 clients, and defer most of the path validation to the user-provided validation policy to ensure flexibility. We specify that a certificate c is likely issued by another certificate c' (Line 23 in Figure 3) if and only if: (1) The validation policy considers c' an issuer of c (by checking, e.g., if the issuer Distinguished Name of c matches the subject Distinguished Name of c'); and (2) the signature of c is valid with respect to the public key of c'.

In the specification of the path builder (Line 9 in Figure 3), we state that the valid path should follow this basic issuing relation, from the leaf to a root certificate. In addition, it should satisfy the user-provided policy, which includes more restrictions such as checking if the leaf certificate actually binds the desired hostname, or if all certificates are marked with suitable usage flags (e.g., Key Usage). This specification is then used in the postcondition of the executable path builder:

```
fn build_path(query: &Query,
  policy: &Policy, leaf: &Certificate,
  ints: &Vec<Certificate>,
  roots: &Vec<Certificate>) -> (res: bool)
ensures res ⟺ // See Line 9 of Figure 3
  exists |path: Seq<Certificate>| ...
```

In the implementation, we perform a depth-first traversal of all simple paths from the leaf to any root certificate. For signature checking, we invoke existing formally verified cryptographic providers: EverCrypt [47] via libcrux [37] for verifying ECDSA P-256 [43] and RSA PKCS #1 v1.5 [28] signatures; and AWS-LC [57] for verifying ECDSA P-384 [43] signatures. We formally verify that the build_path is equivalent to the specification, implying that it only considers valid paths (sound) and always finds a valid path if it exists (complete).

```
struct OpenSSLPolicy;

valid_path(policy: OpenSSLPolicy,
  query: Query, path: Seq<Certificate>)
  = path.len() ≥ 2
  ∧ valid_leaf(policy, query, path[0])
  ∧ forall |i| 1 ≤ i < path.len() - 1
    ⇒ valid_intermediate(policy, query, path[i])
  ∧ valid_root(policy, query, path.last())
  ∧ check_name_constraints(path)

valid_leaf(policy: OpenSSLPolicy,
  query: Query, cert: Certificate)
  = check_cert_key_level(cert)
  ∧ check_cert_time(policy, cert)
  ∧ check_purpose_leaf(cert)
  ∧ cert.sig_alg_inner.bytes
    = cert.sig_alg_outer.bytes
...
```

Figure 6: A snippet of the OpenSSL policy in Verdict's DSL.

4.3 Policy Framework and DSL

Verdict's customizability lies in its policy framework, which allows users to write concise logical specifications of their validation policy, and then automatically derives an efficient, verified implementation for execution.

To define a policy in Verdict, one specifies two predicates:

```
trait Policy {
  spec fn issued(self,
    Certificate, Certificate) -> bool;
  spec fn valid_path(self,
    Query, Seq<Certificate>) -> bool;
}
```

The first spec method `issued` defines the issuing relation between certificates, which usually includes checking distinguished names and/or the Authority Key Identifier and Subject Key Identifier extensions. This method is used in the path builder (§4.2) to define a candidate path.

The second spec method `valid_path` defines additional constraints on the candidate path found by the path builder, which might include more policy-specific checks such as hostname validation, expiration check, key usage check, etc.

To define these two predicates, we can use Verdict's policy DSL that provides a first-order logic in a Rust-like syntax to describe high-level constraints imposed by the policy. The policy DSL supports: (1) basic types, including integers, strings, sequences, and user-defined structs and enums; (2) basic integer/Boolean arithmetic, sequence operations, branching, pattern matching; and (3) universal and existential quantifiers over integers guarded by explicit ranges.

For example, Figure 6 shows a snippet of the OpenSSL policy formalized in Verdict, defining the `valid_path` predicate. At the top-level, the `valid_path` predicate takes a validation query (e.g., hostname, validation time, etc.) and a candidate path (where the first element is the leaf, the last element is the root, and the others are intermediates). It specifies that a path is valid if and only if each certificate on the path is valid, and the Name Constraints extension is satisfied.

At each certificate, the OpenSSL policy also specifies a number of checks. For example, `check_cert_key_level` checks that the public key of the certificate provides a sufficient level of security; `check_cert_time` checks that the certificate has not expired; `check_purpose_leaf` checks that the certificate can be used as a leaf by checking Basic Constraints, Key Usage, and Extended Key Usage extensions. In total, the OpenSSL policy is formalized in less than 400 lines of specification (see §5 for more detail).

All predicates defined in the policy DSL are automatically compiled (via a procedural macro [30]) to two pieces of code: (1) a spec function in Verus, and (2) an executable function with a Verus proof that it is equivalent to the spec function. For example, the `valid_path` predicate in Figure 6 is compiled to an executable function with the following specification:

```
fn exec_valid_path(env: &OpenSSLPolicy,
  path: &Vec<Certificate>, ...) -> (r: bool)
ensures r = valid_path(env, path, ...)
```

In the body of the function, all spec operations are compiled to the corresponding executable operations, and all bounded universal and existential quantifiers are compiled to loops with suitable invariants and proofs. At compilation time, Verus verifies the functional correctness of the executable function, i.e., equivalence to the original high-level specification.

5 Case Study: Chrome, Firefox, and OpenSSL

Using the Verdict policy framework, we formalized the X.509 validation policies used in Google Chrome [9], Mozilla Firefox [19], and OpenSSL [44].[1] In this section, we discuss how we formalize them, key advantages of using Verdict, and the results of proving the RFC conformance of these policies.

For Chrome and Firefox, we based our policies on those from Hammurabi [32], which defined Chrome and Firefox policies in Prolog [10]. In the first iteration, we translated Hammurabi's Prolog policies to our policy DSL. The translation is straightforward, as the Prolog versions do not use any backtracking other than during the path building process. We subsequently fixed a number of inconsistencies we found during differential testing against the original Chrome and Firefox implementations. For OpenSSL, we modeled our policy directly on the source code (with strict mode enabled).

Since a policy definition is part of the TCB of an X.509 validator, it needs to be high-level and easy to understand.

[1]For Chrome, we are modeling `CertVerifyProc::Verify` in `netcertcert_verify_proc.cc` at commit `0590dcf` (Aug, 2020); for Firefox, we are using `CertVerifier::VerifySSLServerCert` in `security/certverifier/CertVerifier.cpp` at (Mercurial) changeset `dbd5ee7` (Aug, 2020); for OpenSSL, we are using `X509_verify_cert` in `crypto/x509/x509_vfy.c` at commit `5c5b8d2` (Nov, 2024).

Our policy DSL allows a concise encoding comparable to the Prolog formulation in Hammurabi, with additional advantages such as type safety and performance (§6.1). In our policy DSL, the Chrome/Firefox/OpenSSL policies are encoded in respectively 405/400/372 lines of code, in addition to 325 lines of common definitions (e.g., the definition of `Certificate`). In comparison, Hammurabi's Prolog encodings have 343 lines of code for Chrome, 418 lines for Firefox, and 58 lines of common utilities.

Our formulation is also significantly more concise than the original implementations. Comparing lines of code directly is difficult, as the original implementations include additional features such as telemetry and error handling that are not formalized in Verdict (see §6.1). However, as some coarse measures, for their main path building and validation logic, Chrome has 1,628 lines of C++,[2] Firefox has 2,348 lines of C++,[3] and OpenSSL has 2,429 lines of C.[4]

5.1 Verifying RFC Conformance

Our policy framework provides a great deal of freedom in defining the validation policy, as users can write their policy as an arbitrary first-order predicate in our policy DSL. In many cases, however, the user may want to ensure that the policy they have defined conforms to various X.509 standards, such as the RFC 5280 [11] and CA/B BRs [20]. Therefore, Verdict also includes a lightweight, semi-automated mechanism to prove that a policy satisfies a set of RFC requirements.

By carefully translating the text of the RFC 5280 and CA/B BRs, we defined a number of conformance properties as additional traits on a `Policy` (§4.3), such as the following.

```
trait NonLeafMustBeCA: Policy {
  proof fn conformance(self, query, path)
    requires self.valid_path(query, path)
    ensures forall |i| 1 ≤ i < path.len()
      ⇒ path[i].basic_constraints.is_ca; }
```

This particular rule states that in a valid path, the policy should ensure that all intermediate and root certificates have a "CA" flag in the Basic Constraints extension (i.e., only certificates marked as belonging to a CA can issue certificates). Then to prove this RFC requirement on a concrete policy specified in Verdict, such as the OpenSSL policy, we can state:

```
impl NonLeafMustBeCA for OpenSSLPolicy {
  proof fn conformance(self, query, path) {}
}
```

This verifies the postcondition in `NonLeafMustBeCA` against the concrete definition of `OpenSSLPolicy::valid_path` in our DSL (Figure 6). In this case, Verus is able to automatically discharge the proof obligations; but in general, users may need to add some additional proof hints to prove the conformance.

In Verdict, we encode a total of 20 requirements from RFC 5280 and CA/B BRs. We attempted to verify the requirements against our Chrome, Firefox, and OpenSSL policies, and found that they conform to 8, 10, and 11 of these requirements, respectively. All of the conforming properties are verified by Verus automatically without requiring proof hints.

Below, we discuss some of the differences and non-conformance we found in these policies. While most of them are not security-critical, they demonstrate a wide range of different policy choices and show the necessity of allowing user-defined policies. Furthermore, they also show that formal verification has a beneficial side effect of uncovering these nuances in the specification.

Expiration Check. One may think that at least the expiration check should be a universally agreed upon policy among all X.509 implementations. However, we found that Chrome does *not* check the expiration of a trusted root certificate, while Firefox and OpenSSL do. We discovered this difference when attempting to verify the following property:

```
pub trait NoExpiration: Policy {
  proof fn conformance(self, query, path)
    requires self.valid_path(query, path)
    ensures forall |i| 0 ≤ i < path.len()
      ⇒ path[i].not_before
        ≤ query.validation_time
        ≤ path[i].not_after; }
```

Skipping the expiration check may either be done for performance, or as a result of the ambiguity in the suggested path validation algorithm in RFC 5280 Section 6.1: it only checks that "the certificate validity period includes the current time" for all certificates *not* including the trusted anchor. Chrome's path builder seems to closely follow this algorithm.

Another minor detail in RFC 5280 is that the expiration check is inclusive; i.e., the certificate is valid only if `notBefore` $\leq t \leq$ `notAfter`, where `notBefore` and `notAfter` are two fields in an X.509 certificate. We found that Chrome and Firefox conform to this requirement, but OpenSSL checks `notBefore` $\leq t <$ `notAfter` instead.

Root CA Flag. For the same reason as the previous difference, Chrome does not require that a root certificate has the CA flag in the Basic Constraints extension. We found this discrepancy when attempting to verify the `NonLeafMustBeCA` property above. On the other hand, Firefox and OpenSSL policies do satisfy this property.

Extended Key Usage. According to CA/B BRs, a leaf certificate must contain the Extended Key Usage extension with purposes including server authentication, and a root certificate must not have the Extended Key Usage extension. However,

[2] `net/cert/{cert_verify_proc,cert_verify_proc_builtin}.cc, net/cert/internal/path_builder.cc`

[3] `security/.../{CertVerifier,NSSCertDBTrustDomain}.cpp`

[4] `crypto/x509/x509_vfy.c`

we found that none of Chrome, Firefox, and OpenSSL enforce these two requirements.

Deprecated DSA Signatures. DSA signatures are not supported by CA/B BRs. While Chrome and Firefox both reject certificates with DSA signatures (and leaf certificates with a DSA public key), OpenSSL does not enforce this policy by default, and requires extra compiler flags to disable DSA.

Root Authority Key Identifier. CA/B BRs require that if a root certificate contains the Authority Key Identifier extension, it must only have the `keyIdentifier` field, but not the optional fields of `authorityCertIssuer` and `authorityCertSerialNumber`. None of Chrome, Firefox, and OpenSSL conform to this requirement. Firefox and OpenSSL do not check for the absence of the latter two fields; Chrome only imposes the RFC 5280 requirement that the latter two fields must be both present or both absent.

6 Evaluation

In this section, we compare Verdict against existing implementations of X.509 validation in the wild, to answer the following three questions:

Q1 Does Verdict have competitive performance against other X.509 implementations? (§6.1)

Q2 Do the formalized Chrome, Firefox, and OpenSSL policies match the original implementations? (§6.2)

Q3 Can Verdict be easily integrated in third-party tools and used in practical application settings? (§6.3)

In addition to answering these three questions, we also discuss (§6.4) some of the security issues that Verdict's verification categorically eliminates but that frequently occur in existing validators (indeed we found new examples while developing Verdict!). Finally, we summarize key verification challenges we encountered and our lessons learned (§6.5).

Our implementation of Verdict contains about 13,000 lines of verified proof/code (written in Verus) and 3,200 lines of unverified Rust code, of which 79% does not need to be trusted (the policy DSL compiler and testing infrastructure), and the rest (glue code for the crypto providers) is trusted.

6.1 Performance

Performance is extremely important for X.509 validation implementations. It affects the usability of browsers, since certificate validation is an unavoidable process of every new TLS connection, and it affects security, as slow X.509 validation might lead to denial-of-service attacks.

In this section, we show that Verdict, in addition to its benefits of formal verification and customizability, also has performance competitive with highly optimized implementations in browsers and TLS libraries.

Our main test suite for performance is a collection of 10,627,993 certificate chains derived from Certificate Transparency (CT) logs around 2020, which was collected by previous work, Hammurabi [32].

We developed benchmarking harnesses for X.509 validation implementations in Chrome, Firefox, and OpenSSL, as well as three related projects (discussed in more detail in §8): ARMOR [18], CERES [17], and Hammurabi [32]. In Figure 8, we summarize the performance statistics. In Figure 7, we show a more detailed performance comparison between our formalized Chrome, Firefox, and OpenSSL policies against their original implementations.

As we discuss in the benchmarking setup below, developing a perfectly fair benchmark for certificate validators is challenging, so the differences presented in these figures should be interpreted as estimates. However, they do show that the performance of Verdict is comparable to that of hand-optimized, production-level implementations. In our benchmarking setup, Verdict, on average, outperforms ARMOR by over 370$\times$, CERES by over 1,300$\times$, and Hammurabi by over 64$\times$. Our Firefox policy is 6% faster than the original implementation in Firefox, and our Chrome and OpenSSL policies are only 2$\times$ and 1.8$\times$ slower than the original implementations.

In both figures, we show an additional set of results for Verdict that uses more performant but unverified implementations of RSA and ECDSA P-256 signature checking from AWS-LC [57] (marked with $\star$), instead of using the verified versions from libcrux [37]. All other parts of Verdict are the same in both sets of results. This shows that Verdict's slower performance primarily comes from the cryptographic primitives, rather than our verified components (e.g., the parser, path builder, and policies). With this change, Verdict's mean performance exceeds Firefox (69 μs vs 167 μs) and OpenSSL (78 μs vs 97 μs), and it is on par with Chrome (86 μs vs 84 μs).

We now expand on our benchmarking setup, and discuss the results in more detail.

Benchmarking Setup. Benchmarking X.509 implementations in a fair way is challenging. Although the high-level goal of X.509 chain validation is the same, implementations can differ substantially in the exact validation tasks they perform. For example, Chrome and Firefox perform additional checks for Extended Validation (EV) certificates [21] and different certificate revocation checks.

We did our best to reduce the effect of these differences. For Chrome and Firefox, we disabled the following components in their X.509 validation code that differ or that are not yet modeled in Verdict: Extended Validation (EV); revocation checking; all logging and telemetry functionality; fallback checks for SHA1 signatures; all networking operations (e.g. Authority Information Access (AIA) [11] fetching

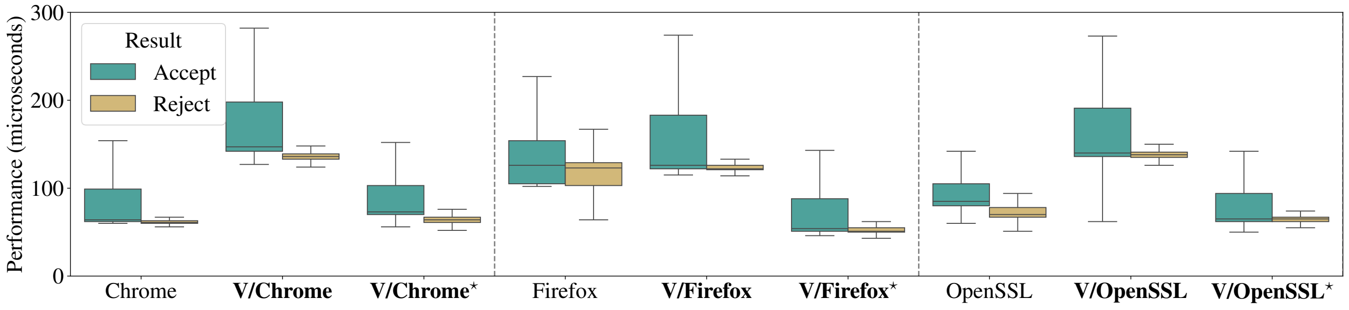

Figure 7: Performance (in microseconds) comparison between Verdict's Chrome, Firefox, and OpenSSL policies (prefixed with "V") and the original implementations, on a set of 10M certificates from CT logs. In each column, the left bar shows the validation time on accepted certificates, and the right bar shows that on rejected certificates. Each bar has five horizontal lines, representing the maximum, 75th quantile, median, 25th quantile, and minimum of each data set. Outliers (more than 1.5× the interquantile range above the 75th quantile or below the 25th quantile) are excluded for clarity. The notation of ⋆ is the same as Figure 8.

Impl.	Mean	Median	Min	Max
V/Firefox*	69	54	8	756
V/OpenSSL*	78	65	8	731
Chrome	84	64	20	553
V/Chrome*	86	73	8	791
OpenSSL	97	85	35	969
V/Firefox	157	126	8	2,498
V/OpenSSL	167	140	8	1,182
Firefox	167	126	20	1,860
V/Chrome	174	147	8	1,183
HM/Chrome	11,219	11,215	271	18,882
HM/Firefox	12,156	12,131	272	20,159
ARMOR	65,747	64,549	32,510	132,328
CERES	238,188	248,406	1,830	342,755

Figure 8: Performance statistics (in microseconds) of Verdict's instantiations on Chrome, Firefox, and OpenSSL policies (prefixed with "V"), along with other tested implementations, on 10M certificates from CT logs. HM/Chrome and HM/Firefox are Hammurabi's Chrome and Firefox policies, respectively. For Verdict, we show an additional set of results using more performant but unverified RSA and ECDSA P-256 implementations from AWS-LC [57] (marked with ⋆).

or OCSP [56]); and Signed Certificate Timestamp (SCT) [35] validation. We do not model some of these features in Verdict because they are declining in both browser support and usage in the wild (e.g., EV, SHA1, and OCSP).

With these changes, there are still some differences in the tested implementations that are difficult to remove but might affect performance. ARMOR and CERES do not support hostname validation nor the Name Constraints extension (so they perform fewer checks than the other tested implementations), while Firefox, Chrome, and OpenSSL have additional logic for error handling and reporting that are not modeled in the other implementations.

In all of the tested implementations, we only measure the time for (1) parsing the leaf and intermediate certificates (from Base64 or PEM), and (2) building and validating the certificate chain (including signature checking). In particular, parsing of the trusted root certificates is not included, and the root parsing results are cached for all tested implementations. Exclusion of root certificate parsing time significantly benefits some implementations, such as ARMOR, for which our profiling shows that root certificate parsing takes about 89% of the total time in ARMOR's original benchmarking setup.

To facilitate the benchmarking of these components, each benchmarking harness is written as a server in the native language of each implementation (i.e., C++ for Chrome and Firefox, C for OpenSSL, Python/Agda for ARMOR, Python for CERES, and Rust for Hammurabi). Each benchmarking server (1) initializes with a fixed set of root certificates, (2) loops to accept benchmarking requests consisting of the validation query (leaf and intermediate certificates, hostname, etc.) and the number of repetitions to perform, and then (3) repeats and times the validation of the request, using the most accurate wall-clock time available in the native language of each implementation.

We ran all experiments on a single machine with an Intel Core i9-13950HX CPU and 32 GiB of RAM. To reduce noise, for each certificate chain, we collected 10 wall-clock time measurements for each implementation, and then we used the minimum sample as the final time for that chain. We disabled hyper-threading and isolated the benchmark process to four performance cores pinned to the highest frequency.

For ARMOR, CERES, and Hammurabi, we were unable to finish testing for all 10M certificates due to time constraints, so we randomly sampled 100K (1%) of all 10M certificates for Hammurabi, and 10K (0.1%) for ARMOR and CERES.

All of the test harnesses and modified source code are available in our artifact [38].

Test	Impl.	A/A	A/R	R/A	R/R
CT	Chrome	10,600,892	0	0	27,101
	Firefox	10,600,919	0	0	27,074
	OpenSSL	10,600,836	0	7	27,150
Limbo	Chrome	541	3	3	4,739
	Firefox	569	2	2	4,713
	OpenSSL	572	10	26	4,678

Figure 9: Differential testing results. "CT" is the test suite of 10,627,993 certificate chains from Certificate Transparency logs. "Limbo" represents 5,286 tests from the x509-limbo [61] test suite. The column name "A/A" means both the original and Verdict versions accept; "A/R" means the original accepts but Verdict rejects; "R/A" means the original rejects but Verdict accepts; "R/R" means both reject.

Discussion. We attribute Verdict's good performance to several factors. First, Rust is a mature and performant language, and the verification tool Verus adds little overhead to its native performance. By implementing Verdict entirely in Verus, we can both state end-to-end correctness specifications, and avoid the overhead of inter-process communication (e.g., in the case of ARMOR, CERES, and Hammurabi).

Second, signature checking takes the majority of validation time in most X.509 implementations. In Verdict, we use a number of caches to avoid recomputing heavy cryptographic operations. For instance, RSA is the most common signature scheme in our test cases (92.7% of the 10M CT log chains use RSA + SHA-256 signatures). We cache the RSA public keys of the root certificates (including certain precomputation required for signature checking), which results in a 19% speedup in our experiments. Optimizations such as this are verified and cause no change to the end-to-end specification.

Compared to ARMOR, Verdict also has a much more performant parser, which we have designed such that most parser combinators are zero-copy.

There is still more room for performance improvements in Verdict in practice. For example, Chrome and Firefox may cache computation (e.g., signature checking of a common intermediate certificate) across multiple validation runs, which we do not yet implement in Verdict.

6.2 Differential Testing

Besides good performance, Verdict is also expressive enough to model complex policies in Chrome, Firefox, and OpenSSL.

We demonstrate this by performing differential testing between our formalized policies and their original implementations, on two test suites: (1) the 10M CT log chains from §6.1, and (2) a third-party synthetic test suite called x509-limbo [61] (Limbo), which is designed to catch inconsistencies on edge cases of X.509 validation. In Limbo, we use 5,286 out of 9,741 test cases, excluding tests that require either IP validation or client authentication, which are not currently supported by Verdict.

The differential testing results are shown in Figure 9. On common certificates from the CT logs, our formalized Chrome and Firefox policies agree completely with the original implementations, and our OpenSSL policy only disagrees with OpenSSL on 7 certificate chains, which are actually caused by a path builder bug in baseline OpenSSL (§6.4).

The Limbo test suite is much more challenging, and our formalizations diverge more from the original implementations. We took effort to fix inconsistencies between all the policies and their original implementations, and at the time of writing, the Chrome, Firefox, and OpenSSL policies are only inconsistent on 0.1%, 0.08%, and 0.7% of the Limbo tests. We attribute these differences to two types of issues:

- Unsupported features: A few tests require features that Verdict does not yet support, e.g., unsupported signature schemes of DSA, ECDSA P-192, and ECDSA with SHA-1. Verdict also does not check for intermediate self-issued certificates yet, and one test checks that an X.509 validator should not count self-issued intermediates when checking the path length limit.

- Chrome test harness issue: We use libfaketime [26] to set the system clock to a specific timestamp when the Chrome test harness starts, since Chrome does not have a direct interface to set the validation time. However, Limbo has tests where the validation time is exactly the same as the certificate's expiration time. In these cases, when Chrome's validator obtains the current time, the system clock has already advanced from the previously set timestamp, resulting in a validation timestamp that is not the value desired by the test harness.

Due to these minor inconsistencies, we do not claim that our formalizations of Chrome/Firefox/OpenSSL policies behave exactly the same as the original implementations. Nonetheless, our differential testing shows that Verdict is expressive enough for most if not all validation logic in complex, production-level X.509 validators.

We also tested ARMOR, CERES, and Hammurabi on Limbo and revealed a large number of issues, despite their focus on high assurance (§8). ARMOR's parser rejects all of the root certificates in the test suite, which is similar to their own results of testing on Frankencerts [4]. CERES fails 490 (9.2%) of the tests compared to the expected test results in Limbo, including 18 out of 75 manually written tests for RFC conformance. This shows that ARMOR and CERES's emphasis on strict RFC compliance (§8) is difficult to achieve and creates practical issues.

Hammurabi's Chrome and Firefox policies differ from the original implementations on 534 (10.1%) and 559 (10.6%) test cases, respectively. Among these inconsistencies, 407 are due to the incorrect handling of Name Constraints. For

Impl.	Mean	Max	Min	$\approx$	$+$	$-$
V/Chrome	0.29%	1.89%	-3.19%	39	7	54
V/Firefox	0.22%	1.56%	-1.17%	48	5	47
V/OpenSSL	0.27%	1.61%	-1.5%	39	11	50
V/Chrome*	0.11%	1.11%	-1.57%	59	2	39
V/Firefox*	0.06%	2.45%	-1.4%	71	5	24
V/OpenSSL*	0.05%	1.09%	-2.85%	60	3	37

Figure 10: Performance overhead (relative to the baseline) of making HTTPS requests to 100 popular websites [36] using Verdict-integrated Rustls. The Mean/Max/Min columns show the geometric mean, maximum, and minimum of the relative differences in HTTPS request time across all websites. The last three columns show the numbers of websites where sample mean differences have no statistical significance ($\approx$, $p \geq 0.05$), Verdict is faster ($+$, $p < 0.05$), and Verdict is slower ($-$, $p < 0.05$). The $\star$ notation is the same as Figure 8.

example, Hammurabi applies Name Constraints only to the leaf certificate, but not intermediate certificates.

6.3 Performance in Rustls

To demonstrate that Verdict can be easily used in existing TLS libraries, we integrated Verdict into a popular Rust TLS library called Rustls [54]. The integration is added directly through Cargo (Rust's package manager) and only involves 107 lines of additional Rust code in Rustls.

The performance overhead of using Verdict in Rustls in practical settings is minuscule. We measured the client-side performance of making HTTPS requests to the 100 most popular websites according to Tranco [36]. We used both the baseline version of Rustls, and our version that integrates Verdict. We found that on average, the overhead of using Verdict in Rustls is around 0.05–0.29% (Figure 10).

To set up this experiment, we first fetch certificate chains and HTTP responses from the first 100 most visited domains from the Tranco list [36]. We then replace public keys in these chains with freshly generated keys to mimic the public server locally, and use Rustls to make HTTPS requests to the local server, with a fixed, simulated network delay of 5 ms (using the `tc` command on Linux). We collected 100 samples for each domain and policy, after discarding warm-up rounds.

In Figure 10, we show that on almost half of the websites, there is no statistically significant difference between Verdict and the baseline, and in a few cases, Verdict outperforms the baseline. In general, the difference is very small (less than 0.3% on average), which is expected as the request time is dominated by network delay.

6.4 Security Issues Preventable by Verdict

In this section, we discuss several broad classes of security bugs that have plagued X.509 validators for years. These issues are *eliminated* through the formal verification of Verdict, where the correctness and security of the complex implementation is reduced to a human-readable, high-level specification.

We also mention a few bugs we found in popular X.509-related libraries while developing Verdict. Although not all of these bugs are immediately exploitable, Verdict's focus is not finding individual bugs, but on ruling out entire classes of bugs.

Malleable ASN.1 Object Identifier Encoding. When testing our parser against a popular Rust DER encoding/decoding library called `der` [53] (with over 80M downloads and used by popular cryptographic libraries in the Rust Crypto project [52]), we found that their encoding of ASN.1 object identifiers is incorrect and malleable in certain ranges. For example, "1.2.128" and "1.2.0" are both encoded to the same byte sequence. This issue has been confirmed by the author of `der` and has since been fixed. In §4.1, we show that Verdict's X.509 parser is formally verified to be non-malleable, thus ruling out any issue of this kind.

Path Building Incompleteness. During differential testing (§6.2), we found that OpenSSL sometimes fails to explore an alternative path when the first candidate fails, which is similar to an issue that caused a wide outage in 2020 [2]. In particular, if the first candidate path has a root missing the Subject Key Identifier extension, OpenSSL fails prematurely without searching for an alternative valid path. We have reported this issue to the OpenSSL developers.

Similarly, when studying Hammurabi's implementation [32], we found that it has another kind of incompleteness: it assumes that the input certificates are ordered such that any issuer certificate always appears after its subject certificate. For example, if two certificates in a chain are swapped, Hammurabi fails to find a valid path.

In Verdict, we formally prove that such incompleteness issues cannot happen.

Memory Safety Issues. X.509 implementations written in C have been riddled with memory issues such as buffer overflows, including some very recent CVEs [13–15]. These issues can be exploited to cause denial-of-service or even remote code execution attacks using adversarial certificates. Verdict's verified portion is free of invalid memory accesses (by using Rust), and it is also formally verified to be free of crashes caused by panics or integer overflows.

6.5 Verification Challenges and Takeaways

We summarize the main challenges we face during the verification effort of Verdict and how our design overcomes them.

End-to-End Specification. Existing X.509 validators, e.g., in Chrome/Firefox/OpenSSL, intermingle intricate policies with path building, cryptographic operations, and parsing. This makes it difficult to write a concise and end-to-end validator specification, as evidenced in part by ARMOR's lack thereof.

In Verdict, the separation of user-defined policy from the rest of the validator is an important factor that makes our verification effort tractable and easy to scale to existing production-level X.509 validators. We factor out the common part of their behavior (parsing and path building), and provide a policy DSL so that we can have a clean and customizable end-to-end specification (Figure 3).

Modeling Complex Policies. Modeling and verifying X.509 policies in a naïve way can easily lead to spec/code duplication, which is hard to maintain. This is because these policies are mostly mechanical checks on certificates, and so the complexity of their spec is almost the same as the complexity of the implementation. For example, in ARMOR [18], one must specify a predicate and an implementation for each RFC rule, even though they mostly express the same logic.

By using a policy DSL in Verdict and automatically compiling policy specs to verified executable code, we do not compromise on formal guarantees and, at the same time, improve the readability and maintainability of the policies.

The Verus community also has a more general need for executable specs, so we are working on upstreaming a generalized version of our policy compiler to Verus, with support for more primitives and robust generation of functional correctness proofs.

X.509/ASN.1 Parsing. The ASN.1 DER [60] format involves sophisticated bit-level rules for ensuring non-malleability and compactness. Prior work on formally verifying ASN.1 parsing, such as ARMOR [18] and ASN1* [41], has compromised on performance due to this complexity.

To achieve both performance and formal verification in our work, we use a modular and extensible design to first formalize ASN.1 primitives, and then use the ASN.1 DSL (§4.1) to easily express and verify complex X.509 formats in ASN.1. This allows us to build an X.509 parser that matches the performance of hand-written, unverified parsers in Chrome, Firefox, and OpenSSL, while providing strong guarantees of non-malleability and prefix security.

Verus vs. Agda. Although verifying imperative Rust code in Verus gives us much better control over performance, it also arguably increases the difficulty of the proofs as there is a larger gap between the pure functional specifications and the imperative code with mutable objects.

Verus's SMT-based automated verification paradigm helps alleviate this burden. A key benefit of using Verus is its fast verification time, which means that we can quickly learn whether a proof attempt fails and then iterate on it. As a rough comparison, on the machine we use for evaluation, it takes 57 seconds to verify the entire Verdict project with Verus (using a single thread), which includes 38 s for the parser, and 15 s for the path builder and policies. In comparison, ARMOR, which is written in Agda [42], takes 1,186 seconds to type check/verify.

7 Limitations

Given the complexity of the X.509 standards and implementations, Verdict still has some limitations.

Support for Revocation. In Verdict's Chrome and Firefox policies, we inherit some revocation mechanisms from Hammurabi by checking the certificate in question against a known list of revoked certificates (e.g. derived from Chrome's CRLSets [46]). However, more complex revocation mechanisms such as OCSP are not modeled in our policies, partially because they are undergoing active changes (e.g., OCSP is being phased out [1]). In general, to implement revocation in Verdict, one can fetch revocation information prior to policy evaluation, and then make the policy parametric in the revocation information, in the same style as Hammurabi.

Error Reporting. Accurately reporting suitable error messages for a rejected certificate is useful for debugging, and also has security implications when the user needs to see the error [4]. Currently, all policies in Verdict are defined as a Boolean predicate on a candidate certificate path. Our policy DSL does support custom enum types, so we leave better error reporting as future engineering work.

Trusted Computing Base. Besides the major tooling components we trust (e.g., the Rust compiler, Verus, LLVM), there are also some low-level utility functions not formally verified in Verdict. These include string manipulating functions in the Rust standard library (e.g., prefix/suffix checking, UTF-8 encoding and case folding), and the conversion from the ASN.1 [60] representations of time to UNIX timestamps (for which we use an unverified Rust library `chrono` [51]). We believe that verification of these components are orthogonal to our goal of formally verifying X.509 validation.

8 Related Work

The ubiquity and complexity of X.509 validation have led to abundant work in applying formal methods and testing techniques to X.509 validators to make them more trustworthy. We discuss related work in the following three categories.

Bug Finding Fuzzing [40] is commonly applied to test existing X.509 implementations. The Frankencerts work [4] randomly mutates existing certificates, and uses them to perform differential testing between different X.509 implementations. Multiple follow-up projects try to improve the coverage of Frankencerts [7, 45, 48]. Symbolic execution tools [6] have also been applied to X.509 validators for better coverage. While helpful for bug finding, these techniques do not provide the same level of guarantees as formal verification.

Separating Policy From Mechanism Larisch et al. disentangle X.509 validation policy from mechanism by expressing the policy as a Prolog program, which is then executed by the Hammurabi engine [32]. They make a compelling case for customization, on the grounds that in practice, there is no one true standard for X.509 certificate validation. Their approach inspired Verdict's support for custom validation policies.

Our contribution compared to Hammurabi is in the orthogonal direction of formal verification. Hammurabi is entirely unverified. It uses OpenSSL's parser, while we verify a performant parser for the complex X.509 format. We also find (§6.4) that its path builder is incomplete (i.e., there could be unexplored paths), while Verdict's path builder is provably complete. Verdict's policy framework also provides much stronger guarantees about policies than Hammurabi. From specifications written in Verdict's policy DSL, we automatically derive provably correct executable policies, verify their formal properties (e.g., RFC/BR conformance), and integrate policies directly into our end-to-end spec.

Using Prolog to express policies is often natural, but in some places leads to contortions. For instance, Hammurabi flattens the subject/issuer Distinguished Names (DNs) of a certificate for a more natural Prolog encoding, whereas in reality, DNs should be considered as a nested sequence of object-identifier-tagged strings. As a result, Hammurabi conflates names such as `[[CN=a], [OU=b]]` and `[[CN=a, OU=b]]`, which is not the case in common implementations.

Relying on Prolog also leads to poor performance (as shown in §6.1) and security risks. Because Hammurabi is unverified, all of its code, including the Prolog engine, must be trusted. In Hammurabi, potentially adversarial inputs (e.g., strings in a certificate) are directly translated into Prolog facts and mixed together with the policy program for execution. Since Prolog is untyped and has a flexible syntax, this opens up the risk of code injection.

CERES [17] models the validation rules from RFC 5280 separately in quantifier-free first-order logic (QFFOL). At runtime, it uses the CVC4 SMT solver [3] to check these rules against the given certificate chain. Compared to Verdict, CERES does not have any formally verified components, although one could consider the QFFOL rules as a policy specification. Even in that case, the lack of quantifiers imposes significant limitations on their expressiveness: they do not support any X.509 extensions that require the theory of arrays (e.g., Name Constraints), nor can they handle arbitrarily long chains (their SMT encoding is parametric in a constant maximum chain length). Adding quantifiers in CERES is not feasible, since SMT solvers cannot guarantee termination in most quantified theories. Using SMT solvers at runtime also leads to poor performance and security concerns, as both CVC4 and Z3 [16] are complex pieces of software written in a non-memory-safe language (C++).

Verified Implementations The ARMOR [18] project formally verifies parsing, chain building, and 27 required checks from RFC 5280 in the theorem prover Agda [42], making ARMOR a significant step towards a reference implementation of the RFC 5280 standard. However, their approach has several limitations compared to Verdict. First, they do not provide a clean, end-to-end specification for their entire pipeline, and there are multiple important components that live outside of the verified Agda code base, such as signature checking and the main Python driver that orchestrates the different modules. The goal of strictly conforming to RFC 5280 also has downsides. In particular, they do not model hostname validation or any rules from CA/B BRs [20], nor are their current RFC 5280 rules complete. They do not allow easy customization of the policy, and authors of most TLS libraries they tested against do not consider the discrepancies between ARMOR and their libraries to be bugs. The use of Agda also has practical limitations. As shown in §6.1, there is a large performance gap between ARMOR and Verdict. It also makes it difficult to maintain and integrate ARMOR into existing projects.

While ARMOR is the only prior work that focuses on full verification of X.509 validation, there is prior work that tackles X.509/ASN.1 DER parsing or serializing. In DICE* [58], authors verify a DICE-specific X.509 certificate serialization library without parsing or validation. The ASN1* library [41] formalizes a subset of the DER parsing specification in the EverParse framework [49], and verifies that the parser is non-malleable. Performance-wise, their parsers are extracted to slow OCaml code, and in some preliminary evaluation we conducted, on 1000 certificates from CT logs, our parser took 0.024 s, while ASN1* took 9.32 s, i.e., 388$\times$ slower.

9 Conclusion

In this work, we present Verdict, the first end-to-end formally verified X.509 certificate validator with practical performance and flexibility. In Verdict's policy DSL, we formalize three complex, production-level X.509 policies from Chrome, Firefox, and OpenSSL, and show that our versions are competitive against the original implementations. We believe our work is an important step towards applying practical formal verification to establish more secure and trustworthy X.509 PKIs.

Acknowledgments

We thank Samvid Dharanikota for his help in prototyping an early version of Verdict; James Larisch for his assistance with the Hammurabi code base and for providing feedback on an early draft of this paper; and Dana Keeler, Matt Mueller, and Lily Chen for their help with profiling X.509 validators in Firefox and Chromium. This work was supported in part by National Science Foundation (NSF) Grant No. 2224279, funding from AFRL and DARPA under Agreement FA8750-24-9-1000, and the Future Enterprise Security initiative at Carnegie Mellon CyLab (FutureEnterprise@CyLab). Pratap Singh was also funded by the NSF Graduate Research Fellowship Program under Grant No. DGE2140739.

Ethical Considerations

This work uses formal verification to construct X.509 validators free of security vulnerabilities for the benefit of the broader computing community. As part of our research, we have discovered a few (minor) issues in public X.509-related projects which have all been responsibly disclosed to their maintainers in a timely manner.

Open Science

The source code of Verdict and benchmarking harnesses for each X.509 implementation tested in §6 are available in our artifact [38] and GitHub repository [39].

References

[1] Josh Aas. Intent to end OCSP service. https://letsencrypt.org/2024/07/23/replacing-ocsp-with-crls/, 2024.

[2] Andrew Ayer. Fixing the breakage from the AddTrust external CA root expiration. https://www.agwa.name/blog/post/fixing_the_addtrust_root_expiration, 2020.

[3] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.

[4] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. Using Frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In *2014 IEEE Symposium on Security and Privacy*, pages 114–129, 2014.

[5] Yi Cai, Pratap Singh, Zhengyao Lin, Jay Bosamiya, Joshua Gancher, Milijana Surbatovich, and Bryan Parno. Vest: Verified, secure, high-performance parsing and serialization for Rust. In *Proceedings of the USENIX Security Symposium*, August 2025.

[6] Sze Yiu Chau, Omar Chowdhury, Endadul Hoque, Huangyi Ge, Aniket Kate, Cristina Nita-Rotaru, and Ninghui Li. Symcerts: Practical symbolic execution for exposing noncompliance in X.509 certificate validation implementations. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 503–520, 2017.

[7] Yuting Chen and Zhendong Su. Guided differential testing of certificate validation in SSL/TLS implementations. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 793–804, New York, NY, USA, 2015. Association for Computing Machinery.

[8] S. Chokhani, W. Ford, R. Sabett, C. Merrill, and S. Wu. RFC 3647: Internet X.509 public key infrastructure certificate policy and certification practices framework, 2003.

[9] The Chromium projects. https://www.chromium.org/, 2024.

[10] William F Clocksin and Christopher S Mellish. *Programming in PROLOG*. Springer Science & Business Media, 2003.

[11] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. RFC 5280: Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile, 2008.

[12] CVE-2014-1266. https://www.cve.org/CVERecord?id=CVE-2014-1266, 2014.

[13] CVE-2024-22041. https://www.cve.org/CVERecord?id=CVE-2024-22041, 2024.

[14] CVE-2024-28835. https://www.cve.org/CVERecord?id=CVE-2024-28835, 2024.

[15] CVE-2024-6119. https://www.cve.org/CVERecord?id=CVE-2024-6119, 2024.

[16] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[17] Joyanta Debnath, Sze Yiu Chau, and Omar Chowdhury. 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*, CCS '21, page 1388–1404, New York, NY, USA, 2021. Association for Computing Machinery.

[18] Joyanta Debnath, Christa Jenkins, Yuteng Sun, Sze Yiu Chau, and Omar Chowdhury. ARMOR: A formally verified implementation of X.509 certificate chain validation. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 1462–1480, Los Alamitos, CA, USA, May 2024. IEEE Computer Society.

[19] Mozilla Firefox. https://www.mozilla.org/firefox/, 2024.

[20] CA/Browser Forum. Baseline requirements for the issuance and management of publicly-trusted TLS server certificates. https://cabforum.org/working-groups/server/baseline-requirements/documents/CA-Browser-Forum-TLS-BR-2.1.2.pdf, 2024.

[21] CA/Browser Forum. EV TLS server certificate guidelines. https://cabforum.org/working-groups/server/extended-validation/documents/, 2024.

[22] R. Frost and J. Launchbury. Constructing natural language interpreters in a lazy functional language. *Comput. J.*, 32(2):108–121, April 1989.

[23] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world: Validating SSL certificates in non-browser software. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, page 38–49, New York, NY, USA, 2012. Association for Computing Machinery.

[24] GnuTLS. https://gnutls.org/, 2024.

[25] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[26] Wolfgang Hommel. libfaketime. https://github.com/wolfcw/libfaketime, 2024.

[27] S. Josefsson. RFC 4648: The Base16, Base32, and Base64 data encodings, 2006.

[28] B. Kaliski. RFC 2313: PKCS #1: RSA encryption version 1.5, 1998.

[29] S. Kent and K. Seo. RFC 4301: Security architecture for the internet protocol, 2005.

[30] Steve Klabnik and Carol Nichols. The Rust programming language. https://doc.rust-lang.org/book/, 2024.

[31] Platon Kotzias, Srdjan Matic, Richard Rivera, and Juan Caballero. Certified PUP: Abuse in Authenticode code signing. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 465–478, New York, NY, USA, 2015. Association for Computing Machinery.

[32] James Larisch, Waqar Aqeel, Michael Lum, Yaelle Goldschlag, Leah Kannan, Kasra Torshizi, Yujie Wang, Taejoong Chung, Dave Levin, Bruce Maggs, Alan Mislove, Bryan Parno, and Christo Wilson. Hammurabi: A framework for pluggable, logic-based X.509 certificate validation policies. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, November 2022.

[33] Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Oded Padon, and Bryan Parno. Verus: A practical foundation for systems verification. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 438–454, New York, NY, USA, 2024. Association for Computing Machinery.

[34] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying Rust programs using linear ghost types. *Proc. ACM Program. Lang.*, 7(OOPSLA1), April 2023.

[35] B. Laurie, A. Langley, and E. Kasper. RFC 6962: Certificate transparency, 2013.

[36] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium*, NDSS 2019, February 2019.

[37] libcrux: The formally verified crypto library for Rust. https://github.com/cryspen/libcrux, 2024.

[38] Zhengyao Lin, Michael McLoughlin, Pratap Singh, Rory Brennan-Jones, Paul Hitchcox, Joshua Gancher, and Bryan Parno. Towards practical, end-to-end formally verified X.509 certificate validators with Verdict. https://doi.org/10.5281/zenodo.15468400, May 2025.

[39] Zhengyao Lin, Michael McLoughlin, Pratap Singh, Rory Brennan-Jones, Paul Hitchcox, Joshua Gancher, and

Bryan Parno. Verdict: Verified X.509 certificate validation. https://github.com/secure-foundations/verdict, 2025.

[40] Valentin J.M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2021.

[41] Haobin Ni, Antoine Delignat-Lavaud, Cédric Fournet, Tahina Ramananandro, and Nikhil Swamy. ASN1*: Provably correct, non-malleable parsing for ASN.1 DER. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2023, page 275–289, New York, NY, USA, 2023. Association for Computing Machinery.

[42] Ulf Norell. Dependently typed programming in Agda. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, TLDI '09, page 1–2, New York, NY, USA, 2009. Association for Computing Machinery.

[43] National Institute of Standards and Technology. Digital signature standard (DSS). https://csrc.nist.gov/pubs/fips/186-4/final, 2013.

[44] OpenSSL. https://www.openssl.org/, 2024.

[45] Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D. Keromytis, and Suman Jana. Nezha: Efficient domain-independent differential testing. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 615–632, 2017.

[46] Chromium Project. CRLSets. https://www.chromium.org/Home/chromium-security/crlsets/, 2024.

[47] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph M. Wintersteiger, and Santiago Zanella-Beguelin. EverCrypt: A fast, verified, cross-platform cryptographic provider. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 983–1002, 2020.

[48] Lili Quan, Qianyu Guo, Hongxu Chen, Xiaofei Xie, Xiaohong Li, Yang Liu, and Jing Hu. SADT: Syntax-aware differential testing of certificate validation in SSL/TLS implementations. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, page 524–535, New York, NY, USA, 2021. Association for Computing Machinery.

[49] Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. EverParse: Verified secure zero-copy parsers for authenticated message formats. In *Proceedings of the USENIX Security Symposium*, August 2019.

[50] The Rust programming language. https://www.rust-lang.org/, 2024.

[51] Chrono: Date and time library for Rust. https://github.com/chronotope/chrono, 2024.

[52] Rust Crypto. https://github.com/rustcrypto, 2024.

[53] Rust Crypto: ASN.1 DER. https://github.com/RustCrypto/formats/tree/master/der, 2024.

[54] Rustls: A modern TLS library in Rust. https://github.com/rustls/rustls, 2024.

[55] P. Saint-Andre and J. Hodges. RFC 6125: 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), 2011.

[56] S. Santesson, M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams. RFC 6960: X.509 internet public key infrastructure online certificate status protocol - OCSP, 2013.

[57] Amazon Web Services. AWS libcrypto. https://github.com/aws/aws-lc, 2024.

[58] Zhe Tao, Aseem Rastogi, Naman Gupta, Kapil Vaswani, and Aditya V. Thakur. DICE*: A formally verified implementation of DICE measured boot. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1091–1107. USENIX Association, August 2021.

[59] International Telecommunications Union. The directory: Authentication framework, 1988.

[60] International Telecommunications Union. Information technology - ASN.1 encoding rules: Specification of basic encoding rules (BER), canonical encoding rules (CER) and distinguished encoding rules (DER). https://www.itu.int/rec/T-REC-X.690/en, 2002.

[61] x509-limbo. https://x509-limbo.com/, 2024.

[62] Scott Yilek, Eric Rescorla, Hovav Shacham, Brandon Enright, and Stefan Savage. When private keys are public: Results from the 2008 Debian OpenSSL vulnerability. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement*, IMC '09, page 15–27, New York, NY, USA, 2009. Association for Computing Machinery.