ARMOR: A Formally Verified Implementation of X.509 Certificate Chain Validation (Full Version)

***Abstract*—X.509 PKI as an authentication mechanism is widely used as a building block for achieving security guarantees in many applications and protocols. At the core of its authentication guarantees lies the assumption that one can *correctly* check whether a given chain of X.509 certificates is legitimate. Since noncompliance with the X.509 standard and other vulnerabilities in implementations of the X.509 certificate chain validation can lead to interoperability issues or even impersonation attacks, they are hailed as the "*most dangerous code in the world*." Almost all existing efforts in evaluating the correctness of implementations of X.509 rely on software testing. In the words of the famous computer scientist Edsger Dijkstra, *"Program testing can be used to show the presence of bugs, but never to show their absence!"* This sentiment is corroborated by the discovery of highly influential bugs and vulnerabilities in widely used and tested open-source X.509 implementations. Therefore, we set out to substantially improve this unsatisfactory status quo by developing a high-assurance implementation for the X.509 certificate chain validation with formal correctness guarantees, called ARMOR. ARMOR features a modular architecture in which each stage of the certificate chain validation process is captured as a separate module. The formal correctness guarantees for each of these modules are then mechanically proved using the `Agda` interactive theorem prover. To demonstrate its efficacy, ARMOR is compared with 11 open-source X.509 implementations for its specificational accuracy and runtime overhead. In our evaluation, ARMOR incurs high overhead but remains strictly standards-compliant. Finally, we show an end-to-end application of ARMOR by integrating it with the TLS 1.3 implementation of BoringSSL and testing it with Curl.**

1. Introduction

The X.509 PKI standard [1] provides a scalable way to verify the authenticity of the binding of an entity's identity with its public key. This identity-public-key binding is represented as an X.509 certificate, which is digitally signed by an issuer (*e.g.*, certificate authority or CA), signifying the issuer's trust in the authenticity and integrity of this binding. For scalably establishing the authenticity and integrity of a certificate, the X.509 standard takes advantage of the *transitivity* of this "*trust*" relationship. This intuition is realized in the X.509 standard [1] through a *certificate chain validation* algorithm. Concretely, when an entity e_1 wants to check whether the certificate (given, as part of an input chain of certificates) of another entity e_2 is authentic, this algorithm *conceptually* starts with the certificate of a trust anchor (*i.e.*, an issuer who is unconditionally trusted by e_1) and then attempts to extend this absolute trust through a chain of the input certificates, all the way down to e_2.

Implementations of X.509 certificate chain validation, hailed as the "*most dangerous code in the world*" [2], are thus critical for ensuring the authentication guarantees promised by X.509 PKI [1]. Along with its authentication guarantees, X.509 also provides a scalable and flexible mechanism for public-key distribution. These desirable guarantees of X.509 PKI are often used as fundamental building blocks for achieving other security assurances such as *confidentiality*, *integrity*, and *non-repudiation* in many applications, including but not limited to SSL/TLS, IPSec, HTTPS, Email, WiFi, code signing, secure boot, firmware/software verification, and secure software update. Given its pivotal role in the overall system, software, and communication security, ensuring the *correctness* of X.509 certificate validation is of utmost importance. Incorrect validation could lead to a system accepting a malicious or invalid certificate, potentially exposing the system to man-in-the-middle (MITM) and impersonation attacks. Similarly, incorrectly rejecting a valid certificate could give rise to interoperability issues.

The majority of prior work focuses on developing software testing mechanisms specialized for checking the correctness of different X.509 libraries [3], [4], [5], [6], [7], [8], [9]. While these methods have been beneficial in identifying numerous vulnerabilities, they often fall short of providing any formal guarantees regarding correctness. This is corroborated through many high impact bugs and vulnerabilities in some widely used applications and open-source libraries [10], [11], [12], [13], [14], [15], [16]. In contrast, a formally-verified implementation of X.509 certificate chain validation can provide mathematical assurances that the implementation behaves correctly, setting a benchmark for developing other such implementations. Such a formally-verified implementation, however, is currently missing from the literature. *The current paper takes a major step to addresses this research gap by designing and developing a high-assurance application for X.509 certificate chain validation, named* ARMOR*, whose compliance with the standard is established by formal, machine-checked proofs.*

Although the current paper, to the best of our knowledge, presents the first implementation of X.509 certificate chain validation with machine-checked proofs of correctness, it draws inspirations from prior work in the area [17], [18], [19], [20], [21], [22]. However, in comparison to ARMOR, prior work has at least one of the following limitations: (1) no formal guarantees; (2) focuses only on parsing and lacks

formal correctness guarantees of semantic aspects; (3) lacks explicit proof of *soundness* and *completeness* of certificate parsing; (4) focuses only on verified encoding of certificates, not parsing.

Challenges. Developing ARMOR required addressing the following technical challenges. *First*, the X.509 specification is distributed across many documents (*e.g.*, ITU-T X.509 [23], RFC 5280 [1], RFC 6125 [24], RFC 4158 [25], RFC 2527 [26], RFC 4518 [27]), and its natural language specification has been shown to suffer from inconsistencies, ambiguities, and under-specification [21], [28], [29]. *Second*, the format of an X.509 certificate is complex and nested, represented in ASN.1 X.690 DER (Distinguished Encoding Rules) [30], and one requires a *context-sensitive* grammar to enforce the syntactic requirements of an X.509 certificate [21], [31]. Thus, proving total correctness of the parser is quite complicated. To make matters worse, parsing just the ASN.1 structure from the certificate byte stream is insufficient because the relevant certificate field value may need to be further decoded from the parsed ASN.1 DER value. *Finally*, the X.509 chain validation can be conceptually decomposed into different stages (*i.e.*, PEM parsing, Base64 decoding, X.690 DER parsing, string canonicalization, chain building, semantic validation, signature verification), each of which can be complex by itself (see [32], [33], [34]).

Approach. ARMOR is designed and developed with modularity in mind. Inspired by prior work [17], [21], we modularly decompose the whole X.509 certificate chain validation process into several modules, making manageable both the implementation and formal verification efforts. In particular, we formulate correctness guarantees for each module, which can then be discharged independently. ARMOR is organized into five modules: parser, chain builder, string canonicalizer, semantic validator, and driver. The *driver*, written in `Python`, stitches together the different components and exposes an interface expected from an X.509 implementation. The rest of the modules, written in the dependently typed functional programming language `Agda` [35], [36], implement all the intermediate stages of certificate chain validation. Notably, one can both write programs in `Agda` and also prove their correctness using *interactive theorem proving*.

Our general approach of verification carefully separates the specificational elements from the implementation elements by using relational specifications. As an example, for our approach to parser verification, we use *relational, parser-independent* specifications of the PEM, X.690 DER, and X.509 formats. Compared to approaches that verify parsers with respect to serializers, our approach greatly reduces the complexity of the specifications and provides a clear distinction between correctness properties of the *language* and the *parser*. To illustrate this distinction, for our X.690 DER and X.509 *parsers* we have proven *soundness* (any bytestring accepted by the parsers conforms to the format specification) and *completeness* (any bytestring that conforms to the format specification is accepted by the parser). For our X.690 DER and X.509 *language formalizations*, we have proven *unambiguousness* (e.g., one bytestring cannot be the encoding of two distinct X.509 certificates) and *non-malleability* (e.g., two distinct bytestrings cannot be the encoding of the same X.509 certificate) required for the above guarantees. For the full listing of properties proven, see Table 4 in the Appendix. Once these proof obligations are discharged, we use `Agda`'s extraction mechanism to obtain an executable, which is then invoked by the driver.

Evaluation. As ARMOR, or any formally verified software, is only as good as its specification, it is crucial that we compare ARMOR to other implementations to gain assurance that our formalization of the natural language specification is indeed correct. We *differentially test* ARMOR against 11 open-source X.509 libraries, using around 3 million certificates from four different datasets. We observe that ARMOR agrees with most libraries at least 99% of the time. For the remaining 1%, we notice that ARMOR strictly follows the requirements in RFC 5280 [1], whereas the other libraries have a more relaxed enforcement of these requirements. Finally, to evaluate the practicality of ARMOR, we measure its runtime overhead in terms of computational time and memory consumption. We notice that ARMOR has a much higher overhead compared to the X.509 libraries that are written in `C/C++`, `Python`, `Java`, and `Go`. Our empirical evaluation signifies that ARMOR *may be a reasonable choice of X.509 certificate validation application in some application domains where formal correctness is more important than runtime overhead.*

Impact. ARMOR can substantially improve the security of critical applications that rely on X.509 PKI (e.g., SSL/TLS). As an example, the existing formally verified TLS 1.2 implementation [37] requires a correct X.509 PKI implementation to ensure its guarantees, which ARMOR can fulfill. To evaluate the practicality of using ARMOR as part of a TLS implementation, we integrate it with the TLS 1.3 implementation of BoringSSL [38] and evaluate its performance. We observe that ARMOR introduces significant overhead during TLS handshake. ARMOR can also be used as an oracle for testing other X.509 implementations. Finally, our relational language specifications can serve as a separate, formal reference for programmers to consult.

Contributions. We make five technical contributions.

1) We present a formalization of the requirements of the X.509 standard and a modular decomposition of them, facilitating development of other such formally-verified implementations in the future.
2) We prove that our formalization of the X.509 syntactic requirements is *unambiguous* and *non-malleable*.
3) We present the design and implementation of ARMOR, whose verified modules enjoy total correctness guarantees with respect to our formalized specification.
4) We evaluate ARMOR with respect to its specificational accuracy and overhead against 11 open-source libraries, and analyze its performance and effectiveness in practice.
5) We show an end-to-end application of ARMOR, integrating it with TLS 1.3 implementation of BoringSSL and testing with the widely-used application Curl [39].

Responsible Disclosure. We are currently in the process of sharing our findings with the library developers.

2. Background and Motivation of ARMOR

This section presents a primer on X.509 certificates and certificate chain validation, and the motivation for ARMOR.

2.1. Preliminaries on X.509 Certificate Chain

Though the X.509 standard is primarily defined in ITU-T X.509 [23], RFC 5280 [1] provides additional restrictions and directions to use X.509 certificate for the Internet domain. Particularly, RFC 5280 concentrates on version 3 of the certificate standard and the usage of different extensions, which is the main focus of this work.

Internal Structure of a Certificate. A version 3 certificate comprises three top-level fields, namely, TBSCertificate, SignatureAlgorithm, and SignatureValue. The TBSCertificate field contains information such as the certificate version, a unique serial number, the validity period, the certificate issuer's name, and the certificate owner's name (*i.e.*, subject). It also includes the public key, the algorithm employed by the issuer for signing the certificate, and a few *optional* fields such as the unique identifiers and a sequence of extensions, specifically for version 3 of the X.509 standard. The issuer CA signs the entire TBSCertificate content, generating a signature, denoted as SignatureValue, which is appended to the end of the certificate, creating a digitally secure and tamper-proof container. The SignatureAlgorithm field specifies the algorithm used by the issuer CA for generating the signature.

Certificate Chain Validation. A certificate chain $\mathcal{C}$ can be *conceptually* viewed as an ordered sequence of certificates, $\mathcal{C} = [C_1, C_2, \ldots, C_{n-1}, C_n]$, in which C_1 to C_{n-1} are the (intermediate) CA certificates whereas C_n is the end-user certificate. Each certificate C_i is issued by its predecessor C_{i-1} (see Figure 1). Roughly, the certificate chain validation logic (**CCVL**) can be *conceptually* decomposed into the following stages: *parsing*, *transformation and pre-processing*, and *semantic condition checking* (See Figure 2).

The parsing stage checks to see whether each certificate C_i in $\mathcal{C}$ is syntactically well-formed and then parses it in an intermediate representation. After parsing, the intermediate representation of $\mathcal{C}$ goes through a series of transformations and pre-processing. The semantic condition checking stage checks to see whether the standard-prescribed semantic conditions are fulfilled. These conditions can be on a single certificate (*e.g.*, the certificate is not expired, the signature is verified) or across certificates (e.g., the subject name of the certificate C_{i-1} is the same as the issuer name of the certificate C_i). Finally, one checks to see whether C_1 is present in the trusted root store. All of these checks together allows one to extend the unconditional trust of C_1 through the intermediate CA certificates (C_2 to C_{n-1}), all the way down to the end-user certificate (C_n).

For ease of exposition, the certificate chain validation described here is intentionally left abstract. An implementation additionally has to take into account different corner cases, such as the presented certificate chain $\mathcal{C}$ not being in the correct hierarchical order, may not contain some CA certificates, or may contain duplicates. It is the implementation's responsibility to construct potential chains and try to verify them. For a detailed description of the entirety of **CCVL**, interested readers can consult RFC 5280 [1].

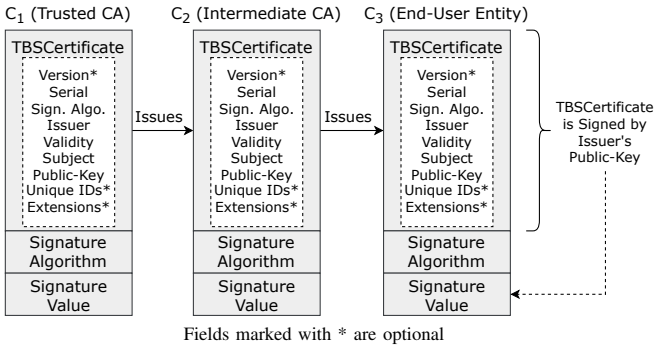


Figure 1: Representation of an X.509 certificate chain

2.2. Motivation of ARMOR

A majority of the existing work focuses on testing the logical correctness of the certificate chain validation. These efforts can be categorized into approaches that use *Fuzzing* [3], [4], [5], [6], [7] and *Symbolic Execution* [8], [9]. One of the main challenges all of these approaches have to address is a lack of *test oracle*. Most of the prior approaches rely on differential testing, where different implementations are used as *cross-checking test oracles*. These approaches, however, can potentially suffer from undetected bugs, especially in the case that the implementations under test have the same logical error. Having a formally verified *test oracle* like ARMOR can substantially decrease the chance of undetected bugs. In addition, these approaches cannot provide much mathematical assurance of correctness for the tested implementations. This is corroborated through many high impact bugs and vulnerabilities found in some widely used applications and open-source libraries over the last decade [10], [11], [12], [13], [14], [15], [16]. In contrast, a formally verified implementation of X.509 certificate validation like ARMOR can give mathematical assurance that it does not suffer from such logical bugs.

3. Design of ARMOR

We now present the design of ARMOR along with its verification philosophy and technical challenges.

3.1. Technical Challenges

Realizing ARMOR's vision requires addressing the following challenges.

Complexities in Specifications. The X.509 specification is distributed across different documents (*e.g.*, ITU-T X.509 [23], RFC 5280 [1], RFC 6125 [24], RFC 4158 [25], RFC 2527 [26], RFC 4518 [27]). The natural language specification has been shown to suffer from inconsistencies,

ambiguities, and under-specification [21], [28], [29]. As an example, consider the following requirements of a certificate's *serial number*, quoted from RFC 5280 [1].

> *"The serial number MUST be a positive integer assigned by the CA to each certificate. [...] CAs MUST force the serialNumber to be a non-negative integer."*

The two requirements here are inconsistent, as one part excludes zero as serial number while the other allows it.

Moreover, RFC 5280 encompasses rules not only for the certificate issuers (*i.e.*, *producer* rules) but also for the implementations that validate certificate chains (*e.g.*, *consumer* rules). In another way, RFC 5280 can be categorized into *syntactic* and *semantic* rules. While the syntactic rules are concerned with the parsing of an X.509 certificate serialized as a byte string, the semantic rules impose constraints on the values of individual fields within a certificate and on the relationships between field values across different certificates in a chain. Unfortunately, these intertwined sets of rules further complicate the specification, making it challenging to determine how an X.509 consumer implementation should respond in certain cases (*i.e.*, whether to accept a chain).

Complexities in DER Parsing. The internal representation of an X.509 certificate, while described in the *Abstract Syntax Notation One* (ASN.1), is eventually serialized using the X.690 Distinguished Encoding Rules (DER) [30]. This DER representation of the certificate byte string internally has the form $\langle t, \ell, v \rangle$, where t denotes the type, v indicates the actual content, and ℓ signifies the length in bytes of the v field. Additionally, the v field can include multiple and nested $\langle t, \ell, v \rangle$ structures, adding additional layers of complexity to the binary data. Parsing such binary data is challenging and error-prone since it always requires passing the value of the ℓ field (length) to accurately parse the subsequent v field. Since the internal grammar of a DER-encoded certificate is *context-sensitive*, developing a *correct* parser for such a grammar is non-trivial [21], [31].

To make matters worse, just correctly parsing the ASN.1 structure from the certificate byte string is insufficient because the relevant certificate field value may need to be further decoded from the parsed ASN.1 value. Take the example of X.509 specification for using the UTCTime format in the certificate validity field. It uses a two-digit year representation, YY, and here lies the potential for misinterpretation. In this format, values from 00 to 49 are deemed to belong to the $21st$ century and are thus interpreted as $20YY$. In contrast, values from 50 to 99 are associated with the $20th$ century and are consequently translated into $19YY$. These restrictions on the UTCTime format allow the representation of years only from 1950 to 2049. Therefore, library developers need to be very careful to decode the actual value of UTCTime to avoid potential certificate chain validation errors, a mistake previously found by Chau *et al.* [9] in some TLS libraries (*e.g.*, MatrixSSL, axTLS).

Supporting Different Certificate Representations. An X.509 implementation has to expose different interfaces for supporting different representations of an X.509 certificate. As an example, the certificates in a root store are saved in the PEM format whereas the certificates obtained during a TLS connection are represented as a DER encoded byte string.

Complexities in Individual Stages. The X.509 certificate chain validation algorithm can be conceptually decomposed into different stages, each of which has its own challenges. To give a few examples: (1) building a valid *certification path* can be difficult due to the lack of concrete directions as well as the possibility of having multiple valid certificate chains [32]; (2) string canonicalization [27], where strings are converted to their *normalized* forms, is also a complex process, since the number of character sets is humongous considering all the languages worldwide; and (3) during signature verification, the implementation needs to carefully parse the actual contents of the SignatureValue field with relevant cryptographic operations to prevent attacks (*e.g.*, *RSA signature forgery* [40], [41]). While these intermediate stages are conceptually straightforward, implementing them securely and proving their correctness is non-trivial.

3.2. ARMOR's Verification Philosophy

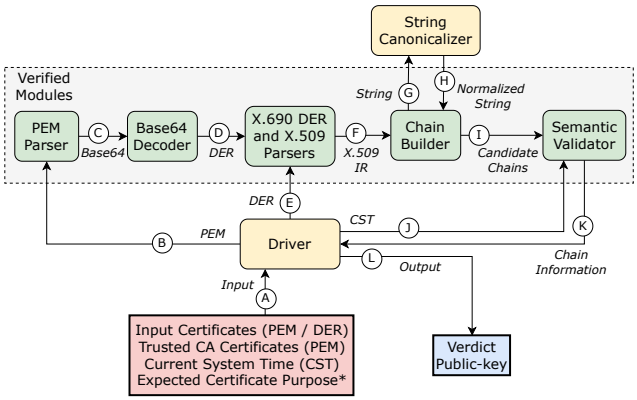
Relational Specifications. The central tenant of our approach to formally verifying ARMOR is to do so with respect to high-level, relational, and implementation-independent specifications. Our motivation for adhering to this discipline is two-fold.

1) **A specification is always part of the trusted computing base (TCB).** Formally verified software is only as trustworthy as the specification with respect to which it is verified. *Relational* specifications that describe how the input and output are related without referencing implementation details are, in general, simpler. Such specifications are also easier for humans to evaluate for trustworthiness than specifications that reference implementation details [21].
2) **A specification can be valuable in its own right.** Specifications are useful documentation, and made all the more valuable by being applicable to a wide range of implementations for a particular software task. Due to the inherent complexity of the X.509 CCVL, there is a vast space for non-trivial variations in implementations (*e.g.*, combining parsing with semantic validation), something that RFCs specifying X.509 CCVL explicitly acknowledge and aim to accommodate. Rather than providing correctness proofs that are limited to our particular implementation, we seek to provide a formal, machine-checked alternative to the RFCs by giving *implementation-agnostic* correctness specifications.

As a concrete example, consider the task of formally verifying a particular sorting algorithm. We could either prove it correct by showing it is extensionally equal to some other sorting algorithm (*e.g.*, *mergesort*), *or* state the correctness property relationally: *the output of the sorting function is a permutation of the input with the property that for every adjacent pair of elements, the first is no greater than the second*. Not only is it clear that it is the second, relational property that we ultimately care about for a sorting

algorithm, if we did not already have this as our intuition for *what a sorting should achieve*, then the usefulness of the first property *as a form of communication* is limited.

Modularity. We decompose ARMOR into independent modules (see Figure 2), which facilitates both our implementation and verification efforts. Also lying behind this design choice is a philosophical concern, namely *what should the formal end-to-end guarantees of X.509 CCVL even be?* The input to ARMOR is a character string and the result is a verdict and a public key. While we could present a relational join of each of the correctness properties of each module as an end-to-end guarantee, in our view this "leaks" implementation details, specifically our modular decomposition of X.509 CCVL (an approach not shared by most implementations). We therefore refrain from positioning our results as an end-to-end guarantee, leaving such a task for future research.



Inputs marked with * are optional

Figure 2: Conceptual design and workflow of ARMOR

3.3. ARMOR's Architecture

Figure 2 shows the architecture and workflow of ARMOR. ARMOR (A) takes a certificate chain, a list of trusted CA certificates, the current system time, and optionally the expected certificate purpose as input, and (L) outputs the certificate validation result (*i.e.*, verdict) as well as the public key of the end-user certificate. (B) The PEM parser reads a PEM certificate file and converts each certificate into its Base64 encoded format (sextets, *i.e.*, unsigned 6 bit integers). (C) The Base64 Decoder converts the sextet strings into octet strings (*i.e.*, unsigned 8 bit integers). (D) The X.690 DER parser and X.509 parser collaboratively parse the DER byte string and convert each certificate into an intermediate representation (X.509 IR). Note that if a certificate is already given in DER format as input, (E) we can directly call the DER parser. Next, (F) The chain builder constructs candidate chains from the parsed certificates, (G) – (H) utilizing the string canonicalizer to normalize strings in the certificate's Name field for accurate comparison. The semantic validator evaluates each candidate chain against certain semantic rules upon receiving (I) the candidate chains and (J) the current system time, and (K) informs the driver whether any chain passes all the semantic checks. In this design, the driver is the central component that orchestrates the entire process. The driver's role is multifaceted: (1) it activates the parser modules with the correct input; (2) it initiates the chain builder to form candidate chains; (3) it directs the semantic validator with the required input; and (4) upon success of the previous stages, the driver checks the consistency of the end-user certificate's purpose with respect to the verifier's given expected purpose, verifies signatures of the chain, and finally displays the validation outcome to the verifier.

Table 1: Correctness guarantees for each module in ARMOR

	PEM Parser	Base64 Decoder	DER Parser	Chain Builder	Semantic Validator
Correctness Properties	Sound Complete Maximal Terminating	Sound Complete Terminating	Sound Complete Terminating	Sound Complete Terminating	Sound Complete Terminating
Language Security Properties	Unambiguous	N/A	Unambiguous Non-malleable Unique prefixes	N/A	N/A

4. Verification Goals and Correctness Proofs

We now discuss ARMOR's correctness proofs. We provide formal correctness guarantees for the following modules of ARMOR: *parsers (i.e., PEM, X.690 DER, and X.509 parsers)*, *Base64 decoder*, *Semantic validator*, and *Chain builder* (see Table 1 for a summary of guarantees organized by module). For these verification tasks, which took 12 person months to complete, we use the `Agda` interactive theorem prover [35], [36]. See Table 4 for a listing and brief description of all formal guarantees proven.

Trusted Computing Base (TCB). Our TCB comprises the `Agda` toolchain (v2.6.2.2), which includes its native type-checker, compiler, and standard library (v1.7.1). Our use of `Agda`'s standard library includes the module `Data.Trie` (for the *String canonicalizer*), which requires the `--sized-types` language feature, and the module `IO`, which requires the `--guardedness` language feature. The use of these two features together *in the declaration of a coinductive type* causes logical inconsistency [42]. In our code base, the only module which enables both features is the *Driver*. It, however, does not define any coinductive types. Finally, ARMOR uses Agda's FFI for two Haskell packages: `time` and `bytestring`.

Termination. By default, `Agda` employs a syntactic termination checker that ensures recursive functions respect a certain well-founded ordering [43]. This syntactic termination checker can be disabled through the explicit use of certain pragmas, or replaced with a *type-based* termination checker through the use of sized types. ARMOR does not use any pragmas that disable termination checking, so its termination is guaranteed by `Agda`'s syntactic checker everywhere except the *String canonicalizer* and its co-dependencies, whose termination guarantee additionally rests on the correctness of `Agda`'s type-based checker.

Other Assumptions. We also make the following assumptions: (1) the GHC `Haskell` compiler correctly generates the executable; (2) the verifier's trusted root CA store is up-to-date and does not contain any malicious certificates; (3) the current system time is accurate.

4.1. Preliminaries on Agda

`Agda` is a *dependently-typed* functional programming language, meaning that types may involve terms (that is, program-level expressions). This capability helps express rich properties of programs *in the types of those programs*, and checking that programs satisfy those properties is reduced to typechecking. This paradigm, known as the *Curry-Howard* correspondence [44], means we can view `Agda`'s types as *propositions* and its programs as *proofs* of the propositions expressed by their types.

Consider the example shown in Figure 3 of nonnegative integers strictly less than some upper bound, provided as part of the `Agda` standard library as *Fin*. *Fin* defines an

```
data Fin : Nat → Set where
  fzero : {n : Nat} → Fin (1 + n)
  fsuc  : {n : Nat} → (i : Fin n) → Fin (1 + n)
toNat : ∀ {n} → Fin n → Nat
toNat fzero = 0
toNat (fsuc i) = 1 + (toNat i)
```

Figure 3: Length-indexed lists in `Agda`

inductive family of types, where the family is indexed by a non-negative integer. In other words, for every nonnegative integer $n : Nat$, $Fin\ n$ is a unique type whose inhabitants correspond to the nonnegative integers strictly less than n.

We now explain the declaration of *Fin*.

- The **data** keyword introduces a new inductive type or type family, in this case *Fin*.
- *Set* is the type of (small) types (we omit the details of `Agda`'s universe hierarchy).
- *Fin* has two constructors, both of which have dual readings as "mere data" and as axiomatizations of the *"is strictly less than"* relation. As mere data, *fzero* corresponds to the integer 0; as an axiom, it states that 0 is strictly less than the successor $1 + n$ of any nonnegative n. Similarly, as mere data *fsuc* is a primitive successor operation (like the Peano numbers), and as an inference rule, it states that if i is strictly less than n, then its successor is strictly less than $1 + n$.
- Curly braces indicate function arguments that need not be passed explicitly. For example, if i has type $Fin\ 5$, then Agda can determine $fsuc\ i$ has type $Fin\ 6$.

Since *Fin* is inductive, we can define functions over it by *pattern matching* and *recursion*. This is shown with function *toNat*, which we can think of as extracting the "mere data" contained in an expression of type $Fin\ n$.

- In the type signature, we use the syntactic sugar $\forall$ to omit the type of the parameter n, as Agda can infer this from the occurrence of n in the rest of the type.
- The definition of *toNat* is given with two equations, one each for the two constructors of *Fin*.
  - In the first equation, we set *fzero* to 0.
  - In the second equation, our argument is of the form $fsuc\ i$. We make a recursive call $toNat\ i$ and increment the result by 1. `Agda`'s termination checker accepts this, as i is structurally smaller than $fsuc\ i$.

4.2. Input Strings and Base64 Decoding

Fin plays a central role as the type of the language alphabet for our X.690 DER and X.509 parsers, as well as the input and output types for Base64 decoding. In general, parser inputs have types of the form $List\ A$, where A is the type of language alphabet; for our X.690 DER and X.509 parsers, this is *UInt8*, an alias for $Fin\ 256$. The ultimate result of our PEM parser is a string of sextets, *i.e.*, a value of type $List\ UInt6$, where *UInt6* is an alias for $Fin\ 64$.

The hand-off between the result of PEM parsing and the input to X.509 parsing (Figure 2, Ⓒ– Ⓓ) is managed by the Base64 decoder, whose formal correctness properties are established with respect to a specificational encoder. Specifically, we prove: (1) that the encoder always produces a result accepted by the decoder; and (2) the encoder and decoder pair forms an isomorphism between octet strings and valid sextet strings for encoding them. This is summarized below in Figure 4 (definitions omitted), which we now explain.

```
Valid64Encoding : List UInt6 → Set
encode : List UInt8 → List UInt6
decode : (bs : List UInt6) → Valid64Encoding bs → List UInt8
encodeValid : ∀ bs → Valid64Encoding (encode bs)
encodeDecode : ∀ bs → decode (encode bs) (encodeValid bs) ≡ bs
decodeEncode : ∀ bs → (v : Valid64Encoding bs)
                    → encode (decode bs v) ≡ bs
```

Figure 4: Base64 encoding and decoding (types only)

- *Valid64Encoding* is a predicate for sextet strings that expresses what it means for them to be valid encodings of an octet string. Recall that Base64 decoding proceeds by mapping each group of four sextets to three octets (24 bits in total).
  - If a single sextet remains after this grouping, then the sextet string is invalid (6 bits is not enough to encode an 8 bit value).
  - If two sextets remain, then they encode a single octet iff the last 4 bits of the second sextet are set to 0.
  - If three sextets remains, then they encode two octets iff the last 2 bits of the third sextet are set to 0.
- Next in the figure are the encoder, *encode*, and decoder, *decode*. While the domain of the encoder is all octet strings, for the decoder the domain is restricted to only those sextet strings for which the predicate *Valid64Encoding* holds.

- Lemma $encodeValid$ is a proof that the specificational Base64encoder always produces a valid Base64 encoding.
- Finally, our main correctness result for the Base64 module is given by the proofs $encodeDecode$ and $decodeEncode$, which together state that the encoder and decoder form an isomorphism ($\equiv$ is the symbol for propositional equality). In the first direction ($encodeDecode$), we pass to the decoder the result of encoding octet string bs together with a proof that this encoding is valid, and the result we get is the very same octet string bs. In the second direction, we assume that the given sextet string bs is already a valid encoding, and we obtain that the result of first decoding and then re-encoding bs is bs itself.

4.3. Verification of Parsers

We conceptually separate each parser verification task into *language specification*, *language security verification*, and *parser correctness verification*.

4.3.1. Language specification. We provide parser-independent formalizations of the PEM, Base64, X.690 DER, and X.509 formats, greatly reducing the complexity of the specification and increasing trust that they faithfully capture the natural language description. Much current research [19], [22] on applying formal methods to parsing uses serializers to specify their correctness properties. Formal proofs of correctness (in any context) are only ever as good as the specification of those correctness properties, and this earlier research swells the trusted computing base by introducing implementation details for serialization. To avoid this issue, we use *relational* specifications of languages. This has two other advantages: (1) it allows for a clear distinction between correctness properties of the *language* and *parser*; and (2) it brings the formal language specification into closer correspondence with the natural language description. This second point also means the formal specification can serve as a machine-checked, rigorous alternative for the developers seeking to understand the relevant specifications.

The relational specifications we give are of the following form. For a given language G with alphabet A, we define a family of types $G : List\ A \to Set$, where the family G is indexed by strings $xs : List\ A$ over the alphabet. Such a family serves dual roles: a value of type $G\ xs$ is both proof that xs is in the language G, and the internal representation of the value decoded from xs.

We illustrate our approach with a concrete example: our specification of X.690 DER integer values, shown in Figure 5. This specification takes the form of an Agda record that is parameterized by a bytestring bs.

- **Erasure annotations.** The annotation $@0$ marks the accompanying identifier as *erased at runtime*. In $IntegerValue$, only val, which is the integer encoded by bs, is present at runtime; the remaining fields and parameter bs are erased by Agda's GHC backend. Runtime erasure annotations not only improve performance, but also document the components that serve only specificational purposes for programmers using ARMOR as a reference.
- **Minimum representation.** X.690 DER requires the two's complement encoding of an integer value consists of the minimum number of octets. We *express* this property with $MinRep$, which defines a relation between the first byte of the encoding and the remaining bytes. We *enforce* the property with field $minRep$ of $IntegerValue$: in order to construct an expression of type $IntegerValue\ bs$, one must prove that $MinRep$ holds for the head and tail of bs.
  The definition of $MinRep$ is by pattern matching on tl.
  1) If tl is empty, we return the trivially true proposition $\top$, because a single byte is always minimal.
  2) Otherwise, if the first byte is 0, the second byte must be no less than 128; and if the first byte is 255, then the second byte must be no greater than 127.
- **Nonempty encoding.** Fields hd, tl, and bs_{eq} together ensure the encoding of an integer value "consists of one or more octets." [30] Specifically, bs_{eq} ensures that bs is of the form $hd :: tl$, where hd is the first content octet and tl contains the remaining octets (if any).
- **Linking the value and its encoding.** Field val_{eq} enforces that val be populated with a value equal to the result of decoding bs as a two's complement binary value ($Base256.twosComp$ is the decoding operation).

```
MinRep : UInt8 → List UInt8 → Set
MinRep hd [ ] = ⊤
MinRep hd (b₂ :: tl) =
  (hd > 0 ∨ (hd ≡ 0 ∧ b₂ ≥ 128))
  ∧ (hd < 255 ∨ (hd ≡ 255 ∧ b₂ ≤ 127))
record IntegerValue (@0 bs : List UInt8) : Set where
  constructor mkIntVal
  field
    @0 hd : UInt8
    @0 tl : List UInt8
    @0 minRep : MinRep hd tl
    val : ℤ
    @0 val_eq : val ≡ Base256.twosComp bs
    @0 bs_eq : bs ≡ hd :: tl
```

Figure 5: Specification of integer values

4.3.2. Language security verification. A major advantage of our approach to specifying X.509 is that it facilitates proving properties *about the grammar* without having to reason about parser implementation details. We have proven: *unambiguousness* for the supported subsets of formats PEM, X.690 DER, and X.509; *non-malleability* for the supported subsets of formats X.690 DER and X.509; and *unique prefixes* for all $\langle t, \ell, v \rangle$ structures.

Unambiguous. We formally define unambiguousness of a language G in Agda as follows.

```
Unambiguous G = ∀ {xs} → (a₁ a₂ : G xs) → a₁ ≡ a₂
```

Read this as saying for every string xs, any two inhabitants of the internal representation of the value encoded by xs in G are equal. In the context of X.509, format ambiguity could result in interoperability issues between standards-compliant producers and consumers (*e.g.*, rejection because the decoded certificate does not match the encoded certificate).

Challenges. One challenging aspect in proving unambiguousness for X.690 DER its support for sequences with *optional* and *default* fields, that is, fields that might not be present in the sequence. We are threatened with ambiguity if it is possible to mistake an optional field whose encoding is present for another optional field whose encoding is absent. To avoid this scenario, the X.690 format stipulates that every field of any "block" of optional or default fields must be given a tag distinct from every other such field. Our proof of unambiguousness for X.509 relies heavily on lemmas proving the X.509 format obeys this stipulation.

Non-malleable. Informally, in the context of X.509 non-malleability means that two distinct bytestrings cannot be encodings for the same certificate. Compared to unambiguousness, non-malleability requires more machinery to express, so we begin by discussing the challenges motivating this machinery. Since the bytestring encodings are part of *the very types of internal representations*, e.g., $IntegerValue\ xs$, it is impossible to express equality between internal representations $a_1 : G\ xs_1$ and $a_2 : G\ xs_2$ without *already assuming* xs_1 *is equal to* xs_2. Thus, to make non-malleability non-trivial, we must express what is the "raw" internal datatype corresponding to G, discarding the specificational components. We express this with Raw, given below.

```
record Raw (G : List A → Set) : Set where
  field
    D : Set
    to : {@0 xs : List A} → G xs → D
```

An inhabitant of $Raw\ G$ consists of a type D (the "mere data" of G) together with a function to that extracts this data from any inhabitant of $G\ xs$. Consider the case for $IntegerValue$ below.

```
RawIntegerValue : Raw IntegerValue
Raw.D RawIntegerValue = ℤ
Raw.to RawIntegerValue = IntegerValue.val
```

This says that the raw representation for X.690 DER integer values is $\mathbb{Z}$, and the extraction function is just the field accessor $IntegerValue.val$.

Once we have defined an instance of $Raw\ G$, we express non-malleability of G with respect to that raw representation with the following property: give two "proof-carrying" internal representations $g_1 : G\ xs_1$ and $g_2 : G\ xs_2$, if the mere data of g_1 and g_2 are equal, then not only are strings xs_1 and xs_2 equal, but also g_1 and g_2. In Agda, we write:

```
NonMalleable : {G : List A → Set} → Raw G → Set
NonMalleable {G} R =
  ∀ {@0 xs₁ xs₂} → (g₁ : G xs₁) (g₂ : G xs₂)
  → Raw.to R g₁ ≡ Raw.to R g₂ → (xs₁ , g₁) ≡ (xs₂ , g₂)
```

Proving $NonMalleable\ RawIntegerValue$ requires proving $Base256.twosComp$ is injective.

Unique prefixes The final language property we discuss, dubbed *"unique prefixes,"* expresses that a language permits parsers no degrees of freedom over which prefix of the input it consumes. Striving for *parser independence*, we formulate this property as follows: for any two prefixes of an input string, if both prefixes are in the language G, then they are equal. In Agda, we express this as $UniquePrefixes$ below.

```
UniquePrefixes G = ∀ {xs₁ ys₁ xs₂ ys₂}
  → xs₁ ++ ys₁ ≡ xs₂ ++ ys₂ → G xs₁ → G xs₂ → xs₁ ≡ xs₂
```

Given that X.509 uses $\langle t, \ell, v \rangle$ encoding, it is unsurprising that we are able to prove $UniquePrefixes$ holds. However, we call explicit attention to this property for two reasons: (1) it is an essential lemma in our proof of *strong completeness* of our X.509 parser (see Section 4.3.3); and (2) this property *does not hold* for the PEM format due to leniency in end-of-line encoding, so to show strong completeness for PEM parsers we need an additional property, *maximality*.

4.3.3. Parser correctness. We now describe our approach to verifying parser soundness and completeness. For a language G, parser *soundness* means every prefix it consumes is in the language, and *completeness* means if a string is in the language, it consumes a prefix of it (we later show a strengthening of this notion of completeness). Our approach to verifying these is to make our parsers *correct-by-construction*, meaning that parsers do not merely indicate success or failure with *e.g.* an integer code, but return *proofs*. Precisely, our parsers are correct-by-construction by being proofs that membership of an input's prefix in G is decidable: parsers return either a proof that some prefix of its input is in the language, or a proof that *no* prefix is.

Correct-by-construction parsers: Our first step is to formally define success. In first-order logic, we would express the condition for the parser's success on a prefix of xs as $\exists ys\ zs, xs = ys +\!\!+ zs \wedge G\ ys$. That is to say, on success the parser consumes some prefix of the input string that is in the language G. In Agda, we express this as the record $Success$, shown below. In the definition, parameters G and xs are the

```
record Success
  (G : List UInt8 → Set) (@0 xs : List UInt8) : Set where
  field
    @0 prefix : List UInt8
    suffix : List UInt8
    @0 ps_eq : prefix ++ suffix ≡ xs
    value : G prefix
```

Figure 6: Success conditions for parsing

language denoted by a production rule and an input string, respectively. The fields of the record are: $prefix$, the consumed prefix of the input (erased at runtime); $suffix$, the remaining suffix of the input from which we parse subsequent productions; ps_{eq}, which relates $prefix$ and $suffix$ to the input string xs (also runtime erased); and $value$, which serves dual roles as both the internal data representation of the value encoded by $prefix$ **and** a proof that $prefix$ is in the language G. As a consequence, *soundness will be immediate*.

Failure is the negation of the success condition, $\neg$ *Success* G xs, meaning *no* prefix of the input xs is in the language of G. To have the parser return *Success* G xs on success and $\neg$ *Success* G xs on failure, we use the `Agda` standard library datatype *Dec*, shown below.

```
data Dec (Q : Set) : Set where
  yes : Q → Dec Q
  no : ¬ Q → Dec Q
```

Reading *Dec* as programmers, *Dec* Q is a tagged union type which can be populated using either values of type Q or type $\neg$ Q; as mathematicians, we read it as the type of proofs that Q is *decidable*. Expressed as a formula of FOL, *Dec* Q is simply $Q \vee \neg Q$; however, note that constructive logic (upon which `Agda` is based) does not admit LEM, so this disjunction must be proven on a case-by-case basis for each Q (there are some undecidable propositions).

We are now able to complete the definition of the type of parsers, shown in Figure 7. *Parser* is a family of types,

```
Parser : (List A → Set) → Set
Parser G = ∀ xs → Dec (Success G xs)
MaximalSuccess : ∀ (G : List A → Set) xs
                 → Dec (Success G xs) → Set
MaximalSuccess G xs (no _) = ⊤
MaximalSuccess G xs (yes s) = ∀ pre suf → pre ++ suf ≡ xs
  → G pre → length pre ≤ length (Success.prefix s)
record MaximalParser (G : List A → Set) : Set where
  field
    p : Parser G
    max : ∀ xs → MaximalSuccess (p xs)
```

Figure 7: Definition of *Parser* and *MaximalParser*

where for each language G, the type *Parser* G is the proposition that, for all bytestrings xs, it is decidable whether some prefix of xs is in G.

Challenges. The guarantee that, on failure, the parser returns $\neg$ *Success* G xs is very strong, as it asserts a property concerning *all possible prefixes* of the input. This strength is double-edged: while having this guarantee makes proving completeness straightforward, *proving* it means ruling out all possible ways in which the input could be parsed. In some cases, we implemented parsers to facilitate the proofs concerning the failure case, at a cost to performance. The clearest example of such a trade-off is in our parsers for X.690 `Choice` values, implemented using back-tracking.

Maximal parsers: The PEM format does not enjoy the *unique prefixes* property. To facilitate our implementation of correct-by-construction PEM parsers and prove a stronger completeness result, we have augmented the specifications of these parsers to guarantee they consume *the largest prefix of the input compliant with the format.* The formalization of this in Agda is shown in Figure 7. Definition *MaximalSuccess* expresses that if parsing xs was successful (*yes* s), then any other prefix *pre* of xs in G is no greater than that consumed by the parser. In the record *MaximalParser*, we couple together a parser p together

```
Sound : (G : List A → Set) → Parser G → Set
Sound G p =
  ∀ xs → (w : IsYes (p xs)) → G (Success.prefix (toWitness w))
Complete : (G : List A → Set) → Parser G → Set
Complete G p = ∀ xs → G xs → IsYes (p xs)
soundness : ∀ {G} → (p : Parser G) → Sound G p
soundness p xs w = Success.value (toWitness w)
trivSuccess : ∀ {G} {xs} → G xs → Success G xs
completeness : ∀ {G} → (p : Parser G) → Complete G p
completeness p xs inG = fromWitness (p xs) (trivSuccess inG)
```

Figure 8: Parser soundness and completeness

with a proof *max* that, for *every* input string xs, if p is successful parsing xs then that success is maximal.

Correctness properties: We now show our formal definitions and proofs of soundness and completeness for parsing, beginning with soundness.

Soundness. The `Agda` definition and proof of soundness for all of our parsers is shown in Figure 8. Beginning with *Sound*, the predicate expressing that parser p is sound with respect to language G, the predicate *IsYes* (definition omitted) expresses the property that a given decision (in this case, one of type *Dec* (*Success* G xs)) is affirmative (i.e., constructed using *yes*). The function *toWitness*:$\forall$ $\{$ Q $\}$ $\{$ d: *Dec* Q $\}$ $\rightarrow$ *IsYes* d $\rightarrow$ Q takes a decision d for proposition Q and proof that it is affirmative, and produces the underlying proof of Q. Thus, we read the definition of *Sound* G p as: "for all input strings xs, if parser p accepts xs, the prefix it consumes is in G."

The proof *soundness* states that *all parsers are sound*. As our parsers are correct-by-construction, the definition is straightforward: we use *toWitness* to extract the proof of parser success, i.e., an expression of type *Success* G xs, then the field accessor *Success.value* obtains the desired proof that the consumed prefix is in G.

Completeness. Figure 8 also shows our definition and proof of *completeness* in `Agda`. The definition of *Complete* directly translates our notion of completeness: for every input string xs, if xs is in G, then parser p accepts some prefix of xs. For the proof, a straightforward lemma *trivSuccess* (definition omitted) states any proof that xs is in G can be turned into a proof that some prefix of xs (namely, xs itself) is in G. With this lemma, the proof of *completeness* uses the function *fromWitness*:$\{$ Q:*Set* $\}$ $\rightarrow$ (d:*Dec* Q) $\rightarrow$ Q $\rightarrow$ *IsYes* d, which intuitively states that if a proposition Q is true, then any decision for Q must be in the affirmative.

Strong completeness. In isolation, completeness does not rule out all bad behavior that threatens security. Specifically, it does not constrain the parser's freedom over (1) which prefix it consumes and (2) how the internal datastructure is constructed. As discussed in Section 4.3.1, these should be thought of as *language* properties. To rule out both bad behaviors, it suffices that G satisfies the properties *Unambiguousness* and *UniquePrefixes*.

Figure 9 shows the types used in our proof of our strong completeness (see the full paper for more).

```
StronglyComplete : (G : @0 List A → Set) → Parser G → Set
StronglyComplete G p = ∀ xs → (inG : G xs)
  → ∃ (w : IsYes (p xs)) (let s = toWitness w in
       (xs , inG) ≡ (Success.prefix x , Success.value s))
strongCompleteness
  : ∀ { G } → Unambiguous G → UniquePrefixes G
    → (p : Parser G) → StronglyComplete G p
strongCompletenessMax : ∀ { G } → Unambiguous G
  → (m : MaximalParser G)
  → StronglyComplete G (MaximalParser.p m)
```

Figure 9: Strong completeness (types only)

StronglyComplete G p says that, if we have a proof *inG* that *xs* is in *G*, then not only does there exist a witness *w* that the parser accepts some prefix of *xs*, but this prefix is *xs* and the proof it returns is *inG*. Recall that the assumption *inG* and the *value* field of the *Success* record serve dual roles: they are not only proofs that a string is in a language, but also the internal data representation of the value encoded by *xs*. So, saying they are equal means the internal representations are equal.

Strong completeness from maximality. For PEM, even though the format lacks the *unique prefixes* property we can still prove strong completeness by leveraging the fact that our parsers are guaranteed to be *maximal*. Intuitively, this is because if *xs* is in *G*, then the largest possible prefix of *xs* in *G* is *xs* itself. We show the formal statement of the theorem in Figure 8 (proof omitted).

4.4. Verification of Chain Builder

The section presents the *Chain builder*, for which we have proven soundness and completeness with respect to a partial specification. Adhering to our discipline of providing high-level, relational specifications, we dedicate the bulk of this section to describing the specification used, presenting at the end the type of our sound-by-construction chain builder and its proof of completeness.

4.4.1. *Chain* Specification. Our operative definition of correctness for the *Chain Builder* module is as follows (cf. RFC 5820, Section 6.1). Given a list of certificates $c_1 \dots c_n$ where $n \geq 2$, this list forms a chain when:

- c_1 is the certificate to be validated;
- c_n is a certificate in the trusted root store;
- for all $i \in \{1 \dots n-1\}$, the issuer field of c_i matches the subject field of c_{i+1}; and
- if c_1 is not a self-signed certificate that is present in the trusted root store, then for all $i, j \in 1 \dots n$, if $c_i = c_j$ then $i = j$.

Note that it is the *Semantic validator* that checks whether the certificate validity period contains the current time, that cryptographic signature verification is outsourced to external libraries (see Section 6), and that we currently perform no policy mapping. Thus, our specification is *partial* in the sense that we do not claim it captures the full set of desired correctness properties of chain building.

```
_IsIssuerFor_ : ∀ { @0 xs₁ xs₂ } → Cert xs₁ → Cert xs₂ → Set
issuer IsIssuerFor issuee =
  NameMatch (Cert.getIssuer issuee) (Cert.getSubject issuer)
_IsIssuerFor_In_ : ∀ { @0 xs₁ xs₂ } → Cert xs₁ → Cert xs₂
                 → (certs : List (∃ Cert)) → Set
issuer IsIssuerFor issuee In certs =
  issuer IsIssuerFor issue ∧ (-, issuer) ∈ certs
removeCertFromCerts : ∀ { @0 xs } → Cert xs
                    → List (∃ Cert) → List (∃ Cert)
removeCertFromCerts cert certs = filter (λc → c ⁇∉ (-, cert)) certs
data Chain (trust candidates : List (∃ Cert))
  : ∀ { @0 xs } → Cert xs → Set where
  root : ∀ { @0 xs₁ xs₂ } { c₁ : Cert xs₁ } (c₂ : Cert xs₂)
       → c₂ IsIssuerFor c₁ In trust
       → Chain trustedRoot candidates c₁
  link : ∀ { @0 xs₁ xs₂ } (issuer : Cert xs₁) { c : Cert xs₂ }
       → issuer IsIssuerFor c In candidates
       → Chain (removeCertFromCerts issuer trust)
               (removeCertFromCerts issuer candidates)
               issuer
       → Chain trust candidates c
```

Figure 10: Definition of a sound *Chain*

Figure 10 lists our formalization of the specification for a sound chain, defined as *Chain*, which we now describe.

- *_IsIssuerFor_* is a binary relation on certificates expressing that the subject field of the first certificate matches the subject of the second. In `Agda`, one can define mixfix operators and relations by using underscores in the identifier to mark the locations of arguments. This allows us to write *issuer IsIssuerFor issuee* as syntactic sugar for *_IsIssuerFor_ issuee issuer*.
- The three-place relation *_IsIssuerFor_In_* augments the previous relation by allowing us to track *where* the issuer came from using the membership relation *_ ∈ _*.
  - In the signature of *_IsIssuerFor_In_*, the type of the third argument, *List (∃ Cert)*, is the type of *lists of tuples* of byte strings *xs : List UInt8* together with proofs *Cert xs* that the byte string encodes a certificate.
  - In the definition of *_IsIssuerFor_In_*, since *certs* is a list of tuples, to express that *issuer* is present in certs we must tuple it together with its octet string encoding. This is neatly achieved with *(-, issuer)*, which forms a tuple where only the second component need by passed explicitly, leaving `Agda` to infer the value of the first component.
- Function *removeCertFromCerts* takes a certificate *cert* and list of tupled certificates *certs* and uses the `Agda` standard library function *filter* to remove all certificates from *certs* that are equal to *cert*.
- Finally, we come to the definition of *Chain*, an inductive family of types indexed by: *trust : List (∃ Cert)*,

```
toList : ∀ {trust candidates} {@0 xs} (c : Cert xs)
       → Chain trust candidates c → List (∃ Cert)
toList c (root issuer _) = (-, c) :: [ issuer ]
toList c (link issuer isIn chain) = (-, c) :: toList issuer chain

ChainUnique : ∀ {trust candidates} {@0 xs} {c : Cert xs}
  → Chain trust candidates c → Set
ChainUnique c = List.Unique (toList c)

chainUnique
  : ∀ trust candidates {@0 xs} {issuee : Cert xs}
    → (-, issuee) ∉ candidates → (-, issuee) ∉ trust
    → (c : Chain trust candidates issuee) → ChainUnique c
```

Figure 11: Chain Uniqueness

```
buildChains
  : ∀ trust candidates {@0 bs} (issuee : Cert bs)
    → List (Chain trust candidates issuee)

ChainEq : ∀ {trust candidates} {@0 bs} {issuee : Cert bs}
        → (c₁ c₂ : Chain trust candidates issuee) → Set
ChainEq c₁ c₂ = toList c₁ ≡ toList c₂

buildChainsComplete
  : ∀ trust candidates {@0 bs} (issuee : Cert bs)
    → (c : Chain trust candidates issuee)
    → Any (ChainEq c) (buildChains trust candidates issuee)
```

Figure 12: Verified chain builder

the certificates in the trusted root store; *candidates* : *List* (∃ *Cert*), the intermediate CA certificates provided by the end entity to facilitate chain building; and the certificate we are attempting to authenticate. *Chain* has two constructors, axiomatizing the two ways we can extend trust to the end entity.

- Constructor *root* expresses that we can trust certificate c_1 when we can find a certificate c_2 in the trusted root store representing an issuer for c_1.
- Constructor *link* expresses that we can trust certificate c if we can find an issuer's certificate *issuer* in *candidates*, and furthermore that we (inductively) trust *issuer* through the construction of a *Chain*. To avoid duplicate certificates in the chain (and ensure termination by ruling out cycles), the chain of trust extended to *issuer* must use a trusted root store and candidate certificate list from which *issuer* has been removed; we express this using function *removeCertFromCerts*.

4.4.2. Chain Uniqueness. As we did with our language formalizations, by having an implementation-independent, relational specification *Chain* we can prove that certain properties hold of *all* chains constructed by our chain builder, *without* reasoning about its implementation details. Given the limited scope of our specification of correctness for chains, we are primarily interested in verifying the *uniqueness* property: "A certificate MUST NOT appear more than once in a propsective certification path." We are able to verify this property under the assumption that the end entity certificate is neither in the candidate list (ensured by a preprocessing step before the *Chain Builder* is invoked) nor in the trusted root store.

The specification and proof of chain uniqueness are listed in Figure 11, which we now describe.

- Function *toList* extracts the list of certificates from the chain, including the issuer found in the trusted root.
- Predicate *ChainUnique* expresses the uniqueness of each certificate in a chain by first using *toList* to extract the underlying list of certificates, then uses the predicate *List.Unique* from `Agda`'s standard library.
- Finally, the proof *chainUnique* (definition omitted) establishes that the predicate *ChainUnique* holds for every chain *c*:*Chain trust candidates issuee*, provided that *issuee* is not present in either the candidate certificate list or the trusted root.

4.4.3. Sound and Complete Chain Building. We now present our chain builder, verified sound and complete with respect to the specification *Chain*, in Figure 12. First, observe that by its type *buildChains* (definition omitted) is *sound by construction*: every chain that it returns has type *Chain trust candidates issuee*. Of course, the *trivial* chain builder (one that always returns the empty list) is also sound by construction, and so the other property we are interested in is *completeness:* if there *exists* a chain of trust extending to the *issuee* from the *trust* store using intermediate certificates pulled from *candidates*, then our chain builder enumerates it. This is formalized in the remainder of the figure, which we now describe.

- Relation *ChainEq* expresses that the underlying certificate lists of two chains c_1 c_2 : *Chain trust candidates issuee* are equal. Observe that were we to define *ChainEq* c_1 c_2 as $c_1 \equiv c_2$, this would be much stronger than is required: a value of type *Chain trust candidates issuee* carries with it not only the underlying certificate list, but also proofs relating each certificate with the next and with *trust* and *candidates*. It is not necessary that these proof terms are also equal, so *ChainEq* discards these using *toList*.
- In the type signature of *buildChainsComplete*, we use *Any* from the `Agda` standard library *Any*. Given any type T, a predicate $Q : T \to Set$ and a list $xs : List\ T$, *Any* Q xs is the proposition that there exists *some* element of xs for which Q holds.
- Putting these together, we can read the type signature of *buildChainsComplete* as follows: for every chain c : *Chain trust candidates issuee*, there exists a chain in the result of *buildChains trust candidates issuee* which is equal to c modulo some proof terms (i.e., the proofs that issuers are present in either *candidates* or *trust* and the proofs that for each adjacent pair of certificates, the issuer of the first matches the subject of the second).

4.5. Verification of Semantic Validator

We now describe our verification approach to the task of *semantic validation*. The checks performed by the *Semantic validator* are separated into two categories: those that apply to a single certificate, and those that apply to a candidate certificate chain. For each property to validate, we formulate in `Agda` a predicate expressing satisfaction of the property by a given certificate or chain, then prove that these predicates are decidable (*Dec*, Section 4.3.3). In what follows, we illustrate with two relatively simple concrete examples: one predicate for a single certificate and one for a certificate chain.

Before we illustrate with examples, we stress that the purpose of this setup is *not* merely to give decidability results for the semantic checks of the X.509 specification, as this fact is intuitively obvious. Instead, and just like with our approach to verified parsing, formulating these semantic checks as decidability proofs (1) *forces* us formalize the natural language property we wish to check *independently of the code that performs the checking,* and (2) *enables* us to write the checking code that is *correct-by-construction*, as these decidability proofs are themselves the very functions called after parsing to check whether the certificate or chain satisfies the property in question.

Single certificate property. For a given certificate, it must be the case that its `SignatureAlgorithm` field contains the same algorithm identifier as the `Signature` field of its `TBSCertificate` (R1 in Table 7 of the Appendix). As a formula of FOL, we could express this property with respect to certificate c as

$$\forall s_1\ s_2, SignAlg(s_1, c) \wedge TBSCertSignAlg(s_2, c) \implies s_1 = s_2$$

where $SignAlg(s_1, c)$ and $TBSCertSignAlg(s_2, c)$ express respectively the properties that s_1 is the signature algorithm identifier of c and that s_2 is the signature algorithm identifier of the `TBSCertificate` of c. In `Agda`, we express this property, and the type of its corresponding decidability proof, as follows (we omit the proof for space considerations).

```
R1 : ∀ {@0 bs} → Cert bs → Set
R1 c = Cert.getTBSCertSignAlg c ≡ Cert.getCertSignAlg c
r₁ : ∀ {@0 bs} (c : Cert bs) → Dec (R1 c)
r₁ c = ...
```

The predicate *R1* expresses that the two signature algorithm fields are equal using the binary relation ≡, which is defined in `Agda`'s standard library. Compared to the proof r_1, *R1* is relatively simple: ≡ is *parametric* in the type of the values it relates (meaning it uses no specifics about the *SignAlg* type family), and is defined as the smallest reflexive relation. In contrast, the checking code r_1 *must* concern itself with the specifics of *SignAlg*. In X.509, signature algorithm fields are defined as a pair where the first component is an object identifier (OID) and the second is an optional field for parameters whose type *depends upon that OID*. So, to implement r_1 we must prove equality is decidable for OIDs *and* for all the signature algorithm parameter types we support.

Certificate chain property.

```
IsConfirmedCA : ∀ {@0 bs} → Cert bs → Set
isConfirmedCA? : ∀ {@0 bs} (c : Cert bs) → Dec (IsConfirmedCA c)
R23 : ∀ {trust candidates} {@0 bs} (issuee : Cert bs)
      → Chain trust candidates issuee → Set
R23 issuee c = All (IsConfirmedCA ∘ proj₂) (tail (toList c))
r23 : ∀ {trust candidates} {@0 bs} (issuee : Cert bs)
      → (c : Chain trust candidates issuee) → Dec (R23 c)
r23 c = All.all? (isConfirmedCA? ∘ proj₂) (tail (toList c))
```

Figure 13: Semantic check for R23

For a certificate chain, it must be the case that every issuer certificate is a CA certificate. Specifically, RFC 5280 (Section 6.1.4) makes the following requirement for issuer certificates:

> *"If certificate i is a version 3 certificate, verify that the basicConstraints extension is present and that cA is set to TRUE. (If certificate i is a version 1 or version 2 certificate, then the application MUST either verify that certificate i is a CA certificate through out-of-band means or reject the certificate. Conforming implementations may choose to reject all version 1 and version 2 intermediate certificates.) "*

In ARMOR, we take the approach suggested in the last line of the quote (see entry R19 of Table 7 in the Appendix), so our task reduces to checking that for each issuer certificate, the `basicConstraints` extension is present and its `cA` field is set to true.

We formalize this semantic condition, listed as R23 in Table 7 in Figure 13. Predicate *IsConfirmedCA* (definition omitted) expresses the condition that the `basicConstraints` extension is present in a certificate with field `cA` set to *true*, and function *isConfirmedCA?* (definition omitted) is the correct-by-construction implementation of that check. Predicate *R23* is extends this property to all issuer certificates of a chain.

- The `Agda` standard library definition *All* is to *Any* (see Section 4.4.3) what ∀ is to ∃. Given a predicate $Q : A \to Set$ and a list $xs : List\ A$, $All\ Q\ xs$ is the proposition that every element of xs satisfies Q.
- The list we are concerned with in predicate *R23* is every certificate in the chain except the first (i.e., the end entity). This is expressed by $tail\ (toList\ c) : List\ (\exists\ Cert)$.
- Since the elements of this list are *tuples* of type $\exists\ Cert$ (where the first component is an octet string and the second is a proof that string encodes a certificate), we form the predicate supplied to *All* by precomposing *IsConfirmedCA* with $proj_2 : (c : \exists\ Cert) \to Cert\ (proj_1\ c)$.

Finally, the sound-by-construction checker for this semantic condition is *r23*, which is defined using *All.all?*, defined in the `Agda` standard library. *All.all?* takes a decision procedure that applies to a single element (in this case, $isConfirmedCA? \circ proj_2$) and returns a decision procedure that decides whether the predicate holds for *all* elements of the given list.

5. Implementation

Driver and Input Interface. ARMOR's driver module is developed using `Python` and `Agda`. The `Python` component is responsible for the user interface, handling inputs such as certificates in DER or PEM formats, trusted CA certificates in PEM format, and optionally the intended purpose of the end-user certificate (e.g., Server Authentication, Client Authentication, Code Signing). After receiving these inputs, the `Python` driver invokes the `Agda` component. On the `Agda` side, the current time is read directly from the system. This component then invokes the parsers, builds the candidate certificate chains, and conducts semantic validation. Finally, it returns a verdict along with some parsed information (*i.e.*, KeyUsage purposes, TBSCertificate bytes, SignatureValue bytes, SignatueAlgorithm) to the `Python` side, which performs signature verification and checks the consistency of the specified purpose in the end-user certificate. The final result of chain validation is then output by the `Python` component.

Chain Building and String Canonicalization. After parsing, we use the chain builder module to build all candidate chains for semantic validation. For ease of formal verification, we first create all candidate chains and then check each of their legitimacy, terminating when we have either identified one such chain, or exhausted all candidates. Our chain builder module uses name matching, instead of using AKI (Authority Key Identifier) and SKI (Subject Key Identifier) extensions as these may not be present in an input certificate. For name matching as part, we normalize the names using LDAP StringPrep profile described in RFC 4518 [27]. Our chain building module's total correctness ensures that we consider all potential chains, the chains all start with a CA certificate in the root store, and the chain builder terminates.

Semantic Validation. For semantic validation, we consider a total of 27 rules. The complete list is provided in Table 7 of the Appendix. The first 18 rules (R1 - R18) are applicable to individual certificates in a chain, whereas the last 9 rules (R19 - R27) are for a chain of certificates. Note that the rules from R1 to R25 are implemented in `Agda` while R26 (signature verification) and R27 (certificate purpose check) are enforced by the `Python` side of driver module. Also, R24 (subject and issuer name chaining) and R25 (trust anchor check) are not explicitly enforced by the semantic validator since these checks are already enforced by the chain builder.

Signature Verification. We currently support only RSA signature verification, primarily because our analysis of the 1.5 billion Censys [45] certificates finds that 96% of certificates employ RSA public keys. However, the RSA signature verification process requires specific cryptographic operations: calculating modular exponentiation over the SignatureValue field, computing hash of TBSCertificate, and the execution of the actual verification process. Given that we do not model or verify cryptography in `Agda`, we use `Python`'s cryptography library for doing *modular exponentiation*. However, for high-assurance, we also utilize HACL* [46] and Morpheus [33]. HACL* is a formally-verified cryptographic library developed in F^* and compiled down to C. In ARMOR, we specifically utilize HACL*'s *hash function* implementations. In contrast, Morpheus is a formally verified implementation of the RSA $PKCS\#1 - v1.5$ [47] signature verification. Morpheus checks the correctness of the signature format after performing the modular exponentiation of the SignatureValue using the public exponent of the certificate issuer's RSA public key, avoiding signature forgery attacks [40].

Supported Extensions. Currently, ARMOR supports 14 certificate extensions for parsing: Basic Constraints, Key Usage, Extended Key Usage, Authority Key Identifier, Subject Key Identifier, Subject Alternative Name, Issuer Alternative Name, Certificate Policy, Policy Mapping, Policy Constraints, Inhibit anyPolicy, CRL Distribution Points, Name Constraints, and Authority Information Access. These extensions are selected based on their frequency of occurrence in practice, providing a comprehensive coverage for the most common scenarios encountered in certificate parsing [21]. When any other extension is present, our parser only consumes the corresponding bytes of the extension and continues parsing rest of the certificate fields. Our supported semantic validation rules are spread across the following 5 extensions: Basic Constraints, Key Usage, Extended Key Usage, Subject Alternative Name, and CRL Distribution Points. ARMOR rejects any unrecognized critical extensions.

From `Agda` to Executable Binary. `Agda` is primarily used as a proof assistant. However, the `Agda` toolchain can produce executable binaries by first compiling `Agda` code to `Haskell`, then using the `Haskell` compiler GHC [48] to generate an executable.

6. Empirical Evaluation

This section evaluates ARMOR's efficiency, robustness, and applicability in real-world scenarios. Particularly, we conduct differential testing against 11 open-source X.509 implementations to evaluate the performance of ARMOR. We aim to find answers to the following questions.

Q1. Correctness of Specification's Interpretation. How accurate is our interpretation of the specification? This can be shown by comparing the certificate chain validation results of ARMOR with the results from the test libraries.

Q2. Runtime Overhead. What are the execution time and memory consumption overheads of ARMOR during runtime? This is assessed by comparing the corresponding overhead of each test library with those of ARMOR to determine whether ARMOR introduces significant overhead or is comparable with the test libraries.

Q3. Performance as a Drop-in Replacement. How much delay does ARMOR introduce when it is used as a drop-in replacement for certificate chain validation logic in another application? To determine this, two instances of the application are run: one in its normal state and the other with ARMOR integrated. The execution times and outputs of both instances are recorded and compared to assess the impact of ARMOR on performance.

6.1. Experimental Setup

Certificate Datasets. We used four certificate datasets for our experiments on Q1 and Q2: Censys [45], Frankencert [3], OpenSSL [49], and EFF [50]. (1) The Censys is a large-scale certificate repository, from which we took a snapshot of 1.5 billion real certificates in January 2022. We then randomly selected 2 million certificates from this snapshot. As the original dataset contained only leaf certificates, we used the cert-chain-resolver [51] tool to retrieve the associated CA certificates. (2) Our Frankencert dataset contains 1 million synthetic certificates chains generated by the Frankencert fuzzer [3] to mimic bad inputs. (3) The OpenSSL dataset contains $2,242$ DER certificates, which are used as part of OpenSSL's regression testing, each time the library is updated. It includes a comprehensive collection of known ASN.1 vulnerabilities and additional variants created through fuzzing. All certificates in the dataset are intentionally invalid, with some errors becoming apparent not during parsing but during semantic validation. (4) The EFF dataset is part of the SSL Observatory project and is created by attempting TLS handshakes with all accessible IPv4 addresses on port 443 (HTTPS), and recording the received certificates. For our evaluation, we used a subset of this EFF dataset ($12,000$ DER certificates). Note that, among these four datasets, we used Censys and Frankencert for testing the end-to-end certificate chain validation implementations, and OpenSSL and EFF for testing only the DER parsers. Table 2 shows a summary of our datasets.

Test Subjects. Our differential testing with ARMOR involved the latest versions (till June 2023) of 11 open-source X.509 implementations– OpenSSL-v3.1.1 [49], Mbed TLS-v3.4.0 [52], GnuTLS-v3.7.9 [53], BoringSSL-vfips-20220613 [38], MatrixSSL-v4.7.0 [54], WolfSSL-v5.6.2 [55], Sun-v1.20 [56], Certvalidator-v0.11.1 [57], Crypto-v1.21rc2 [58], Bouncy Castle-v1.75 [59], and CERES [21]. Among these, OpenSSL, Mbed TLS, GnuTLS, BoringSSL, MatrixSSL, and WolfSSL are written in `C/C++`, Sun and Bouncy Castle are in `Java`, Certvalidator and CERES are in `Python`, and Crypto is in `Go`. We developed *test harness* for each X.509 implementation, consulting the official documentation of their certificate validation APIs.

Adjustment of System-Time. There is a 1.5 years of time difference between the collection of our Censys certificate dataset and our actual evaluation. Therefore, using these certificate chains directly in the experiment could result in the expiration of many of the certificate chains. To solve this challenge, we implemented a probabilistic approach within our experimental setup. Specifically, for 95% certificate chains (randomly selected), we adjusted the system-time to older dates falling within the validity periods of the leaf certificates. For the remaining 5% cases, we maintained the current system-time. Our time adjustment process is based on the Libfaketime [60] library, which allows modifying the system-time a program sees without having to change the time system-wide. This strategy allowed us to parallelize, even in a Docker [61] environment, and evaluate all the semantic rules, not only navigating the issue of certificate expiration but also ensuring a comprehensive and realistic assessment of the certificate validation process.

Testbed Configuration. To answer $Q1$, we relied on a Linux server with Intel Xeon 2.10 GHz 100 core CPU. We distributed subsets of our certificate chain datasets across those 100 cores, running them simultaneously against the test harnesses. The results of the chain validation were then recorded in files for subsequent manual analysis. For $Q2$, we adopted a different strategy. In an effort to obtain realistic data on the execution time and memory consumption, we used another Linux machine with a 3.1 GHz Intel Core-$i7$ CPU. In this run, we used $100,000$ certificate chains randomly selected from our Censys dataset. Finally, for $Q3$, our approach involves modifying the TLS 1.3 implementation in the BoringSSL library to incorporate ARMOR. This modified version of BoringSSL was then compiled with the Curl tool [39], a popular data transfer utility. Using this setup, the top 1,000 websites from Alexa were visited. To evaluate the impact of the ARMOR integration, these visits were also conducted using the standard (unmodified) BoringSSL implementation, and we compared of execution times and outcomes between the normal and modified cases.

Table 2: Summary of datasets and their usage

Dataset	Censys (PEM)	Frankencert (PEM)	OpenSSL (DER)	EFF (DER)	Alexa's Top Websites
Count	2,000,000	1,000,000	2,242	12,000	100
Experiements	Full Chain Runtime	Full Chain	Parser	Parser	End-to-End Application

6.2. Results

We now present our findings for each dataset.

6.2.1. Censys. Table 3 illustrates the rigorous approach ARMOR takes toward certificate validation compared to most libraries. This is particularly evident in the 'Rej-Acc' column, highlighting instances where ARMOR rejected a certificate chain that some other libraries accepted, and in the 'Acc-Rej' column, highlighting instances where ARMOR accepted a certificate chain that some other libraries rejected. A closer investigation of these discrepancies by ARMOR reveals that they stem from violations of guidelines specified in RFC 5280, indicating ARMOR's adherence to compliance with the specifications. Moreover, ARMOR agrees with most certificate chain validations conducted by the test libraries. In the 'Acc-Acc' and 'Rej-Rej' columns, ARMOR matches the results with almost all test libraries (*i.e.*, $> 99\%$ similarity). Now, we discuss the noncompliance issued found by our experiment on the Censys dataset.

a. Allowing Invalid Serial Number. ARMOR rejected $5,053$ certificate chains because at least one certificate in those chains had 0 as serial number, contrary to the RFC 5280 requirement for a positive integer (violation of R3 of Table 7 in Appendix). This violation is present in all the libraries except CERES.

b. Allowing Invalid CRL Distribution Point. ARMOR rejected 5 certificate chains because they did not enforce a

Table 3: Analysis on validation outcomes of Censys chains

Acc = Accept Rej = Reject Sim = Similarity Diff = Difference

ARMOR **vs Others**	**Acc-Acc**	**Acc-Rej**	**Rej-Acc**	**Rej-Rej**	**Sim**	**Diff**
BoringSSL	1,435,897	0	5,058	5,59,045	99.75%	0.25%
GnuTLS	1,435,897	0	5,058	5,59,045	99.75%	0.25%
MatrixSSL	1,435,897	0	5,058	5,59,045	99.75%	0.25%
Mbed TLS	1,435,897	0	5,058	5,59,045	99.75%	0.25%
OpenSSL	1,435,897	0	5,058	5,59,045	99.75%	0.25%
WolfSSL	1,435,897	0	5,058	5,59,045	99.75%	0.25%
Crypto	1,435,897	0	5,058	5,59,045	99.75%	0.25%
Bouncy Castle	1,430,644	5,253	5,058	5,59,045	99.48%	0.52%
Sun	1,430,644	5,253	5,058	5,59,045	99.48%	0.52%
Certvalidator	1,435,806	91	5,058	5,59,045	99.74%	0.26%
CERES	1,430,629	5,268	0	5,64,103	99.74%	0.26%

semantic restriction on the values presented in CRL Distribution Points of subsequent certificates (violation of R21). This violation is present in all the libraries except CERES.

c. Failure to Build Valid Chain. There are $5,253$ inputs for which Bouncy Castle, Sun, and CERES failed to build any valid certificate chain, indicating the presence of bugs in their chain building algorithms. On a closer look of those inputs, we found that multiple candidate chains can be built from them; however, just one chain is rooted to a trust anchor. Since the input list of certificate did not have the certificate of that trust anchor, these libraries failed to find the trusted path. However, we expect that in such scenarios chain building must prioritize finding a certificate for an issuing CA *in the trusted root store.*

d. No Support for `emailAddress` in `Name`. There were 15 chains CERES rejected, due to parsing failure of the `Name` field, that ARMOR accepted. These certificates contain strings of type IA5String to represent `emailAddress`. Although RFC 5280 recommends new certificates include `emailAddress` in the `Subject Alternative Name` extension, the specification does not prohibit including it in `Name` (see 4.1.2.6 in RFC 5280). ARMOR correctly accepts those certificate chains.

e. No Support for Standard Extension. Certvalidator rejected 91 certificate chains that ARMOR accepted. Upon examination, we found that this discrepancy arises from Certvalidator's lack of support for the `Subject Alternative Name` extension, reporting parsing errors for these chains. However, this is a standard extension documented in RFC 5280. ARMOR supports this extension and does not reject these certificate chains.

Runtime Analysis. Tables 5 and 6 in the Appendix show our execution time and memory consumption analysis of the test libraries during runtime, respectively. Considering the different programming languages in which the libraries are written, `C/C++` libraries (*i.e.*, OpenSSL, GnuTLS, Mbed TLS, WolfSSL, MatrixSSL, BoringSSL) generally exhibit greater efficiency regarding memory usage and execution time. This can be attributed to their low-level access to hardware and memory. However, libraries written in higher-level languages, such as ARMOR and the rest, tend to consume more memory and have longer execution times. We found ARMOR on average takes 2.641 seconds when a certificate chain is accepted and 2.518 seconds when a certificate chain is rejected. In terms of memory consumption, it on average takes 1049 megabytes when a certificate chain is accepted and 1069 megabytes when a certificate chain is rejected. Compared to other libraries, ARMOR's runtime overhead is very large, but still within a practical range.

6.2.2. Frankencert. Surprisingly, we found the Frankencert fuzzer could not generate a single valid certificate chain for our dataset, and all our test libraries rejected those chains for different parsing issues. This highlights a potential limitation of the Frankencert fuzzer in creating valid certificate chains.

6.2.3. OpenSSL and EFF. From the set of 2,242 OpenSSL certificates, ARMOR's DER parser accepted 55 certificates and rejected 2,187 certificates. In contrast, out of 12,000 EFF certificates, it accepted 10,958 certificates and rejected 1,042 certificates. This significant difference in acceptance rates between the two datasets was anticipated because the OpenSSL dataset was primarily composed of intentionally flawed certificates, while the EFF dataset contained real-world certificates. A comparison with other test libraries showed that ARMOR's results were consistent with those libraries. Further manual inspection of both "accepted" and "rejected" cases confirmed that ARMOR's parser correctly enforced syntactic restrictions.

6.3. Evaluation on End-to-End Application

Our findings indicate that both the modified and unmodified versions of the BoringSSL library, when used with Curl, successfully connected to the tested websites. However, there was a noticeable difference in the time taken for these connections. With the modified BoringSSL (which integrated ARMOR), the average time for a visit was 3.45 seconds. In contrast, using the standard, unmodified BoringSSL, the average visit time was significantly shorter, at 0.75 seconds. This shows that the integration of ARMOR into BoringSSL increases the time required for website connections.

7. Discussion

Threat to Validity. Recall that the specification of ARMOR is part of its TCB. Although ARMOR's compliance with its specification is mechanically proven, we cannot in principle guarantee the specification's consistency with the natural language description in RFC 5280. Empirical evaluation with real and synthetic certificate chains did not reveal any inconsistencies, which gives confidence in our interpretation of the RFC's natural language specification. Additionally, ARMOR does not include formal guarantees on its cryptographic operations, instead outsourcing signature verification to external libraries like HACL* and Morpheus. Notably, an attempt to use the formally-verified WhyMP library [62] for *modular exponentiation* proved unsuccessful for some inputs, leading to our reliance on `Python`'s cryptography library for this task.

Room for improvement. Although ARMOR makes a substantial stride towards having a high-assurance implementation of X.509 PKI with formally proven correctness properties, there is still room for improvement before it can be incorporated to an application such as a web browser. As an example, in contrast to existing open-source libraries, ARMOR does not yet support *hostname verification* and *revocation*. Although hostname verification is a critical step towards achieving the desired security guarantees of X.509 PKI, we follow the lead of RFC 5280, in which it is not part of the standard but is left to the application developer. Concerning extensions, we currently do not support the enforcement of Subject key identifier (SKI) and Authority key identifier (AKI) extensions. SKI and AKI can substantially simplify the construction of candidate certificate chains. However, in a recent measurement study on Censys data [21], SKI and AKI are found to be present only on $\sim 95\%$ of the certificates. For generality, we use name matching as our basis of certificate chain building instead of AKI and SKI. Dictated by CA/B forum, browsers often enforce more stringent requirements that are not necessarily warranted by RFC 5280. These additional constraints are currently missing from ARMOR. Finally, to realize the incorporation of ARMOR into a web browser, its overhead must be reduced and it must come with formal guarantees of memory safety. Improving ARMOR in these directions is left as future work.

Lessons learned. ARMOR currently does not feature a formally-verified string canonicalizer. ARMOR's string canonicalizer does not handle bidirectional characters and only supports UTF-8 encoded unicode characters. We, however, observe that *none of the existing libraries* performs this suggested step. Similarly, ARMOR does not yet enforce name constraints and Policy Checking, which are also unsupported by some mainstream libraries. These are only a few examples of features present in RFC 5280 whose complexities make them daunting to implement correctly in practice. Furthermore, some constraints RFC 5280 places on issuers lack clear directions regarding whether *consumers* should reject noncompliance. Overall, we believe that the specification can and should be substantially simplified and streamlined, removing bloat due to historical features (such as the widely unsupported string canonicalization), to ensure improved interoperability and security.

8. Related Work

Extensive research has previously been conducted to test the X.509 CCVL of SSL/TLS libraries using techniques such as fuzzing [3], [4], [5], [6], [7] and symbolic execution [8], [9]. Fuzzing is a popular software testing technique in which malformed inputs are automatically generated and injected into a target application to find implementation flaws [63]. Symbolic execution, on the other hand, is a way of executing a program abstractly so that one abstract execution covers multiple possible inputs of the program that share a particular execution path through the code [64]. Though these approaches found numerous implementation flaws and noncompliance issues, none can avoid false negatives in differential testing due to the lack of a formally-verified reference implementation of X.509 CCVL. Despite several efforts to implement and formally verify cryptographic libraries [46], [65], [66], a formally-verified implementation of X.509 CCVL is still missing from the literature.

Although our work presents a major step to address this research gap, there are other works that motivate our high-assurance implementation. As an example, we rely on the prior re-engineering effort of the X.509 specification and implementation (nqsb-TLS [17], CERES [21], Hammurabi [22]) to distinguish between the syntactic and semantic requirements of X.509 and design ARMOR in a modular way. However, in comparison to ARMOR, these works lack any formal correctness guarantees. Although Ramananandro *et al.* proposed EverParse [19], a framework for generating verified parsers and serializers from Type-Length-Value ($\langle t, \ell, v \rangle$) binary message format descriptions, with memory safety, functional correctness (*i.e.*, parsing is the inverse of serialization and vice versa), and non-malleable guarantees, it only focuses on parsing, and lacks formal correctness guarantees of other stages of the certificate chain validation. Barenghi *et al.* proposed an approach to automatically generate a parser for X.509 with the ANTLR parser generator [18]; however, they do major simplifications of the X.509 grammar to avoid complexities in parsing. Tao *et al.* developed a memory-safe and formally correct encoder for X.509 certificates [20], while our work does the reverse task, *certificate decoding*.

Parallel to our research, some studies have unveiled that the X.509 PKI is intentionally deployed to allow TLS interceptions by antivirus programs, parental control applications, middleboxes, and proxy servers [67], [68], [69], [70], [71]. This intervention disrupts the end-to-end security guarantee that TLS is supposed to provide, posing potential security risks. Furthermore, several studies also underlined a key issue: *user unawareness*. Many users lack a proper understanding of X.509 PKI and TLS, potentially overlooking their browser's certificate-related warnings and, in the worst case helping adversaries compromise users own trust anchors [72], [73], [74], [75], [76], [77].

9. Conclusion

We presented ARMOR, which is an X.509 implementation with formal correctness guarantees. ARMOR distinguishes itself from other research on formally verifying components of X.509 through its broader coverage of the standard and its emphasis on simpler, *relational* specifications to demarcate format and parser correctness properties. Concerning this second point, we argued the philosophical and practical merits of relational specifications over those that involve implementation details, with the upshot being that this approach increases trustworthiness and usefulness of formal verification efforts. We evaluated ARMOR's specificational accuracy by differentially testing it with 11 open-source libraries and observed no inaccuracies. Analysis of

ARMOR's runtime overhead suggests that it is a suitable option for applications where correctness is preferred and overhead can be tolerated. Our experience and analysis leads us to believe the current standard is bloated with historical features and lacks clear directions on enforcing certain constraints, which both impedes formal verification efforts and imposes a high engineering overhead. Streamlining and simplifying the standard can improve the overall standard compliance and correctness of these libraries substantially.

References

[1] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate 5280," Tech. Rep.

[2] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The most dangerous code in the world: validating SSL certificates in non-browser software," in *ACM Computer and Communications Security (CCS)*, 2012, pp. 38–49.

[3] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov, "Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations," in *IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 114–129.

[4] Y. Chen and Z. Su, "Guided differential testing of certificate validation in SSL/TLS implementations," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 793–804.

[5] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana, "Nezha: Efficient domain-independent differential testing," in *IEEE Symposium on Security and Privacy*. IEEE, 2017, pp. 615–632.

[6] L. Quan, Q. Guo, H. Chen, X. Xie, X. Li, Y. Liu, and J. 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*, 2020, pp. 524–535.

[7] C. Chen, P. Ren, Z. Duan, C. Tian, X. Lu, and B. Yu, "Sbdt: Search-based differential testing of certificate parsers in ssl/tls implementations," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 967–979.

[8] C. Tian, C. Chen, Z. Duan, and L. Zhao, "Differential testing of certificate validation in SSL/TLS implementations: An rfc-guided approach," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 4, pp. 1–37, 2019.

[9] S. Y. Chau, O. Chowdhury, E. Hoque, H. Ge, A. Kate, C. Nita-Rotaru, and N. Li, "Symcerts: Practical symbolic execution for exposing noncompliance in X. 509 certificate validation implementations," in *IEEE Symposium on Security and Privacy*. IEEE, 2017, pp. 503–520.

[10] "CVE-2020-14039," https://nvd.nist.gov/vuln/detail/CVE-2020-14039.

[11] "CVE-2016-11086," https://nvd.nist.gov/vuln/detail/CVE-2016-11086.

[12] "CVE-2020-1971," https://nvd.nist.gov/vuln/detail/CVE-2020-1971.

[13] "CVE-2020-35733," https://nvd.nist.gov/vuln/detail/CVE-2020-35733.

[14] "CVE-2020-36229," https://nvd.nist.gov/vuln/detail/CVE-2020-36229.

[15] "CVE-2023-33201," https://nvd.nist.gov/vuln/detail/CVE-2023-33201.

[16] "CVE-2023-40012," https://nvd.nist.gov/vuln/detail/CVE-2023-40012.

[17] D. Kaloper-Meršinjak, H. Mehnert, A. Madhavapeddy, and P. Sewell, "{Not-Quite-So-Broken}{TLS}: Lessons in {Re-Engineering} a security protocol specification and implementation," in *USENIX Security Symposium*, 2015, pp. 223–238.

[18] A. Barenghi, N. Mainardi, and G. Pelosi, "Systematic parsing of x. 509: eradicating security issues with a parse tree," *Journal of Computer Security*, vol. 26, no. 6, pp. 817–849, 2018.

[19] T. Ramananandro, A. Delignat-Lavaud, C. Fournet, N. Swamy, T. Chajed, N. Kobeissi, and J. Protzenko, "Everparse: Verified secure zero-copy parsers for authenticated message formats." in *USENIX Security Symposium*, 2019, pp. 1465–1482.

[20] Z. Tao, A. Rastogi, N. Gupta, K. Vaswani, and A. V. Thakur, "Dice*: A formally verified implementation of dice measured boot." in *USENIX Security Symposium*, 2021, pp. 1091–1107.

[21] J. Debnath, S. Y. Chau, and O. Chowdhury, "On re-engineering the x. 509 pki with executable specification for better implementation guarantees," in *ACM Computer and Communications Security (CCS)*, 2021, pp. 1388–1404.

[22] H. Ni, A. Delignat-Lavaud, C. Fournet, T. Ramananandro, and N. 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*, 2023, pp. 275–289.

[23] I. Rec, "X.509 information technology–open systems interconnection–the directory: Public-key and attribute certificate frameworks," Technical report, ITU, Tech. Rep., 2005.

[24] P. Saint-Andre and J. Hodges, "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)," Tech. Rep., 2011. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc6125

[25] M. Cooper, Y. Dzambasow, P. Hesse, S. Joseph, and R. Nicholas, "Rfc 4158: Internet x. 509 public key infrastructure: Certification path building," 2005.

[26] S. Chokhani, "Internet x.509 public key infrastructure certificate policy and certification practices framework," Tech. Rep., 1999. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc2527

[27] K. Zeilenga, "Rfc 4518: Lightweight directory access protocol (ldap): Internationalized string preparation," Tech. Rep., 2006.

[28] J. Larisch, W. Aqeel, M. Lum, Y. Goldschlag, L. Kannan, K. Torshizi, Y. Wang, T. Chung, D. Levin, B. M. Maggs *et al.*, "Hammurabi: A framework for pluggable, logic-based x. 509 certificate validation policies," in *ACM Computer and Communications Security (CCS)*, 2022, pp. 1857–1870.

[29] J. Yen, R. Govindan, and B. Raghavan, "Tools for disambiguating rfcs," in *Proceedings of the Applied Networking Research Workshop*, 2021, pp. 85–91.

[30] I. Rec, "X.690 Information technology–ASN. 1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)," Technical report, ITU, Tech. Rep., 2002.

[31] D. Kaminsky, M. L. Patterson, and L. Sassaman, "PKI layer cake: New collision attacks against the global X. 509 infrastructure," in *International Conference on Financial Cryptography and Data Security*. Springer, 2010, pp. 289–303.

[32] R. Sleevi, "Path Building vs Path Verifying: The Chain of Pain," Tech. Rep., 2020. [Online]. Available: https://medium.com/@sleevi_/path-building-vs-path-verifying-the-chain-of-pain-9fbab861d7d6

[33] M. Yahyazadeh, S. Y. Chau, L. Li, M. H. Hue, J. Debnath, S. C. Ip, C. N. Li, E. Hoque, and O. Chowdhury, "Morpheus: Bringing the (pkcs) one to meet the oracle," in *ACM Computer and Communications Security (CCS)*, 2021, pp. 2474–2496.

[34] S. Y. Chau, M. Yahyazadeh, O. Chowdhury, A. Kate, and N. Li, "Analyzing semantic correctness with symbolic execution: A case study on pkcs# 1 v1. 5 signature verification," in *Network and Distributed Systems Security (NDSS) Symposium*, 2019.

[35] A. Bove, P. Dybjer, and U. Norell, "A brief overview of agda–a functional language with dependent types," in *Theorem Proving in Higher Order Logics: 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings 22*. Springer, 2009, pp. 73–78.

[36] U. Norell, "Towards a practical programming language based on dependent type theory," Ph.D. dissertation, 2007.

[37] K. Bhargavan, C. Fournet, and M. Kohlweiss, "mitls: Verifying protocol implementations against real-world attacks," *IEEE Security & Privacy*, vol. 14, no. 6, pp. 18–25, 2016.

[38] "BoringSSL," https://boringssl.googlesource.com/boringssl/.

[39] "Curl," https://curl.se/.

[40] H. Finney, "Bleichenbacher's rsa signature forgery based on implementation error," *http://www. imc. org/ietf-openpgp/mail-archive/msg14307. html*, 2006.

[41] D. Bleichenbacher, "Chosen ciphertext attacks against protocols based on the rsa encryption standard pkcs# 1," in *Advances in Cryptology—CRYPTO'98: 18th Annual International Cryptology Conference Santa Barbara, California, USA August 23–27, 1998 Proceedings 18*. Springer, 1998, pp. 1–12.

[42] "Guardedness checker inconsistency with copatterns #1209 ," https://github.com/agda/agda/issues/1209.

[43] "The Agda User Manual (v2.6.2.2," https://agda.readthedocs.io/en/v2.6.2.2/index.html.

[44] M. H. Sørensen and P. Urzyczyn, *Lectures on the Curry-Howard Isomorphism, Volume 149 (Studies in Logic and the Foundations of Mathematics)*. Elsevier Science Inc., 2006.

[45] "Censys," https://censys.com/.

[46] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "Hacl*: A verified modern cryptographic library," in *ACM Computer and Communications Security (CCS)*, 2017, pp. 1789–1806.

[47] K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch, "Pkcs# 1: Rsa cryptography specifications version 2.2," Tech. Rep., 2016.

[48] "The Glasgow Haskell Compiler," https://www.haskell.org/ghc/.

[49] "OpenSSL," https://www.openssl.org/.

[50] "The EFF SSL Observatory." https://www.eff.org/observatory, 2010.

[51] "cert-chain-resolver," https://github.com/zakjan/cert-chain-resolver.

[52] "Mbed TLS," https://www.trustedfirmware.org/projects/mbed-tls/.

[53] "GnuTLS," https://www.gnutls.org/.

[54] "MatrixSSL," https://github.com/matrixssl/matrixssl/.

[55] "wolfSSL," https://www.wolfssl.com/.

[56] "Java," https://www.java.com/en/.

[57] "certvalidator," https://github.com/wbond/certvalidator.

[58] "crypto," https://github.com/golang/crypto.

[59] "The Legion of the Bouncy Castle," https://www.bouncycastle.org/java.html.

[60] "libfaketime," https://github.com/wolfcw/libfaketime.

[61] "Docker," https://www.docker.com/.

[62] G. Melquiond and R. Rieu-Helft, "Whymp, a formally verified arbitrary-precision integer library," in *Proceedings of the 45th International Symposium on Symbolic and Algebraic Computation*, 2020, pp. 352–359.

[63] P. Godefroid, "Fuzzing: Hack, art, and science," *Communications of the ACM*, vol. 63, no. 2, pp. 70–76, 2020.

[64] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[65] B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. Setty, and L. Thompson, "Vale: Verifying {High-Performance} cryptographic assembly code," in *USENIX Security Symposium*, 2017, pp. 917–934.

[66] J. Protzenko, B. Parno, A. Fromherz, C. Hawblitzel, M. Polubelova, K. Bhargavan, B. Beurdouche, J. Choi, A. Delignat-Lavaud, C. Fournet *et al.*, "Evercrypt: A fast, verified, cross-platform cryptographic provider," in *IEEE Symposium on Security and Privacy*. IEEE, 2020, pp. 983–1002.

[67] X. d. C. de Carnavalet and M. Mannan, "Killed by proxy: Analyzing client-end TLS interception software," in *Network and Distributed Systems Security (NDSS) Symposium*, 2016.

[68] Z. Durumeric, Z. Ma, D. Springall, R. Barnes, N. Sullivan, E. Bursztein, M. Bailey, J. A. Halderman, and V. Paxson, "The Security Impact of HTTPS Interception," in *Network and Distributed Systems Security (NDSS) Symposium*, 2017.

[69] L. Waked, M. Mannan, and A. Youssef, "To intercept or not to intercept: Analyzing tls interception in network appliances," in *ACM Computer and Communications Security (CCS)*, 2018, pp. 399–412.

[70] L. S. Huang, A. Rice, E. Ellingsen, and C. Jackson, "Analyzing forged SSL certificates in the wild," in *IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 83–97.

[71] J. Debnath, S. Y. Chau, and O. Chowdhury, "When tls meets proxy on mobile," in *Applied Cryptography and Network Security: 18th International Conference, ACNS 2020, Rome, Italy, October 19–22, 2020, Proceedings, Part II 18*. Springer, 2020, pp. 387–407.

[72] A. Sasse, "Scaring and bullying people into security won't work," *IEEE Security & Privacy*, vol. 13, no. 3, pp. 80–83, 2015.

[73] M. Ukrop, L. Kraus, V. Matyas, and H. A. M. Wahsheh, "Will you trust this tls certificate? perceptions of people working in it," in *Proceedings of the 35th annual computer security applications conference*, 2019, pp. 718–731.

[74] A. P. Felt, R. W. Reeder, H. Almuhimedi, and S. Consolvo, "Experimenting at scale with google chrome's ssl warning," in *Proceedings of the SIGCHI conference on human factors in computing systems*, 2014, pp. 2667–2670.

[75] D. Akhawe and A. P. Felt, "Alice in warningland: A large-scale field study of browser security warning effectiveness." in *USENIX security symposium*, vol. 13, 2013, pp. 257–272.

[76] S. E. Schechter, R. Dhamija, A. Ozment, and I. Fischer, "The emperor's new security indicators," in *IEEE Symposium on Security and Privacy*. IEEE, 2007, pp. 51–65.

[77] J. Sunshine, S. Egelman, H. Almuhimedi, N. Atri, and L. F. Cranor, "Crying wolf: An empirical study of ssl warning effectiveness." in *USENIX Security Symposium*. Montreal, Canada, 2009, pp. 399–416.

Appendix

Table 4: Formal Guarantees

Property	Proven For	Description
$Unambiguous$	PEM, X.690 DER, X.509	One string cannot be the encoding of two distinct values.
$NonMalleable$	X.690 DER, X.509	Two distinct strings cannot be the encoding of the same value.
$UniquePrefixes$	X.690 DER, X.509 ($\langle t, \ell, v \rangle$)	At most one prefix of a string is in the language.
Isomorphism	Base64 decoder	The Base64 decoder forms an isomorphism with a specificational encoder between the set of octet strings and the subset of sextet strings that are valid encodings.
$MaximalParser$	PEM	If the parser consumes a prefix, that prefix is the longest one in the language.
$Sound$ (parser)	PEM, X.690 DER, X.509	If the parser accepts some prefix, that prefix is in the language.
$Complete$ (parser)	PEM, X.690 DER, X.509	If the string is in the language, the parser accepts some prefix of it.
$StronglyComplete$	PEM, X.509	If a string is in the language and encodes value v, the parser consumes *exactly* that string and produces v.
Valid chain	X.509	Our specification $Chain$ for chains consisting of a sequence of n certificates satisfies the following properties by construction: (a) for all $x \in \{1 \ldots n-1\}$, the subject of certificate x is the issuer of certificate $x+1$; (b) certificate 1 is issued by a trusted CA; (c) certificate n is the certificate to be validated
Chain uniqueness	X.509	Under the following assumptions, sequences of certificates satisfying our $Chain$ specification have no duplicates. • The input certificate sequence has no duplicates. • The certificate to be validated is not in the trusted root store.
Sound chain builder	X.509	By construction, the chain builder produces only chains satisfying the specification $Chain$.
Complete chain builder	X.509	The chain builder generates all certificate lists satisfying the specification $Chain$.
Sound semantic checker	X.509	If a certificate or chain passes the semantic checker, it satisfies the semantic properties.
Complete semantic checker	X.509	If a certificate or chain satisfies the semantic properties, it passes the semantic checks.

Table 5: Execution time analysis on Censys chains

S.D. = Standard Deviation

	Accept						Reject					
Library	Count	Min sec	Max sec	Mean sec	Median sec	S.D. sec	Count	Min sec	Max sec	Mean sec	Median sec	S.D. sec
BoringSSL	74,956	0.004	1.119	0.029	0.029	0.009	25,044	0.004	0.340	0.028	0.028	0.006
GnuTLS	74,956	0.004	0.340	0.028	0.028	0.006	25,044	0.001	0.952	0.015	0.014	0.006
MatrixSSL	74,956	0.009	0.257	0.011	0.011	0.003	25,044	0.003	0.065	0.009	0.009	0.004
Mbed TLS	74,956	0.008	0.125	0.009	0.009	0.002	25,044	0.007	0.129	0.009	0.008	0.002
OpenSSL	74,956	0.026	1.014	0.051	0.050	0.011	25,044	0.026	0.491	0.051	0.049	0.011
WolfSSL	74,956	0.006	1.039	0.009	0.009	0.006	25,044	0.007	0.072	0.009	0.008	0.002
Crypto	74,956	0.187	8.891	0.269	0.260	0.101	25,044	0.006	3.484	0.194	0.246	0.138
Bouncy Castle	74,956	0.573	6.019	0.956	0.920	0.382	25,044	0.251	5.714	0.709	0.627	0.219
Sun	74,956	0.129	2.140	0.285	0.271	0.085	25,044	0.147	1.882	0.215	0.194	0.075
Certvalidator	74,951	0.221	2.855	0.269	0.263	0.060	25,049	0.143	1.779	0.254	0.254	0.061
CERES	74,801	0.033	5.735	0.755	0.821	0.338	25,199	0.151	5.621	0.541	0.594	0.263
ARMOR	74,801	2.207	4.553	2.641	2.618	0.118	25,199	0.053	4.665	2.518	2.544	0.300

Table 6: Memory consumption analysis on Censys chains

S.D. = Standard Deviation

	Accept						Reject					
Library	Count	Min mb	Max mb	Mean mb	Median mb	S.D. mb	Count	Min mb	Max mb	Mean mb	Median mb	S.D. mb
BoringSSL	74,956	4.01	4.49	4.21	4.21	0.06	25,044	3.62	4.36	4.13	4.17	0.12
GnuTLS	74,956	8.18	8.82	8.51	8.52	0.13	25,044	4.50	8.57	7.74	8.00	0.91
MatrixSSL	74,956	3.02	3.50	3.31	3.32	0.08	25,044	2.34	3.49	3.17	3.29	0.30
Mbed TLS	74,956	3.82	4.20	3.99	3.98	0.07	25,044	3.80	4.19	4.00	4.01	0.07
OpenSSL	74,956	6.72	7.51	6.90	6.89	0.08	25,044	6.60	7.06	6.87	6.87	0.08
WolfSSL	74,956	7.86	8.61	8.35	8.41	0.17	25,044	8.27	8.58	8.44	8.46	0.06
Crypto	74,956	59.59	68.30	64.41	62.89	2.54	25,044	60.52	68.29	64.10	62.66	2.53
Bouncy Castle	74,956	84.34	130.99	105.79	101.91	8.42	25,044	82.55	119.71	89.96	86.02	6.44
Sun	74,956	47.50	62.83	53.60	53.19	1.19	25,044	44.42	61.52	50.30	49.88	1.86
Certvalidator	74,951	26.67	28.42	27.06	27.04	0.14	25,049	23.90	27.30	26.62	26.79	0.71
CERES	74,801	21.03	40.70	39.08	39.45	2.24	25,199	21.02	31.79	27.03	28.04	3.23
ARMOR	74,801	998	1187	1049	1032	61	25,199	994	1185	1069	1075	135

Table 7: Semantic restrictions enforced by ARMOR

Name	Description
R1	`SignatureAlgorithm` field MUST contain the same algorithm identifier as the `Signature` field in the sequence `TbsCertificate`.
R2	`Extension` field MUST only appear if the `Version` is 3 .
R3	The `Serial` number MUST be a positive integer assigned by the CA to each certificate. Certificate users MUST be able to handle `Serial` number values up to 20 octets.
R4	The `Issuer` field MUST contain a non-empty distinguished name (DN).
R5	If the `Subject` is a CA (*e.g.*, the `Basic Constraints` extension, is present and the value of `CA` is TRUE), then the `Subject` field MUST be populated with a non-empty distinguished name.
R6	`Unique Identifiers` fields MUST only appear if the `Version` is 2 or 3. These fields MUST NOT appear if the `Version` is 1.
R7	Where it appears, the `PathLenConstraint` field MUST be greater than or equal to zero.
R8	If the `Subject` is a CRL issuer (*e.g.*, the `Key Usage` extension, is present and the value of `CRLSign` is TRUE), then the `Subject` field MUST be populated with a non-empty distinguished name.
R9	When the `Key Usage` extension appears in a certificate, at least one of the bits MUST be set to 1.
R10	If subject naming information is present only in the `Subject Alternative Name` extension , then the `Subject` name MUST be an empty sequence and the `Subject Alternative Name` extension MUST be critical.
R11	If the `Subject Alternative Name` extension is present, the sequence MUST contain at least one entry.
R12	If the `KeyCertSign` bit is asserted, then the `CA` bit in the `Basic Constraints` extension MUST also be asserted. If the `CA` boolean is not asserted, then the `KeyCertSign` bit in the `Key Usage` extension MUST NOT be asserted.
R13	A certificate MUST NOT include more than one instance of a particular `Extension`.
R14	A certificate-using system MUST reject the certificate if it encounters a critical `Extension` it does not recognize or a critical `Extension` that contains information that it cannot process.
R15	A certificate policy OID MUST NOT appear more than once in a `Certificate Policies` extension.
R16	While each of these fields is optional, a `DistributionPoint` MUST NOT consist of only the `Reasons` field; either `distributionPoint` or `CRLIssuer` MUST be present.
R17	The certificate `Validity` period includes the current time.
R18	If a certificate contains both a `Key Usage` extension and an `Extended Key Usage` extension, then both extensions MUST be processed independently and the certificate MUST only be used for a purpose consistent with both extensions. If there is no purpose consistent with both extensions, then the certificate MUST NOT be used for any purpose.
R19	Conforming implementations may choose to reject all `Version` 1 and `Version` 2 intermediate CA certificates .
R20	The `PathLenConstraint` field is meaningful only if the `CA` boolean is asserted and the `Key Usage` extension, if present, asserts the `KeyCertSign` bit. In this case, it gives the maximum number of non-self-issued intermediate certificates that may follow this certificate in a valid certification path.
R21	For `DistributionPoint` field, if the certificate issuer is not the CRL issuer, then the `CRLIssuer` field MUST be present and contain the Name of the CRL issuer. If the certificate issuer is also the CRL issuer, then conforming CAs MUST omit the `CRLIssuer` field and MUST include the `distributionPoint` field.
R22	A certificate MUST NOT appear more than once in a prospective certification path.
R23	Every non-leaf certificate in a chain must be a CA certificate.
R24	Certificate users MUST be prepared to process the `Issuer` distinguished name and `Subject` distinguished name fields to perform name chaining for certification path validation.
R25	Validate whether the chain starts from a trusted CA.
R26	Validate RSA signatures.
R27	Validate leaf certificate purpose against user's expected certificate purpose.