AERES: Summary of verification architecture

Christa Jenkins

April 7, 2023

1 Operative Notions

- **soundness** If the parser accepts the string, so does the grammar
- **completeness** If the grammar accepts the string, so does the parser
- **secure completeness** If the grammar accepts the string, so does the parser, and there are no two distinct ways for the grammar to accept the string.

  NOTE TO OMAR: I'm not sure if this is good terminology, or even if it is a good idea to group completeness (a relation between the grammar and parser) and uniqueness (a property of the grammar).

2 Overview

1. The Aeres external driver is invoked with the filepath of the certificate chain we wish to check. The driver invokes Aeres with the contents of this file.
2. Aeres uses its verified PEM parser library to parse the PEM certificate chain, then decodes the Base64-encoded certificates into a single bytestring.[1]

   (Sound and complete parsing)
3. Aeres uses its verified X.509 parser library to parse the bytestring into a list of certificates.

   (Sound, complete, secure)

[1] We maybe could have decoded it to a list of bytestrings and parsed each, come to think of it...

4. Aeres then checks several semantic properties not suitable for expressing in the grammar (e.g., validity period of cert contains current time)

5. For each cert, Aeres outputs the bytestring serializations for the TBS certificate, signature, and public key, and also outputs the signature algorithm OIDLeastBytes

6. The external driver verifies the public key signatures.

3 Design (Challenges and Solutions)

3.1 Grammar

Challenge Our first and most fundamental question is: how shall we represent the grammar? Recall that our operative notion of soundness is "if the parser accepts the string, then so does the grammar." We also wish for our formulation of the grammar to serve as a readable formalization of the X.509 and X.690 specification.

Solution In general purpose functional languages, inductive types are a natural choice for expressing the grammar of a language. Our choice of formalizing X.509 and X.680 is *inductive families*, the generalization of inductive types to a dependently typed setting.

Let us consider a simple example: X.690 DER Boolean values. The BER require that Boolean values consists of a single octet with `FALSE` represented by the setting all bits to 0, and the DER further stipulates that `TRUE` be represented by setting all bits to 1. We represent these constraints as follows.

```
module BoolExample where
  data BoolRep : Bool → UInt8 → Set where
    falseᵣ : BoolRep false (UInt8.fromℕ 0)
    trueᵣ : BoolRep true (UInt8.fromℕ 255)
  record BoolValue (@0 bs : List UInt8) : Set where
    constructor mkBoolValue
    field
      v     : Bool
      @0 b : UInt8
      @0 vᵣ : BoolRep v b
      @0 bs≡ : bs ≡ [ b ]
```

1. First, we define a binary relation BoolRep that relates Agda Bool values to the octet values specified by X.690 DER (UInt8.fromℕ converts a

non-negative unbounded integer to its UInt8 representation, provided Agda can verify automatically the given number is less than 256).

2. Next, we define a record BoolValue for the representation of the X.690 Boolean value itself.

   - Each production rule of the grammar, such as BoolValue, is represented by a type family of type @0 List UInt8 → Set, which we interpret as the type of predicates over byte-strings (we will explain the @0 business shortly).
   - The fields of the record are the Boolean value v, its byte-string representation b, a proof of type BoolRep v b that b is the correct representation of b, and a proof that the byte-string representation of this terminal of the grammar is the singleton list consisting of b (written [b])

The @0 annotations on types and fields indicate that the values are *erased at run-time*. We do this for two reasons: to reduce the space and time overhead for executions of Aeres, and to serve as machine-enforced documentation delineating the parts of the formalization that are purely for the purposes of verification.

3.2 Parser

Challenge: Next, we must design the parser. We desire that the parser by sound and complete *by construction*.

Solution: For our parser to be sound, when it succeeds we have it return a proof that the byte-string conforms to grammar. For completeness, when it fails we have it return a *refutation* — a proof that there is no possible way for the grammar to accept the given byte-string. The two of these together are captured nicely by the notion of *decidability*, formalized in the Agda standard library as Dec (we show a simplified, more intuitive version of this type below)

```
module DecSimple where
  data Dec (P : Set) : Set where
    yes : P → Dec P
    no : ¬ P → Dec P
```

Let us examine (a slightly simplified version of) the definition of Parser used in Aeres. Below, module parameter S is the type of the characters of the alphabet over which we have defined a grammar.

```
module ParserSimple (S : Set) where
  record Success (@0 A : List S → Set) (@0 xs : List S) : Set where
    constructor success
    field
      @0 prefix : List S
      read   : ℕ
      @0 read≡ : read ≡ length prefix
      value : A prefix
      suffix : List S
      @0 ps≡ : prefix ++ suffix ≡ xs
  record Parser (M : Set → Set) (@0 A : List S → Set) : Set where
    constructor mkParser
    field
      runParser : (xs : List S) → M (Success A xs)
  open Parser public
```

- We first must specify what the parser returns when it succeeds. This is given by the record Success.
  - Parameter A is the production rule (e.g., BoolValue), and xs is the generic-character string which we parsed. Both are marked erased from run-time
  - Field prefix is the prefix of our input string consumed by the parser. We do not need to keep this at run-time, however for the purposes of length-bounds checking we do keep its length read available at run-time.
  - Field value is a proof that the prefix conforms to the production rule A.
  - Field suffix is what remains of the string after parsing. We of course need this at run-time to continue parsing any subsequent production rules.
  - Finally, field ps≡ relates prefix and suffix to the string xs that we started with, i.e., they really are a prefix and suffix of the input.
- Next, we define the type Parser for parsers.
  - Parameter M is used to give us some flexibility in the type of the values returned by the parser. Almost always, it is instantiated

with Logging ∘ Dec, where Logging provides us lightweight debugging information. Parameter A is, again, the production rule we wish to parse.

- Parser consists of a single field runParser, which is a dependently type function taking a character string xs and returning M (Success A xs) (again, usually Logging (Dec (Success A xs)))

3.2.1 Example

It is helpful to see an example parser.

```
private
  here' = "X690-DER: Bool"

parseBoolValue : Parser (Logging ∘ Dec) BoolValue
runParser parseBoolValue [] = do {- 1 -}
  tell $ here' String.++ ": underflow"
  return ∘ no $ λ where
    (success prefix read read≡ value suffix ps≡) →
      contradiction (++-conicalˡ _ suffix ps≡) (nonempty value)
runParser parseBoolValue (x :: xs) {- 2 -}
  with x ≟ UInt8.fromℕ 0 {- 3 -}
... | yes refl =
  return (yes (success _ _ refl (mkBoolValue _ _ falseᵣ refl) xs refl))
... | no x≢0
  with x ≟ UInt8.fromℕ 255 {- 3 -}
... | yes refl =
  return (yes (success _ _ refl (mkBoolValue _ _ trueᵣ refl) xs refl))
... | no x≢255 = do {- 4 -}
  tell $ here' String.++ ": invalid boolean rep"
  return ∘ no $ λ where
    (success prefix _ _ (mkBoolValue v _ vᵣ refl) suffix ps≡) → !!
      (case vᵣ of λ where
        falseᵣ → contradiction (::-injectiveˡ (sym ps≡)) x≢0
        trueᵣ → contradiction (::-injectiveˡ (sym ps≡)) x≢255)
```

1. When the input string is empty, we emit an error message, then return a proof that there is no parse of a BoolValue for the empty string

   We use Agda's do-notation to sequence our operations

2. If there is at least one character, we check
3. whether it is equal to 0 or 255. If so, we affirm that this conforms to the grammar.
4. If it is not equal to either, we emit an error message then return a parse refutation: to conform to BoolValue, our byte must be either 0 or 255.

3.2.2 Backtracking

Although backtracking is not required to parse X.509, our parser has been implemented with some backtracking to facilitate the formalization. For an example, the X.690 specification for DisplayText states it may comprise an IA5String, VisibleString, VisibleString, or UTF8String. In the case that the give byte-string does not conform to DisplayTex providing a refutation is easier when we have direct evidence that it fails to conform to each of these.

```
private
  here' = "X509: DisplayText"

parseDisplayText : Parser (Logging ∘ Dec) DisplayText
runParser parseDisplayText xs = do
  no ¬ia5String ← runParser (parseTLVLenBound 1 200 parseIA5String) xs
    where yes b → return (yes (mapSuccess (λ {bs} → ia5String{bs}) b))
  no ¬visibleString ← runParser (parseTLVLenBound 1 200 parseVisibleString) xs
    where yes b → return (yes (mapSuccess (λ {bs} → visibleString{bs}) b))
  no ¬bmp ← runParser (parseTLVLenBound 1 200 parseBMPString) xs
    where yes b → return (yes (mapSuccess (λ {bs} → bmpString{bs}) b))
  no ¬utf8 ← runParser (parseTLVLenBound 1 200 parseUTF8String) xs
    where yes u → return (yes (mapSuccess (λ {bs} → utf8String{bs}) u))
  return ∘ no $ λ where
    (success prefix read read≡ (ia5String x) suffix ps≡) →
      contradiction (success _ _ read≡ x _ ps≡) ¬ia5String
    (success prefix read read≡ (visibleString x) suffix ps≡) →
      contradiction (success _ _ read≡ x _ ps≡) ¬visibleString
    (success prefix read read≡ (bmpString x) suffix ps≡) →
      contradiction (success _ _ read≡ x _ ps≡) ¬bmp
    (success prefix read read≡ (utf8String x) suffix ps≡) →
      contradiction (success _ _ read≡ x _ ps≡) ¬utf8
```

4 Complete and Secure Parsing

Completeness of the parser is by construction, and straightforward to explain: given a byte-string, if it conforms to the grammar then the parser accepts the byte-string. The heart of the proof is proof by contradiction (which is constructively valid, since the parser is itself evidence that conformance to the grammar is decidable): suppose the parser rejects a string which conforms to the grammar. Then, this rejection comes with a refutation of the possibility that the string conforms with the grammar, contradicting our assumption.

When it comes to security, we also care that the grammar is *unambiguous*, by which we mean that there is at most one way in which the grammar might be parsed. This is formalized as UniqueParse below

```
Unique : Set → Set
Unique P = (p₁ p₂ : P) → p₁ ≡ p₂

UniqueParse : (List S → Set) → Set
UniqueParse A = ∀ {@0 xs} → Unique (Success A xs)
```

We have a lemma that establishes a sufficient condition for when UniqueParse holds, whose premises are stated only in terms of properties of the grammar itself. These properties are: 1. Any two witness[fn::By which we mean inhabitants of a type, when we interpret that type as a proposition under the Curry-Howard isomorphism] that a given string conforms to the grammar are equal (Unambiguous), and 2. If two prefixes of the same string conform to the grammar, those prefixes are equal (NonNesting)

```
Unambiguous : (A : List S → Set) → Set
Unambiguous A = ∀ {xs} → Unique (A xs)

NonNesting : (A : List S → Set) → Set
NonNesting A =
  ∀ {xs₁ ys₁ xs₂ ys₂}
  → (prefixSameString : xs₁ ++ ys₁ ≡ xs₂ ++ ys₂)
  → (a₁ : A xs₁) (a₂ : A xs₂) → xs₁ ≡ xs₂

module _ {A : List S → Set} (uA : Unambiguous A) (nnA : NonNesting A) where
  @0 uniqueParse : UniqueParse A
  uniqueParse p₁ p₂
  {- = ... -}
```

This finally brings us to the statement and proof of complete and secure parsing.

```
Yes_And_ : {P : Set} → Dec P → (P → Set) → Set
Yes (yes pf) And Q = Q pf
Yes (no ¬pf) And Q = ⊥

CompleteParse : (A : List S → Set) → Parser Dec A → Set
CompleteParse A p =
  ∀ {bs} → (v : Success A bs) → Yes (runParser p bs) And (v ≡_)

module _ {A : List S → Set}
  (uA : Unambiguous A) (nnA : NonNesting A) (parser : Parser Dec A)
  where
  @0 completeParse : CompleteParse A parser
  completeParse{bs} v
    with runParser parser bs
  ... | (yes v') = uniqueParse uA nnA v v'
  ... | no ¬v    = contradiction v ¬v
```

1. We define an auxiliary predicate Yes_And_ over decisions, expressing that the decision is yes and the predicate Q holds for the affirmative proof that comes with it.

2. The predicate CompleteParse is defined in terms of Yes_And_, expressing that if v is a witness that some prefix of bs conforms to A, then the parser returns in the affirmative and the witness it returns is equal to v.

3. We then prove the property CompleteParse under the assumption that A is Unambiguous and NonNesting.

   The proof proceeds by cases on the result of running the parser on the given string.

   - If the parser produces an affirmation, we appeal to lemma uniqueParse.
   - If the parser produces a refutation, we have a contradiction

5 Semantic Checks

Some properties that we wish to verify are not as suitable for formalization as part of the grammar. For example, the X.509 specification requires that the signature algorithm field of the TBS certificate matches the signature algorithm listed in the outer certificate — a highly non-local property. Aeres checks such properties after parsing. For each of these, we first write a specification of the property, then a proof that that property is *decidable*. This proof is itself the function that we call to check whether the property holds, and interpreted as such, it is sound and complete by construction for the same reasons that our parser is.

```
SCP1 : ∀ {@0 bs} → Cert bs → Set
SCP1 c = Cert.getTBSCertSignAlg c ≡ Cert.getCertSignAlg c

scp1 : ∀ {@0 bs} (c : Cert bs) → Dec (SCP1 c)
scp1 c =
  case (proj₂ (Cert.getTBSCertSignAlg c) ≅? proj₂ (Cert.getCertSignAlg c)) ret (const _) of λ whe
    (yes ≅-refl) → yes refl
    (no ¬eq) → no λ where refl → contradiction ≅-refl ¬eq
```