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)

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 nonnegative 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 bytestrings (we will explain the @0 business shortly).
   - The fields of the record are the Boolean value v, its bytestring representation b, a proof of type BoolRep v b that b is the correct representation of b, and a proof that the bytestring 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 runtime.* 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.