1 Model

There are two mathematical models generally used to analyze cryptographic protocols. The first, so-called *computational* model, aims for realism: messages are modeled as sequences of bits, and efficient computations are modeled as probabilistic polynomial-time Turing machines. The second, so-called *symbolic* or *Dolev-Yao* model, aims for simplicity: messages are modeled as abstract syntax, and efficient computations are modeled as derivations in a system of inference rules. While a computational analysis makes apparently weaker assumptions about the capabilities of adversaries and therefore can give stronger confidence in security, a symbolic analysis is still useful while being significantly easier to perform and therefore is often favored in practice.

In this work we will use the symbolic model.

1.1 Message deduction

Fix sets $\mathcal{X} = \{x, \ldots\}$ of literals and $\mathcal{K} = \{k, \ldots\}$ of secret keys. The set of messages $\mathcal{M}$ is generated by the following grammar.

$$m, n, o \quad ::= \quad x \;\mid\; k \;\mid\; \langle m, n \rangle \;\mid\; \sigma_k\,(m)$$

The judgment

$$\Gamma \Vdash m$$

asserts that if an agent knows (e.g. has received and stored) the set of messages $\Gamma = m_0, m_1, \ldots$, then it can efficiently construct the message m. It is defined by the following inference rules.

Hypothesis If an agent knows m, then it can construct m.

$$\frac{}{\Gamma, m \Vdash m}\ \mathsf{hyp}$$

Substitution If an agent can construct m, and if once it knows m it can construct n, then it can construct n.

$$\frac{\Gamma \Vdash m \quad \Gamma, m \Vdash n}{\Gamma \Vdash n}\ \mathsf{subst}$$

Literal introduction An agent can construct any literal x.

$$\frac{}{\Gamma \Vdash x}\ xI$$

Pair introduction If an agent can construct m, and it can construct n, then it can construct the pair of m and n.

$$\frac{\Gamma \Vdash m \quad \Gamma \Vdash n}{\Gamma \Vdash \langle m, n \rangle}\ \langle\rangle\, I$$

Pair elimination If an agent can construct the pair of m and n, then it can construct m, and it can construct n.

$$\frac{\Gamma \Vdash \langle m, n \rangle}{\Gamma \Vdash m}\ \langle\rangle E1 \qquad \frac{\Gamma \Vdash \langle m, n \rangle}{\Gamma \Vdash n}\ \langle\rangle E2$$

Signature introduction If an agent can construct m, and it can construct the secret key k, then it can construct m signed with k.

$$\frac{\Gamma \Vdash k \quad \Gamma \Vdash m}{\Gamma \Vdash \sigma_k(m)}\ \sigma I$$

Signature elimination If an agent can construct m signed with k, then it can construct m.

$$\frac{\Gamma \Vdash \sigma_k(m)}{\Gamma \Vdash m}\ \sigma E$$

1.2 Proving secrecy

Proving derivability is simply a matter of exhibiting a derivation.

Example 1. $k, m, n \Vdash \sigma_k(\langle m, n \rangle)$ is derivable.

Proof.

$$\dfrac{\dfrac{}{k, m, n \Vdash k}\ \mathsf{hyp} \qquad \dfrac{\dfrac{}{k, m, n \Vdash m}\ \mathsf{hyp} \quad \dfrac{}{k, m, n \Vdash n}\ \mathsf{hyp}}{k, m, n \Vdash \langle m, n \rangle}\ \langle\rangle I}{k, m, n \Vdash \sigma_k(\langle m, n \rangle)}\ \sigma I$$

□

Proving *non*-derivability, on the other hand, is not so immediate.

Example 2. $m, n \Vdash \sigma_k(\langle m, n \rangle)$ is not derivable.

Aborted proof attempt. Suppose there exists a derivation of the form

$$\frac{\vdots}{m, n \Vdash \sigma_k(\langle m, n \rangle)}\ ???$$

???

Contradiction. □

How to proceed?

1.2.1 Natural deduction

$$\dfrac{}{\Gamma, m \Vdash m}\ \mathsf{hyp} \qquad \dfrac{\Gamma \Vdash m \quad \Gamma, m \Vdash n}{\Gamma \Vdash n}\ \mathsf{subst}$$

$$\dfrac{}{\Gamma \Vdash x}\ xI$$

$$\dfrac{\Gamma \Vdash m \quad \Gamma \Vdash n}{\Gamma \Vdash \langle m, n\rangle}\ \langle\rangle I \qquad \dfrac{\Gamma \Vdash \langle m, n\rangle}{\Gamma \Vdash m}\ \langle\rangle E1 \qquad \dfrac{\Gamma \Vdash \langle m, n\rangle}{\Gamma \Vdash n}\ \langle\rangle E2$$

$$\dfrac{\Gamma \Vdash k \quad \Gamma \Vdash m}{\Gamma \Vdash \sigma_k\left(m\right)}\ \sigma I \qquad \dfrac{\Gamma \Vdash \sigma_k\left(m\right)}{\Gamma \Vdash m}\ \sigma E$$

1.2.2 Sequent calculus

$$\dfrac{}{\Gamma, m \Vdash m}\ \mathsf{ax} \qquad \dfrac{\Gamma \Vdash m \quad \Gamma, m \Vdash n}{\Gamma \Vdash n}\ \mathsf{cut}$$

$$\dfrac{\Gamma \Vdash n}{\Gamma, m \Vdash n}\ \mathsf{w} \qquad \dfrac{\Gamma, m, m \Vdash n}{\Gamma, m \Vdash n}\ \mathsf{c}$$

$$\dfrac{}{\Gamma \Vdash x}\ xR$$

$$\dfrac{\Gamma \Vdash m \quad \Gamma \Vdash n}{\Gamma \Vdash \langle m, n\rangle}\ \langle\rangle R \qquad \dfrac{\Gamma, m \Vdash o}{\Gamma, \langle m, n\rangle \Vdash o}\ \langle\rangle L1 \qquad \dfrac{\Gamma, n \Vdash o}{\Gamma, \langle m, n\rangle \Vdash o}\ \langle\rangle L2$$

$$\dfrac{\Gamma \Vdash k \quad \Gamma \Vdash m}{\Gamma \Vdash \sigma_k\left(m\right)}\ \sigma R \qquad \dfrac{\Gamma, m \Vdash n}{\Gamma, \sigma_k\left(m\right) \Vdash n}\ \sigma L$$

Theorem 1. *The sequent calculus system is equivalent to the natural deduction system.*

Proof. Translation from sequent calculus to natural deduction:

$$\dfrac{\Gamma, m \Vdash o}{\Gamma, \langle m, n\rangle \Vdash o}\ \langle\rangle L1 \quad \rightsquigarrow \quad \dfrac{\dfrac{\dfrac{}{\Gamma, \langle m, n\rangle \Vdash \langle m, n\rangle}\ \mathsf{hyp}}{\Gamma, \langle m, n\rangle \Vdash m}\ \langle\rangle E1 \quad \dfrac{\Gamma, m \Vdash o}{\Gamma, \langle m, n\rangle, m \Vdash o}\ \mathsf{w}}{\Gamma, \langle m, n\rangle \Vdash o}\ \mathsf{subst}$$

$$\dfrac{\Gamma, n \Vdash o}{\Gamma, \langle m, n\rangle \Vdash o}\ \langle\rangle L2 \quad \rightsquigarrow \quad \dfrac{\dfrac{\dfrac{}{\Gamma, \langle m, n\rangle \Vdash \langle m, n\rangle}\ \mathsf{hyp}}{\Gamma, \langle m, n\rangle \Vdash n}\ \langle\rangle E2 \quad \dfrac{\Gamma, n \Vdash o}{\Gamma, \langle m, n\rangle, n \Vdash o}\ \mathsf{w}}{\Gamma, \langle m, n\rangle \Vdash o}\ \mathsf{subst}$$

$$\dfrac{\Gamma, m \Vdash n}{\Gamma, \sigma_k\left(m\right) \Vdash n}\ \sigma L \quad \rightsquigarrow \quad \dfrac{\dfrac{\dfrac{}{\Gamma, \sigma_k\left(m\right) \Vdash \sigma_k\left(m\right)}\ \mathsf{hyp}}{\Gamma, \sigma_k\left(m\right) \Vdash m}\ \sigma E \quad \dfrac{\Gamma, m \Vdash n}{\Gamma, \sigma_k\left(m\right), m \Vdash n}\ \mathsf{w}}{\Gamma, \sigma_k\left(m\right) \Vdash n}\ \mathsf{subst}$$

Translation from natural deduction to sequent calculus:

$$\dfrac{\Gamma \Vdash \langle m, n\rangle}{\Gamma \Vdash m}\ \langle\rangle E1 \quad \rightsquigarrow \quad \dfrac{\Gamma \Vdash \langle m, n\rangle \quad \dfrac{\overline{\Gamma, m \Vdash m}\ \mathsf{ax}}{\Gamma, \langle m, n\rangle \Vdash m}\ \langle\rangle L1}{\Gamma \Vdash m}\ \mathsf{cut}$$

$$\dfrac{\Gamma \Vdash \langle m, n\rangle}{\Gamma \Vdash n}\ \langle\rangle E2 \quad \rightsquigarrow \quad \dfrac{\Gamma \Vdash \langle m, n\rangle \quad \dfrac{\overline{\Gamma, n \Vdash n}\ \mathsf{ax}}{\Gamma, \langle m, n\rangle \Vdash n}\ \langle\rangle L2}{\Gamma \Vdash n}\ \mathsf{cut}$$

$$\dfrac{\Gamma \Vdash \sigma_k\,(m)}{\Gamma \Vdash m}\ \sigma E \quad \rightsquigarrow \quad \dfrac{\Gamma \Vdash \sigma_k\,(m) \quad \dfrac{\overline{\Gamma, m \Vdash m}\ \mathsf{ax}}{\Gamma, \sigma_k\,(m) \Vdash m}\ \sigma L}{\Gamma \Vdash m}\ \mathsf{cut}$$

□

1.2.3 Focused cut-free sequent calculus

$$\overline{\Gamma, m \Vdash [m]}\ \mathsf{ax}$$

$$\dfrac{\Gamma \Vdash [m]}{\Gamma \Vdash m}\ \mathsf{foc}$$

$$\dfrac{\Gamma \Vdash n}{\Gamma, m \Vdash n}\ \mathsf{w} \qquad \dfrac{\Gamma, m, m \Vdash n}{\Gamma, m \Vdash n}\ \mathsf{c}$$

$$\overline{\Gamma \Vdash [x]}\ xR$$

$$\dfrac{\Gamma \Vdash [m] \quad \Gamma \Vdash [n]}{\Gamma \Vdash [\langle m, n\rangle]}\ \langle\rangle R \qquad \dfrac{\Gamma, m \Vdash o}{\Gamma, \langle m, n\rangle \Vdash o}\ \langle\rangle L1 \qquad \dfrac{\Gamma, n \Vdash o}{\Gamma, \langle m, n\rangle \Vdash o}\ \langle\rangle L2$$

$$\dfrac{\Gamma \Vdash [k] \quad \Gamma \Vdash [m]}{\Gamma \Vdash [\sigma_k\,(m)]}\ \sigma R \qquad \dfrac{\Gamma, m \Vdash n}{\Gamma, \sigma_k\,(m) \Vdash n}\ \sigma L$$

Theorem 2. *The focused cut-free sequent calculus system is equivalent to the natural deduction system.*

Proof. LONG COMPLICATED PROOF TO BE FILLED IN HERE □

Theorem 3. *The focused cut-free sequent calculus has the subformula property.*

Proof. Immediate by inspection. □

1.2.4 Proof technique

With these tools in hand, the proof is now easy:

Theorem 4. *If neither k nor $\sigma_k\,(m)$ is a (left) subformula of Γ, then $\Gamma \Vdash \sigma_k\,(m)$ is not derivable.*

Proof. It suffices to prove that there does not exist a derivation of $\Gamma \Vdash \sigma_k\,(m)$ in the focused cut-free sequent calculus. Such a derivation has only two possible forms:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Delta \Vdash [k]}\,\mathsf{ax} \quad \dfrac{\vdots}{\Delta \Vdash [m]}}{\Delta \Vdash [\sigma_k\,(m)]}\,\sigma\mathsf{R}}{\Delta \Vdash \sigma_k\,(m)}\,\mathsf{foc}}{\dfrac{\vdots}{\Gamma \Vdash \sigma_k\,(m)}} \qquad \text{and} \qquad \dfrac{\dfrac{\dfrac{}{\Delta \Vdash [\sigma_k\,(m)]}\,\mathsf{ax}}{\Delta \Vdash \sigma_k\,(m)}\,\mathsf{foc}}{\dfrac{\vdots}{\Gamma \Vdash \sigma_k\,(m)}}$$

where, by the subformula property, Δ contains only (left) subformulas of Γ. The ax rule in the first case requires k to be in Δ, while the ax rule in the second case requires $\sigma_k\,(m)$ to be in Δ. But, by assumption, neither k nor $\sigma_k\,(m)$ is a subformula of Γ, so neither can be in Δ. Therefore no such derivation exists. ☐

1.3 Haskell embedding

We now give a strongly typed embedding in Haskell. The embedding enables an interpretation of derivations as programs that actually construct a message, given an implementation of the primitives (pairing and signing). Strong typing means that only well-formed derivations are accepted by the type checker, which confers a formal guarantee of correctness.

We need the following extensions and imports to approximate dependent types.

```haskell
{-# LANGUAGE DataKinds, GADTs, KindSignatures, TypeOperators #-}
import GHC.TypeLits
```

Messages are defined using (somewhat arbitrarily) integers as literals and keys. We actually use the automatic lifting of this definition to the type level, so we use the `Nat` type from `GHC.TypeLits` instead of the `Integer` type in order to be able to reflect back it to the value level.

```haskell
type Lit = Nat
type Key = Nat
data Message :: * where
  Lit  :: Lit -> Message
  Key  :: Key -> Message
  Pair :: Message -> Message -> Message
  Sign :: Key -> Message -> Message
```

We use standard encoding techniques for type-level programming in Haskell, representing inference rules as constructors for a generalized algebraic data type (GADT), and using strongly typed de Bruijn indices to access the context, which is represented as a type-level list.

Note that the `LitI` constructor takes a value-level proxy for a type-level integer, rather than a value-level integer. The latter can be recovered using `natVal`, e.g. `natVal (Proxy :: Proxy 42) == 42`.

```
type Context = [Message]

data (∈) :: Message -> Context -> * where
  Z :: -----------
       m ∈ (m : g)

  S :: m ∈ g
       -----------
    -> m ∈ (n : g)

data (⊩) :: Context -> Message -> * where
  Hyp    :: m ∈ g
            -----
         -> g ⊩ m

  Subst  :: g ⊩ m
         -> (m : g) ⊩ n
            -----------
         -> g ⊩ n

  LitI   :: KnownNat x
         => proxy x
            ----------
         -> g ⊩ 'Lit x

  PairI  :: g ⊩ m
         -> g ⊩ n
            -------------
         -> g ⊩ 'Pair m n

  PairE1 :: g ⊩ 'Pair m n
            -------------
         -> g ⊩ m

  PairE2 :: g ⊩ 'Pair m n
            -------------
         -> g ⊩ n
```

```
  SignI  :: g ⊩ 'Key k
         -> g ⊩ m
            -------------
         -> g ⊩ 'Sign k m

  SignE  :: g ⊩ 'Sign k m
            -------------
         -> g ⊩ m
```

An implementation of the primitive operations is an algebra for the signature functor of the derivation type, minus the `Hyp` and `Subst` constructors and forgetting the extra typing information.

```
data Signature :: * -> * where
  LitI'   :: Integer -> Signature a
  PairI'  :: (a, a)  -> Signature a
  PairE1' :: a       -> Signature a
  PairE2' :: a       -> Signature a
  SignI'  :: (a, a)  -> Signature a
  SignE'  :: a       -> Signature a

type Algebra f a = f a -> a
```

Finally, the interpreter is straightforward.

```
data Environment :: Context -> * -> * where
  Nil  :: Environment '[] a
  Cons :: a -> Environment g a -> Environment (m : g) a

eval :: Algebra Signature a -> g ⊩ m -> Environment g a -> a
eval _   (Hyp Z)      (Cons x _)   = x
eval alg (Hyp (S n))  (Cons _ env) = eval alg (Hyp n) env
eval alg (Subst d d') env          = eval alg d' (Cons (eval alg d env) env)
eval alg (LitI proxy) _   = (alg . LitI')   (natVal proxy)
eval alg (PairI d d') env = (alg . PairI')  (eval alg d env, eval alg d' env)
eval alg (PairE1 d)   env = (alg . PairE1') (eval alg d env)
eval alg (PairE2 d)   env = (alg . PairE2') (eval alg d env)
eval alg (SignI d d') env = (alg . SignI')  (eval alg d env, eval alg d' env)
eval alg (SignE d)    env = (alg . SignE')  (eval alg d env)
```