Sample Examination Questions for module MPRI 2-30
Cryptographic protocols: formal and computational proofs

(All documents are allowed; duration: 3h)

February 10, 2020

For the ease of the correctors, please use different sheets for the two parts of the exam.

2 Automated Protocol Analysis (1h30, 1/2 the mark)

We consider the following protocol, where Alice (A) wishes to send a secret, authenticated message m to Bob (B). We assume that both A and B know each other's public keys (pk_A,pk_B) but that only A and B know their own private keys (sk_A,sk_B).

The protocol consists of a single message from A to B as shown below: A first generates a fresh encryption key ek and a fresh MAC key mk, encrypts them both under the public key of B (pk_B), then signs the result of encryption using its own private key (sk_A), and finally encrypts then MACs the secret message m using ek and mk.

$$A \longrightarrow B : \mathsf{sign}(\mathsf{penc}((ek, mk), pk_B), sk_A), \mathsf{mac}(\mathsf{enc}(m, ek), mk)$$

Hence, the protocol uses four cryptographic primitives: public-key encryption (penc, pdec), public-key signatures (sign, verify, sign_val), symmetric encryption (enc, dec), and a message authentication code (mac, verifymac, mac_val). The types and symbolic rules (in ProVerif syntax) for these primitives are given in Appendix A.

The two security goals of the protocol can be informally stated as follows:

Confidentiality Any message m sent from A to B should only be known to A and B.

Authenticity If B accepts a message m from A, then m must have been sent by A to B.

We expect the above goals to hold as long as sk_A and sk_B are secret, even if:

- the adversary controls the network;
- A is willing to send other messages to other recipients, including the adversary;
- B is willing to accept messages from multiple senders, including the adversary.

2.1 Exercise 1

Does the protocol satisfy its two security goals?
If yes, informally explain why.
If not, demonstrate an attack, and then fix the protocol so that it achieves its goals.

2.2 Exercise 2

Write an F* implementation for the protocol above.
The model will consist of:

- a function for A that takes m, pk_B, and sk_A as input and returns the (encrypted) message c that will be sent on the network;
- a function for B that takes c, sk_A, and pk_B as input and returns the message m.

You can assume the F* declarations in Appendix A for the cryptographic primitives. If you need additional functions, declare them.

2.3 Exercise 3

Appendix B presents a labeled cryptographic API. Write labeled types for the two functions (for A and B) by assigning labels to all the inputs and outputs. State why your types guarantee the secrecy of m.

2.4 Exercise 4

Rewrite the protocol code to use the labeled API. By appealing to the labeled crypto API in Appendix B, and by reasoning line-by-line in your code, argue why your protocol code is well-typed.

2.5 Exercise 5

Informally describe how would you encode the authenticity goal, and what are the steps needed to prove it for your protocol code.

A Unlabeled Cryptographic API

```
(* A type for byte arrays *)
val bytes: Type0
val concat: bytes → bytes → bytes
val split: bytes → option (bytes * bytes)
val concat_split_lemma: b1:bytes → b2:bytes →
  Lemma (split (concat b1 b2) == Some (b1,b2))
  [SMTPat (split (concat b1 b2))]



(* Symmetric Encryption *)
val sym_key: Type0
val sym_keygen: unit → ST sym_key
                         (requires (λ h0 → ⊤))
                         (ensures (λ h0 _h1 → h0 == h1))
val sym_enc: k:sym_key → p:bytes → c:bytes
val sym_dec: k:sym_key → c:bytes → option bytes
val sym_enc_dec_lemma: k:sym_key → p:bytes →
  Lemma (sym_dec k (sym_enc k p) == Some p)
  [SMTPat (sym_dec k (sym_enc k p))]

(* Message Authentication Codes (MAC) *)
val mac_key: Type0
val mac_keygen: unit → ST mac_key
                         (requires (λ h0 → ⊤))
                         (ensures (λ h0 _h1 → h0 == h1))
```

```
val mac: k:mac_key → p:bytes → tag:bytes

(* Public Key Encryption – Asymmetric *)
val pub_key: Type0
val priv_key: Type0
val priv_keygen: unit → ST priv_key
                        (requires (λ h0 → ⊤))
                        (ensures (λ h0 _h1 → h0 == h1))
val priv_to_pub: priv_key → pub_key
val pke_enc: pk:pub_key → k:ae_key → c:bytes
val pke_dec: sk:priv_key → c:bytes → option ae_key
val pke_enc_dec_lemma: sk:priv_key → k:ae_key →
  Lemma (pke_dec sk (pke_enc (priv_to_pub sk) k) == Some k)
  [SMTPat (pke_dec sk (pke_enc (priv_to_pub sk) k))]


(* Signatures – Asymmetric *)
val sig_key: Type0
val verif_key: Type0
val sig_keygen: unit → ST sig_key
                       (requires (λ h0 → ⊤))
                       (ensures (λ h0 _h1 → h0 == h1))
val sig_to_verif: sig_key → verif_key
val sign: sk:sig_key → msg:bytes → sg:bytes
val verify: vk:verif_key → msg:bytes → sg:bytes → bool
val sign_verify_lemma: sk:sig_key → msg:bytes →
  Lemma (verify (sig_to_verif sk) msg (sign sk msg) == true)
  [SMTPat (verify (sig_to_verif sk) msg (sign sk msg))]
```

B Labeled Cryptographic API

```
(* Principals: participants in protocols *)
val principal: eqtype

(* Raw bytes *)
val bytes: Type0

(* Secrecy Labels: sets of principals *)
type label =
  | Public: label
  | Secret: list principal → label
let includes l1 l2 =
  match l1,l2 with
  | Public,_ → ⊤
  | Secret pl1, Secret pl2 → ∀p. List.Tot.mem p pl2 ⟹ List.Tot.mem p pl1
  | _,_ → ⊥

(* A type for *labeled* byte arrays *)
val lbytes: label → Type0
val coerce: l1:label → l2:label{includes l1 l2} → b1:lbytes l1 → b2:lbytes l2

(* Concatenation preserves labels *)
val concat: l:label → lbytes l → lbytes l → lbytes l
val split: l:label → lbytes l → option (lbytes l * lbytes l)
val concat_split_lemma: l:label → b1:lbytes l → b2:lbytes l →
  Lemma (split l (concat l b1 b2) == Some (b1,b2))
  [SMTPat (split l (concat l b1 b2))]

(* Symmetric Encryption *)
val sym_key: l:label → Type0
val coerce_sym_key: l1:label → l2:label{includes l1 l2} → sym_key l1 → sym_key l2
val sym_keygen: l:label → ST (sym_key l)
                       (requires (λ h0 → ⊤))
```

```
                        (ensures (λ h0 _h1 → h0 == h1))

val sym_enc: kl:label → ml:label{includes ml kl} → k:sym_key kl → m:lbytes ml → c:lbytes Public
val sym_dec: kl:label → k:sym_key kl → c:lbytes Public → option (lbytes kl)

val sym_enc_dec_lemma: kl:label → ml:label → k:sym_key kl → m:lbytes ml →
  Lemma (requires (includes ml kl))
        (ensures (sym_dec kl k (sym_enc kl ml k m) == Some (coerce ml kl m)))
  [SMTPat (sym_dec kl k (sym_enc kl ml k m))]

(* Message Authentication Codes (MAC) *)
val mac_key: label → Type0
val coerce_mac_key: l1:label → l2:label{includes l1 l2} → mac_key l1 → mac_key l2
val mac_keygen: l:label → ST mac_key l
                        (requires (λ h0 → ⊤))
                        (ensures (λ h0 _h1 → h0 == h1))
val mac: l:label → k:mac_key l → p:lbytes l → tag:lbytes l

(* Public Key Encryption – Asymmetric *)
val pub_key: a:principal → Type0
val priv_key: a:principal → Type0
val priv_keygen: a:principal → ST (priv_key a)
                        (requires (λ h0 → ⊤))
                        (ensures (λ h0 _h1 → h0 == h1))
val priv_to_pub: a:principal → priv_key a → pub_key a
val pke_enc: r:principal → kl:label{includes kl (Secret [r])} →
             pk:pub_key r → k:ae_key kl → c:lbytes Public

val pke_dec: r:principal → sk:priv_key r → c:lbytes Public → option (ae_key (Secret [r]))
val pke_enc_dec_lemma: kl:label → r:principal → sk:priv_key r → k:ae_key kl →
  Lemma (requires (includes kl (Secret [r])))
        (ensures (pke_dec r sk (pke_enc r kl (priv_to_pub r sk) k) == Some (coerce_ae_key kl (Secret [r]) k)))
  [SMTPat (pke_dec r sk (pke_enc r kl (priv_to_pub r sk) k))]

(* Signatures – Asymmetric *)

val sig_key: a:principal → Type0
val verif_key: a:principal → Type0
val sig_keygen: a:principal → ST (sig_key a)
                        (requires (λ h0 → ⊤))
                        (ensures (λ h0 _h1 → h0 == h1))
val sig_to_verif: a:principal → sig_key a → verif_key a
val sign: a:principal → ml:label → sk:sig_key a → msg:lbytes ml → sg:lbytes ml
val verify: a:principal → ml:label → vk:verif_key a → msg:lbytes ml → sg:lbytes ml → bool
val sign_verify_lemma: a:principal → ml:label → sk:sig_key a → msg:lbytes ml →
  Lemma (verify a ml (sig_to_verif a sk) msg (sign a ml sk msg) == true)
  [SMTPat (verify a ml (sig_to_verif a sk) msg (sign a ml sk msg))]
```