1 Security verification of the SAKE$^+$ with ProVerif

We use the ProVerif tool [2] to formally prove the SAKE$^+$ protocol proposed in main.pdf. The ProVerif enables us to verify the **concurrent** execution of our protocol and to make sure whether our protocol achieves the desired security objectives or not. Parties involved in the protocol use a channel to communicate with each other. This channel is assumed to be controlled by an adversary that is able to read, change, delete, and create messages, and the model in which the attacker operates is called "Dolev-Yao" [3]. The attacker is capable of the modification of the protocol's messages in that s/he can decrypt messages (if s/he gets access to the related keys) and can even compute the i_th element of a tuple. We can encode our desired protocol and its objectives using the ProVerif's input language formally, enabling the ProVerif tool to verify our claimed security properties. The cryptographic primitives used in ProVerif is assumed to be perfect, i.e., the adversary is not able to make use of any polynomial-time algorithm and s/he can only makes use of the cryptographic primitives defined by the user. With the help of rewrite rules and/or an equational theory, the cryptographic primitives are associated with each other.

A protocol that is written in the ProVerif tool's input language (the typed pi calculus [5]) includes the following components: the *declarations*, the *processmacros* and the *mainprocess*. These components are discussed as follows:

- Declarations: The declarations consists of the user types, the functions describing the cryptographic primitives, and the security properties as well.
- Process macros: The process macros include sub-process definitions; each sub-process is a sequence of events.
- Main process: It is defined with the help of macros. In our particular SAKE$^+$ protocol, the main process is defined as the parallel composition (denoted by |) of the unbounded replication (denoted by !) of two process macros representing the *processInitiator*, and *processResponder* nodes.

ProVerif can prove both reachability property and correspondence assertions [1]. The Reachability property allows us to check which information an attacker can access, i.e. secrecy. Correspondence property is of the form "if some event is executed, then another event has been executed previously", and could be used for checking various types of authentication [4]. We encoded the SAKE$^+$ AKE protocol in the ProVerif language. In general, a protocol model can be divided into three different parts: the declarations (lines 1-49), the process macros (lines 49-255), and the main process (lines 256-264).

```
1 (*-SAKE+ channel-*)
2 free c: channel.
3 (*-SAKE+ types-*)
4 type key.
5 type nonce.
6 type host.
7 (*-SAKE+ keys-*)
8 free SKa,SKb,K,Kj0,Kj1,Kj2:bitstring [private].
9 free Bj0,Bj1,Bj2,Rb0,B:bitstring.
10 free A: host.
```

```
11 (*-SAKE+ constants-*)
12 const f0,f1,Dhel0,Dhel1,DhelN,Epsi0,Epsi1,X: bitstring.
13 table TA(host,key,bitstring,bitstring,bitstring,bitstring,nonce).
14 table TB(bitstring,key,bitstring,bitstring).
15 (*-SAKE+ functions-*)
16 fun nonce_to_bitstring(nonce): bitstring [data,typeConverter].
17 fun bitstring_to_key(bitstring): key [data,typeConverter].
18 fun host_to_bitstring(host): bitstring [data,typeConverter].
19 fun bitstring_to_nonce(bitstring): nonce [data,typeConverter].
20 fun mac(bitstring, key): bitstring.
21 reduc forall m: bitstring, k: key; get_message(mac(m,k)) = m.
22 fun PRF(bitstring,key): bitstring.
23 fun con(bitstring,bitstring): bitstring.
24 (*-SAKE+ events-*)
25 event beginAparam(host, nonce).
26 event endAAuth(host, nonce).
27 event beginBparam(bitstring, nonce).
28 event endBAuth(bitstring, nonce).
29 event beginsyncBkey(bitstring, host, nonce, key).
30 event endsyncBkey(bitstring, host, nonce, key).
31 (*-SAKE+ queries-*)
32 query attacker(SKa).
33 query attacker(SKb).
34 query attacker(K).
35 query attacker(Kj0).
36 query attacker(Kj1).
37 query attacker(Kj2).
38 query x: host, y: nonce; inj-event(endAAuth(x, y)) ==> inj-event(beginAparam(
       x, y)).
39 query x: bitstring, y: nonce; inj-event(endBAuth(x, y)) ==> inj-event(
       beginBparam(x, y)).
40 query x: bitstring, y: host, z: nonce, t: key; inj-event(endsyncBkey(x,y,z,t)
       ) ==> inj-event(beginsyncBkey(x,y,z,t)).
```

1.1 Declarations

The declarations include the user types, the functions that describe the cryptographic primitives, and the security properties. Additional user types can be declared as in lines 4-6 apart from the built-in types: **channel** and **bitstring**. Free names are defined as in lines 2 and 8-10 where the channel with names **c** is declared. By default, the free names are accessible to the attacker unless qualified by [**Private**]. Finally, constant values are declared by **const**. The language supports tables for persistent storage. In lines 13 and 14, tables that model the subscribers database is declared.

Constructors are functions used to build terms. A constructor is declared by defining its names, the types of its arguments and the return value (see lines 16-20, 22-23). Functions, by default, are one-way; i.e., the attacker cannot infer the arguments from the return value, unless qualified by [**data**]. Destructors (line 21) are special functions that are used to manipulate terms. Constructors and destructors jointly are used to capture the relationship between cryptographic primitives.

Message authentication codes (MAC) can be declared by a constructor (with no associated destructor or equation), much like a keyed hash function as follow:

type key.

fun mac (bitstring,key):bitstring.

This model is strong in the sense that it considers the MAC as a random oracle. If the MAC is considered to be a pseudo-random function (PRF), it is probably the best possible model (in line 22, it is presented as **fun PRF (bitstring,key):bitstring.**).

Considering that the MAC is unforgeable (UF-CMA), one can declare a destructor which leaks the MACed message as follow:

reduc for all m:bitstring, k: key; getmessage (mac(m,k)) = m.

A sequences of events presented in lines 25-30, are defined as follows:

- The *beginAparam* event declares that the initiator A starts the authentication protocol with its identity A and a fresh nonce.
- The *endAAuth* event declares that the initiator A will authenticated with the responder that received the fresh nonce generated by A.
- The events *beginBparam* and *endBAuth* for the entity B are the same as *beginAparam* and *endAAuth* events for the entity A, respectively.
- The *eginsyncBkey* and *endsyncBkey* events declare that the responder B is in the synchronize state with the initiator A.

We model correspondence assertions of the form: "if an event *end* has been executed, then event *begin* has been previously executed." with the queries presented in lines 38-40 that the first two queries (lines 38 and 39) are for the mutual authentication and the last one is for the synchronization. The rest of the queries which are presented in lines 32-37 is base on a built-in predicate attacker used to check which terms are compromised.

1.2 Process macros

The process macros consist of sub-process definitions that are a sequence of events. Messages are represented by terms, i.e., a name, a variable, a tuple of terms, a constructor or destructor application. The language, additionally, supports some common Boolean functions ($=, \&\&, ||, <>$) with the infix notation.

There are term evaluation, restriction, communication and condition events defined as follows:

- The pattern **x : t** matches any term of type **t** and binds it to **x**.
- the **let x = M in** binds the term **M** to **x**.
- The name restriction event **new** declares a fresh name of a specific type and binds it inside the events. For instance, line 43 binds the type **nonce** to the fresh name R_a.
- The communication event **in (c,(x:host,y:nonce))** listen from a channel **c** and binds the received terms to **x** and **y** where the first one has type **host** and the second one has type **nonce**.
- The communication event textbfout (c,(x:host,y:nonce)), sends the terms **x** and **y** on channel **c**.

– The conditional **if M else P then Q** continues as the process **P** if the term **M** evaluates to true, continues as the process **Q** if **M** evaluates to another value.

```
(* Role of the initiator *)
let processInitiator =
new Ra: nonce;
new aDhel: bitstring;
new aEpsi: bitstring;
new akj: bitstring;
get TA(aA,aK,aKj0,aKj1,aBj0,aBj1,aRb0) in
let A = aA in
let m0 = (A,Ra) in
event beginAparam(A, Ra);
(* ---->A||r_A *)
out(c,m0);
(* m_B<---- *)
in(c,(aB:bitstring,aRb:nonce,aTb:bitstring));
let aBj2 = PRF(X, bitstring_to_key(aBj1)) in
if aB = aBj0 then let B = aBj0 in
if aB = aBj1 then let B = aBj1 in
if aB = aBj2 then let B = aBj2 in
if aB <> aBj0 && aB <> aBj1 && aB <> aBj2 then let B = aBj1 in
if aRb0 <> aRb then
(
  let Tbin = con(B,con(host_to_bitstring(A),con(nonce_to_bitstring(aRb),
    nonce_to_bitstring(Ra)))) in
  let Tb = mac(Tbin, bitstring_to_key(aKj1)) in
  if Tb = aTb then
  (
  let aDhel = Dhel0 in
  let akj = aKj1 in
  let SKa = PRF(con(nonce_to_bitstring(Ra),nonce_to_bitstring(aRb)), aK) in
  let aK = PRF(X, aK) in
  let aBj1 = PRF(X, bitstring_to_key(con(aBj1,aKj1))) in
  let aKj0 = aKj1 in
  let aKj1 = PRF(X, bitstring_to_key(aKj1)) in
  let aBj0 = aBj1 in
  let aEpsi = Epsi0 in
  let aRb0 = aRb in
  insert TA(A,bitstring_to_key(aK),aKj0,aKj1,aBj0,aBj1,aRb0);
        let Tain = con(aEpsi,con(host_to_bitstring(A),con(B,(
    nonce_to_bitstring(Ra),nonce_to_bitstring(aRb))))) in
  let Ta = mac(Tain, bitstring_to_key(akj)) in
  let ma = (B,aEpsi,Ta) in
  out(c,ma)
    (* ---->m_A *)
    )
    else
  let Tbin = con(B,con(host_to_bitstring(A),con(nonce_to_bitstring(aRb),
    nonce_to_bitstring(Ra)))) in
  let Tb = mac(Tbin, bitstring_to_key(aKj0)) in
  if Tb = aTb then
  (
```

```
    let aDhel = Dhel1 in
    let akj = aKj0 in
    let aEpsi = Epsi1 in
    let aRb0 = aRb in
    insert TA(A,aK,aKj0,aKj1,aBj0,aBj1,aRb0);
            let Tain = con(aEpsi,con(host_to_bitstring(A),con(B,(
   nonce_to_bitstring(Ra),nonce_to_bitstring(aRb))))) in
    let Ta = mac(Tain, bitstring_to_key(akj)) in
    let ma = (B,aEpsi,Ta) in
    out(c,ma)
      (* ---->m_A *)
  )
    else
  let Tbin = con(B,con(host_to_bitstring(A),con(nonce_to_bitstring(aRb),
   nonce_to_bitstring(Ra)))) in
  let Tb = mac(Tbin, bitstring_to_key(PRF(X, bitstring_to_key(aKj1)))) in
  if Tb = aTb then
  (
    let aDhel = DhelN in
    let akj = PRF(X, bitstring_to_key(aKj1)) in
    let aK = PRF(X, aK) in
    let aBj1 = PRF(X, bitstring_to_key(con(aBj1,aKj1))) in
    let aKj0 = aKj1 in
    let aKj1 = PRF(X, bitstring_to_key(aKj1)) in
    let aBj0 = aBj1 in
    let SKa = PRF(con(nonce_to_bitstring(Ra),nonce_to_bitstring(aRb)),
   bitstring_to_key(aK)) in
    let aK = PRF(X, bitstring_to_key(aK)) in
    let aBj1 = PRF(X, bitstring_to_key(con(aBj1,aKj1))) in
    let aKj0 = aKj1 in
    let aKj1 = PRF(X, bitstring_to_key(aKj1)) in
    let aBj0 = aBj1 in
    let aEpsi = Epsi0 in
    let aRb0 = aRb in
    insert TA(A,bitstring_to_key(aK),aKj0,aKj1,aBj0,aBj1,aRb0);
            let Tain = con(aEpsi,con(host_to_bitstring(A),con(B,(
   nonce_to_bitstring(Ra),nonce_to_bitstring(aRb))))) in
    let Ta = mac(Tain, bitstring_to_key(akj)) in
    let ma = (B,aEpsi,Ta) in
    out(c,ma)
      (* ---->m_A *)
  )
  else
  yield
)
else
(* B||T^P_B<---- *)
in(c,(aB:bitstring,aTpb:bitstring));
let B = aB in
if aEpsi = Epsi0 then
(
  let akj = aKj1 in
  let Tpbin = con(nonce_to_bitstring(aRb),nonce_to_bitstring(Ra)) in
```

```
  let Tpb = mac(Tpbin, bitstring_to_key(akj)) in
  if Tpb = aTpb then
  (
    let Tpain = con(nonce_to_bitstring(Ra),nonce_to_bitstring(aRb)) in
    let Tpa = mac(Tpain, bitstring_to_key(akj)) in
    let m3 = (B,Tpa) in
    event beginsyncBkey(B, A, aRb, bitstring_to_key(akj));
    out(c,m3)
      (* ---->B||T^p_A *)
  )
  else
  yield
)
else
if aEpsi = Epsi1 then
(
  let akj = PRF(X, bitstring_to_key(aKj1)) in
  let Tpbin = con(nonce_to_bitstring(aRb),nonce_to_bitstring(Ra)) in
  let Tpb = mac(Tpbin, bitstring_to_key(akj)) in
  if Tpb = aTpb then
  (
    let SKa = PRF(con(nonce_to_bitstring(Ra),nonce_to_bitstring(aRb)), aK) in
    let aK = PRF(X, aK) in
    let aBj1 = PRF(X, bitstring_to_key(con(aBj1,aKj1))) in
    let aKj0 = aKj1 in
    let aKj1 = PRF(X, bitstring_to_key(aKj1)) in
    let aBj0 = aBj1 in
    let Tpain = con(nonce_to_bitstring(Ra),nonce_to_bitstring(aRb)) in
    let Tpa = mac(Tpain, bitstring_to_key(akj)) in
    let m3 = (B,Tpa) in
    event beginsyncBkey(B, A, aRb, bitstring_to_key(akj));
    event endBAuth(B, aRb);
    out(c,m3)
      (* ---->B||T^p_A *)
  )
  else
  yield
)
else
0.
(* Role of the responder *)
let processResponder =
(* A||r_A<---- *)
in(c,(bA:host, bRa:nonce));
get TB(bB,bK,bKj,bf) in
new Rb: nonce;
let bTbin = con(bB,con(host_to_bitstring(bA),con(nonce_to_bitstring(Rb),
    nonce_to_bitstring(bRa)))) in
let bTb = mac(bTbin, bitstring_to_key(bKj)) in
new Ralfa: nonce;
let mb = (if bf= f1 then (nonce_to_bitstring(Ralfa),Rb,bTb) else (bB,Rb,bTb))
     in
let bf= f1 in
```

```
event beginBparam(bB, Rb);
(* ---->m_B *)
out(c,(mb));
(* m_A<---- *)
in(c,(bBp: bitstring,bEpsi: bitstring,bTa: bitstring));
let Tapin = con(bEpsi,con(host_to_bitstring(bA),con(bB,con(nonce_to_bitstring
    (bRa),nonce_to_bitstring(Rb))))) in
let Tap = mac(Tapin, bitstring_to_key(bKj)) in
if Tap = bTa then
if bEpsi = Epsi1 then
(
  let bK = PRF(X, bK) in
  let bB = PRF(X, bitstring_to_key(con(bB,bKj))) in
  let bKj = PRF(X, bitstring_to_key(bKj)) in
  let SKb = PRF(con(nonce_to_bitstring(bRa),nonce_to_bitstring(Rb)),
    bitstring_to_key(bK)) in
  let bK = PRF(X, bitstring_to_key(bK)) in
  let bB = PRF(X, bitstring_to_key(con(bB,bKj))) in
  let bKj = PRF(X, bitstring_to_key(bKj)) in
  let bf= f0 in
  insert TB(bB,bitstring_to_key(bK),bKj,bf);
  let Tpbpin = con(nonce_to_bitstring(Rb),nonce_to_bitstring(bRa)) in
  let Tpbp = mac(Tpbpin, bitstring_to_key(bKj)) in
  let m2 = (bB,Tpbp) in
  (* ---->B||T^p_B *)
  out(c,m2)
)
else
let SKb = PRF(con(nonce_to_bitstring(bRa),nonce_to_bitstring(Rb)), bK) in
let bK = PRF(X, bK) in
let bB = PRF(X, bitstring_to_key(con(bB,bKj))) in
let bKj = PRF(X, bitstring_to_key(bKj)) in
let bf= f0 in
insert TB(bB,bitstring_to_key(bK),bKj,bf);
let Tpbpin = con(nonce_to_bitstring(Rb),nonce_to_bitstring(bRa)) in
let Tpbp = mac(Tpbpin, bitstring_to_key(bKj)) in
let m2 = (bB,Tpbp) in
(* ---->B||T^p_B *)
out(c,m2);
(* B||T^p_A<---- *)
in(c,(bBpp:bitstring,bTpa:bitstring));
let bTpapin = con(nonce_to_bitstring(bRa),nonce_to_bitstring(Rb)) in
let bTpap = mac(bTpapin, bitstring_to_key(bKj)) in
if bTpap = bTpa then
(
event endsyncBkey(bBpp, bA, Rb, bitstring_to_key(bKj));
event endAAuth(bA, bRa)
)
else
0.
```

1.3 Main process

Finally, the main process is defined by means of two process macros that represent the *processInitiator* (line 242) and *processInitiator* (line 243) nodes. The initialization phase of the scheme is presented in lines 237-240 for an initiator A and a responder B in line 237, and lines 238-240, respectively. Finally, in lines 240-243, the parallel compositions of *processInitiator* and *processResponder* denoted by | with the unbounded replication (denoted by !).

```
236 process
237   insert TA(A,bitstring_to_key(K),Kj0,Kj1,Bj0,Bj1,bitstring_to_nonce(Rb0));
238   new f: bitstring;
239   let f = f0 in
240   insert TB(B,bitstring_to_key(K),Kj0,f);
241   (
242   (!processInitiator) |
243   (!processResponder)
244   )
```

1.4 Security properties

Security properties are declared with the keyword. In our example of SAKE$^+$, the goal is to establish the shared session key $SK_a = SK_b$ between A and B after mutual authentication by preserving the forward secrecy. The protocol should be robust against the traceability and de-synchronization attacks. In order to check this, we consider the following queries.

```
31 (*-SAKE+ queries-*)
32 query attacker(SKa).
33 query attacker(SKb).
34 query attacker(K).
35 query attacker(Kj0).
36 query attacker(Kj1).
37 query attacker(Kj2).
38 query x: host, y: nonce; inj-event(endAAuth(x, y)) ==> inj-event(beginAparam(
       x, y)).
39 query x: bitstring, y: nonce; inj-event(endBAuth(x, y)) ==> inj-event(
       beginBparam(x, y)).
40 query x: bitstring, y: host, z: nonce, t: key; inj-event(endsyncBkey(x,y,z,t)
       ) ==> inj-event(beginsyncBkey(x,y,z,t)).
```

- The first six queries presented in lines 32-37 is base on a built-in predicate attacker used to check which terms are compromised.
- The query presented in line 38 proves that B successfully authenticates A, if ProVerif returns **true**. The event *beginAparam* is called in line 50 on the new nonce R_a generated by the initiator A and the event *endAAuth* is called by the responder B in line 23 after successful authentication of the initiator and establishing the session key.
- The query presented in line 39 proves that A successfully authenticates B, if ProVerif returns **true**. The events of this query are presented as *beginBparam* and *endBAuth* in lines 188 and 168, respectively.

– The query presented in line 40 proves that the responder B will be successful in the synchronization state using the events *eginsyncBkey* and *endsyncBkey* on the master key K' (see lines 143,167 and 231) in case that **true** is resulted from ProVerif.

The results are illustrated bellow and show that all the events result in **true**, which prove that the $SAKE^+$ can preserve all the mentioned security queries.

```
Verification summary:
Query not attacker(SKa[]) is true.
Query not attacker(SKb[]) is true.
Query not attacker(K[]) is true.
Query not attacker(Kj0[]) is true.
Query not attacker(Kj1[]) is true.
Query not attacker(Kj2[]) is true.
Query inj-event(endAAuth(x,y)) ==> inj-event(beginAparam(x,y)) is true.
Query inj-event(endBAuth(x,y)) ==> inj-event(beginBparam(x,y)) is true.
Query inj-event(endsyncBkey(x,y,z,t)) ==> inj-event(beginsyncBkey(x,y,z,t))
    is true.
```

1.5 Analysis and discussion

As mentioned earlier, all the queries are solved as expected, that is, the correspondence and secrecy ones are proved. We encode our security goals using the ProVerif queries as follows:

Secrecy for a message, such as m2, encoded using MAC function that asks the adversary to guess the value of K'_j; if the adversary succeeds, the ProVerif issues **false** to the query **query attacker(Kj0).**. We have the same discussion for the other terms **free X:bitstring [private].** in our scheme. For all of these queries, we received the result **true** from ProVerif, which means that the protocol satisfies secrecy.

Forward secrecy for a master key K'_{j-1}, we use the **free Kj1:bitstring.** instead of **free Kj1:bitstring [private].**, which gives the attacker knowledge of the current master key K'_j and asks the adversary to guess the value of K'_{j-1} using the query **query attacker(Kj0).**; if the adversary succeeds, the ProVerif issues **false** to the query. We received the result **true** from ProVerif meaning that the protocol satisfies forward secrecy.

Authentication Considering the query presented in line 38, if ProVerif returns **true**, it proves that B successfully authenticates A and considering the query presented in line 39, if ProVerif returns **true**, it proves that A successfully authenticates B. By satisfying these two queries, we can ensure that our scheme provides mutual authentication successfully.

Replay The events *beginAparam*, *endAAuth*, *beginBparam* and *endBAuth* used for the authentication are based on the fresh nonces R_a and R_b. With regard to the result **true** from both queries presented in lines 38 and 39, it proves that the scheme is secure against the replay attack.

synchronization The query presented in line 40 proves that the responder B will be successful in the synchronization state using the events *eginsyncBkey* and *endsyncBkey* on the master key K' (see lines 143,167 and 231) in case that ProVerif shows the **true** results.

In addition to the above queries, our scripts also include built-in predicate attacker used to check which terms are compromised (presented in lines 32-37).

References

1. B. Blanchet. Automatic verification of correspondences for security protocols. Journal of Computer Security, 17(4):363–434, 2009.
2. B. Blanchet, B. Smyth, V. Cheval, and M. Sylvestre. Proverif 2.00: automatic cryptographic protocol verifier, user manual and tutorial, 2018.
3. D. Dolev and A. Yao. On the security of public key protocols. IEEE Transactions on Information Theory, 29(2):198–208, 1983.
4. G. Lowe. A hierarchy of authentication specifications. In CSF, pages 31–43. IEEE, 1997.
5. M. D. Ryan and B. Smyth. Applied pi calculus. 2011.