

ProVerif Code for Verification of the NTS Specification

Kristof Teichel

Physikalisch-Technische Bundesanstalt,
Bundesallee 100, 38116 Braunschweig, Germany

This document provides some description and clarification for the ProVerif source code used in the first verification of the NTS specification. There have been different code versions involved in the analysis:

- **Code version c030ut:** It models protocol version 0.3.0 but has no dedicated type for hostnames.
- **Code version c030:** It models protocol version 0.3.0 and does have a dedicated hostname type.
- **Code version c031:** It models protocol version 0.3.1.
- **Code version c032:** It models protocol version 0.3.2.

All of the code below is taken from code version c030, which formed the basis of the analysis. Presenting the other versions in full would take up a lot of space, whereas presenting only the differences is difficult. This is because although the changes between the different code versions are minor, they still include numerous lines that are spread far apart. Comment lines and some structuring have been left out for the presentation. Hopefully, these explanations are helpful in understanding the ProVerif code files in this repository.

Cryptographic Primitives

The first lines of the ProVerif source code form the cryptographic primitives that are needed.

```
(* Basics: Keys and Hostnames *)
type key.
type hostname.

(* Symmetric Encryption *)
fun senc(bitstring, key): bitstring.
reduc forall m: bitstring, k: key;
  sdec(senc(m,k),k) = m.

(* Asymmetric Encryption *)
type skey.
type pkey.
fun sk_of(hostname): skey [private].
fun pk(skey): pkey.
letfun pk_of(X: hostname) = pk(sk_of(X)).
```

```

fun aenc(bitstring, pkey): bitstring.
reduc forall m: bitstring, k: skey;
  adec(aenc(m, pk(k)), k) = m.

(* Asymmetric Signatures *)
type skey.
type spkey.
fun ssk_of(hostname): skey [private].
fun spk(sskey): spkey.
letfun spk_of(X: hostname) = spk(ssk_of(X)).

fun sign(bitstring, sskey): bitstring.
reduc forall m: bitstring, k: sskey;
  getmess(sign(m,k)) = m.
reduc forall m: bitstring, k: sskey;
  checksign(sign(m,k), spk(k)) = m.
letfun signed_message(m: bitstring, k: sskey)
  = (m, sign(m, k)).

(* Hash and HMAC Functions *)
fun hash(bitstring): bitstring.
fun keyhash(pkey): bitstring.

fun seed_of(hostname): bitstring [private].

fun cookie_gen(bitstring, bitstring): key.
fun hmac(key, bitstring): bitstring.

```

Global Variables and Constants

Next we have the declarations of channels and other global variables and constants.

```

(* The Channel for All "Network" Protocol Communications *)
free c: channel.

(* The Channel and Hostname for Communication with the TA *)
free ta: channel.
free TA: hostname.

(* Two Possible Version Identifiers *)
free version_1: bitstring.
free version_2: bitstring.

```

Events

The declarations of ProVerif “events” follow that are used for better traceability of what is happening in what order.

```

(* Server Side Events *)
event serverError().

event serverHasOwnCert(hostname, pkey, spkey).
event serverRespondsTime(bitstring, bitstring,
                        bitstring).

```

```

event serverSaysCookie(key, pkey, hostname).
event serverGeneratesCookie(key, hostname).

event serverAcceptsCert(hostname).
event serverRejectsCert().

```

```

(* Client Side Events *)
event clientError().
event clientHasOwnCert(hostname, pkey, spkey).

event clientAcceptsCookie(key, hostname, hostname).
event clientAcceptsSomeCookie().

event clientRejectsCookie(key, hostname, hostname).
event clientRejectsSomeCookie().

event cookieCompromised(key).

event clientAcceptsCert(hostname, spkey).
event clientRejectsCert().

event client_timereq(bitstring).
event clientAcceptsTime(bitstring, bitstring,
                        bitstring).
event clientDiscards(bitstring).

```

```

(* Authority Side Event(s) *)
event authorityGivesCert(hostname, pkey, spkey).

```

The Trusted Authority Process

The code then moves on to the processes by which the participants are modeled. We first present the process that models the trusted authority. Its only purpose is to generate and distribute certificates for server and client type participants.

```

(*-----+
|--- Trusted Authority Side Process ---|
+-----*)

(* [Process] :: TRUSTED AUTHORITY :: [NTS v0.3.0]
   |: Issues certificates on request. *)
let authority() =
  let skTA = ssk_of(TA) in

  (* The authority receives a certificate request. *)
  in(ta, H: hostname);

  (* The authority determines the correct public keys
     for the requester. *)
  let pkH = pk(ssk_of(H)) in
  let spkH = spk(ssk_of(H)) in

  (* From that information, the authority assembles
     and signs the appropriate certificate. *)
  let certificate = (H, pkH, spkH) in
  let signature = sign(certificate, skTA) in
  let certificate_sig = (certificate, signature) in

  (* The authority sends the response back to the

```

```

requesting participant. *)
event authorityGivesCert(H, pkH, spkH);
out(ta, certificate_sig).

```

The Server Side Processes

Inner Server Processes

Next are those processes that make up the model of the server. The first process represents the server module which deals with the certification message exchange.

```

(*-----+
|--- Server Side Processes ---|
+-----*)

(* [Process] :: SERVER CERTIFIER MODULE :: [NTS v0.3.0]
|: Replies to a client_cert message with a server_cert
|: message as specified. *)
let server_certifier(B: hostname, pkB: spkey,
                    skB: sskey) =

  (* The server acquires the TA's public key. *)
  let pkTA = spk(ssk_of(TA)) in

  (* The server receives a client's certification
  request. *)
  in(c, X_client_cert: bitstring);

  (* The server extracts the necessary information. *)
  let (version_x: bitstring, A_x: hostname)
    = X_client_cert in

  (* The server requests its certificate chain from the
  trusted authority and performs a validity check on
  the response that it receives. *)
  out(ta, B);
  in(ta, Z_certificate: bitstring);
  let (=B, some_key: pkey, =pkB,
      Z_cert_signature: bitstring)
    = Z_certificate in
  if (B, some_key, pkB)
  <> checksign(Z_cert_signature, pkTA)
  then event serverError()
  else event serverHasOwnCert(B, some_key, pkB);

  (* The server creates a server_cert response
  as specified. *)
  let msg_server_cert = (A_x, Z_certificate) in
  let msg_server_cert_sign
    = (msg_server_cert,
      sign(msg_server_cert, skB))
    in

  (* The server sends the composed response to the
  requesting client. *)
  out(c, msg_server_cert_sign).

```

The next process involves the server module whose purpose it is to execute the cookie message exchange as well as the required calculations.

Then the server module follows that takes care of the time synchronization message exchange.

```
(* [Process] :: SERVER TIMESYNC MODULE :: [NTS v0.3]
|: replies to a time_request message with a
|: time_response message as specified in NTS v0.3. *)
let server_time_response(B: hostname, pkB: spkey,
                        skB: sskey, seed: bitstring) =

  (* The server receives a time_request message
  from a client. *)
  in(c, Y: bitstring);

  (* It extracts the necessary data. *)
  let (t1_y: bitstring, n_y: bitstring,
      pkA_hash_y: bitstring) = Y in

  (* It creates the appropriate time sync data for
  its response. *)
  new t2: bitstring;

  (* It re-computes the cookie. *)
  let cookie = cookie_gen(pkA_hash_y, seed) in

  (* It composes its response. *)
  let response = (n_y, t1_y, t2,
                  hmac(cookie, (n_y, t1_y, t2)))
  in
```

```
(* It sends its response back to the requesting
client. *)
event serverRespondsTime(n_y, t1_y, t2);
out(c, response).
```

Outer Server Process

We then see the “outer” server process whose purpose is simply to execute iterations of all the “inner” processes (the modules listed above in Subsubsection 1).

```
(* [Process] ::: SERVER GLOBAL PROCESS ::: [NTS v0.3.0]
|: executes all server modules at once, running
|: arbitrarily many instantiations of each of them
|: in parallel. *)
let server(B: hostname) =

  (* Before running any modules, the server generates an
  unpredictable seed value and remembers its own
  key pair. *)
  let seed = seed_of(B) in
  let skB = ssk_of(B) in
  let pkB = spk(skB) in

  (* The server then runs all modules. *)
  !server_certifier(B, pkB, skB)
  | !server_cookie(B, pkB, skB, seed)
  | !server_time_response(B, pkB, skB, seed).
```

The Client Side Processes

Inner Client Process

Moving on to the client side processes, there is first the “inner” process which takes care of the time synchronization message exchange, including the necessary checks on the MAC.

```
(*-----+
|--- Client Side Processes ---|
+-----*)

(* [Process] ::: CLIENT TIMESYNC MODULE ::: [NTS v0.3.0]
|: Generates time_request messages as specified in
|: NTS v0.3 and sends them to a time server. It then
|: awaits a time_response message on which it performs
|: the necessary checks as specified. *)
let client_time_request(A: hostname, pkA: pkey,
  B: hostname, cookie: key) =

  (* The client generates time data and a nonce. *)
  new t1: bitstring;
  new n1: bitstring;

  event client_timereq(t1);

  (* The client constructs its time_request message and
```

```

sends it. *)
let request = (t1, n1, keyhash(pkA)) in
out(c, request);

(* It receives a time_response message and
extracts the necessary information. *)
in(c, X: bitstring);
let (=n1, =t1, t2x: bitstring, hmacx: bitstring)
  = X in

(* Depending on the result of validity checks, it
either accepts the response as authentic or
discards it. *)
if hmacx = hmac(cookie, (n1, t1, t2x))
then event clientAcceptsTime(n1, t1, t2x)
else event clientDiscards(X).

```

Outer Client Process

The “outer” client side process follows, which performs the initial message exchanges (server certification and cookie exchange) and then executes instantiations of the inner process.

```

(* [Process] :: CLIENT GLOBAL PROCESS :: [NTS v0.3.0]
|: Executes the steps for association, certification
|: and cookie exchange, one of each and sequentially.
|: Then it executes arbitrarily many instances of
|: the client timesync module in parallel. *)
let client(A: hostname, B: hostname) =

  let skA = sk_of(A) in
  let pkA = pk(skA) in

  let pkTA = spk(ssk_of(TA)) in

  (* CERTIFICATE PHASE -----*)

  (* The client sends a client_cert message,
  as specified in NTS v0.3. *)
  let msg_client_cert = (version_1, A) in
  out(c, msg_client_cert);

  (* The client receives a response of type
  server_cert. *)
  in(c, X_server_cert: bitstring);

  (* The client extracts data from the response. *)
  let (=A, certificate_x: bitstring,
      signature_x: bitstring)
    = X_server_cert in

  (* The client reads the certificate. *)
  let (=B, other_key: pkey, spkB_x: spkey,
      cert_signature: bitstring)
    = certificate_x in

  (* The client performs the necessary test.
  On failure, it exits with an error.
  On success, the client accepts the key
  given in the certificate as B's public key. *)
  event check();
  if ((B, other_key, spkB_x)

```

```

    <> checksign(cert_signature, pkTA))
  || ((A, certificate_x)
    <> checksign(signature_x, spkB_x))
  then event clientRejectsCert()

else event clientAcceptsCert(B, spkB_x);

(* COOKIE PHASE -----*)
let pkB = spkB_x in

(* The client sends a client_cook
message as specified. *)
new n_cook: bitstring;
let msg_client_cook = (n_cook, pkA)
in
out(c, msg_client_cook);

(* The client receives a response
of type server_cook. *)
in(c, X_server_cook: bitstring);

(* It decodes the response and extracts
the data from it. *)
let X_dec = adec(X_server_cook, skA)
in
let ((cookie_x: key, =n_cook),
signature_x: bitstring) = X_dec
in

(* It performs the necessary checks as
specified. On success, it starts sending
out time_request messages as specified. *)
if (cookie_x, n_cook)
= checksign(signature_x, pkB)
then event
  clientAcceptsCookie(cookie_x, A, B)
(* TIMESYNC PHASE -----*)
| !client_time_request(A, pkA,
  B, cookie_x)
| ( in(c, =cookie_x);
  event cookieCompromised(cookie_x)

else event
  clientRejectsCookie(cookie_x, A, B)).

```

Note that the first `else`-branch includes all the code below it. Note also the dedicated listener process given by

```

| ( in(c, =cookie_x);
  event cookieCompromised(cookie_x)

```

which is started when the client accepts a cookie and does not really represent client behavior according to the protocol, but only listens for the cookie on an open channel. This enables us to check for the loss of a cookie by querying whether the event `cookieCompromised()` is ever executed at all (see Subsubsection 1).

The Environment Process

The ProVerif queries, which make up the next part of the code, are actually not a fixed part of it. In fact, it is most practical to have only one query in the code at any given time. But here we want to present a set of different queries that were used during the verification process and talk about them in more detail.

For this reason the queries are presented last, in subsection 1. Here, we present instead the global ProVerif process which takes care of instantiating all participants, although this process actually needs to be at the very end of any ProVerif code.

```
(*-----+
|----- MAIN PROVERIF PROCESS -----|
|--- MUST ALWAYS BE AT THE END OF SOURCE CODE ---|
+-----*)

(* [Process] MAIN OVERALL PROCESS ::: [NTS v0.3.0]
|: Runs everything that needs to be run. *)
process

(* More strongly typed version with hostnames,
would otherwise be "bitstring" type variables *)
new B: hostname;
new A: hostname;

(* There are arbitrarily many clients running,
but only one server. This is for simplification
in the earlier phase of this document. *)
!server(B) | !client(A, B) | !authority()
| out(c, A) | out(c, B) )
```

ProVerif Queries

Now we present the ProVerif queries. We first consider those queries that concern the cookie exchange.

Sanity – Cookie Exchange

This query makes sure that there is some cookie x which is accepted by the honest client A as coming from the honest server B , i.e. the cookie exchange can be completed successfully.

```
query x: key;
event(clientAcceptsCookie(x, new A, new B)).
```

This query holds for all four code versions.

Weak Authenticity – Cookie

This query ensures that if the honest client A accepts a cookie x for communication with the honest server B , then B has in fact generated x and released it into the network (note that this gives no guarantee that B intended x for communication with A in particular).

```
query x: key;
    event(clientAcceptsCookie(x, new A, new B))
==> event(serverGeneratesCookie(x, new B)).
```

Applying this query to the different ProVerif code versions yields the following results:

- It does **not** hold for code version c030ut. The attack that ProVerif discovers is the one described in Subsection ??.
- It holds for the code versions c030, c031 and c032.

Authenticity – Cookie

This query strengthens the guarantee acquired with the previous query: it ensures that if the honest client A accepts a cookie x for communication with the honest server B , then B has in fact issued x based on A 's public key, and has also encrypted the appropriate message with said public key.

```
query x: key;
    event(clientAcceptsCookie(x, new A, new B))
==> event(serverSaysCookie(x, pk(sk_of(new A)), new B)).
```

The results for this query are as follows:

- It does **not** hold for code version c030ut. Authenticity for the cookie would require weak authenticity for it, which is not given (see above). Also, the Man-in-the-Middle attack described in Subsection ?? works on this version as well as for code version c030 (see below).
- It does **not** hold for code version c030. The corresponding attack is the Man-in-the-Middle attack described in Subsection ??.
- It holds for the code versions c031 and c032

Secrecy – Cookie

This query asserts that if the honest client A accepts a cookie x , then the attacker does not know x . This is realized via the event `cookieCompromised()` from the dedicated listener process as described in Subsection 1.

```
query x: key;
event(cookieCompromised(x)).
```

This query does **not** hold for code version c030ut and **neither** does it hold for c030: Both of the possible attacks enable Mallory to make Alice accept a cookie that Mallory knows. In the case of the Blind-signature attack he manufactures said cookie himself; in the case of the Man-in-the-Middle attack he maliciously re-distributes a valid key, signed by Bob.

- This query holds for the code versions c031and c032.

Next, we take a look at some queries that concern the time synchronization message exchange.

Sanity – Time Synchronization

This query checks whether it is possible for the protocol to be run such that the honest client A successfully accepts a timesync response as valid and authentic from the honest server B .

```
query nonce: bitstring, x: bitstring, y: bitstring;
event(clientAcceptsTime(new A, new B, nonce, x, y)).
```

This query holds for all four code versions.

Authenticity – Time Synchronization

This query ensures that if a timesync message t is accepted by the honest client A as authentic from an honest server B , then B has actually issued a message with the exact time data as in t and secured it with the cookie which is generated based on A 's public key.

```
query nonce: bitstring, x: bitstring, y: bitstring;
event(clientAcceptsTime(new A, new B, nonce, x, y))
==> event(serverRespondsTime(keyhash(pk(sk_of(new A))),
                             new B, nonce, x, y)).
```

As might be expected due to the lack of cookie secrecy, this query also does **not** hold for code version c030ut and **neither** does it hold for c030. Since for these code versions Mallory can gain access to cookies that Alice accepts as valid, he can use those cookies to generate time synchronization packets with maliciously manufactured synchronization information that Alice will accept.

- This query holds for the code versions c031and c032.