```
(* ===== TYPE DECLARATIONS ===== *)
type principal.
type nonce.
type puf_challenge.
type puf_response.
type session_key.
type temp_identity.

(* ===== FUNCTION DECLARATIONS ===== *)

(* Hash function *)
fun h(bitstring): bitstring.

(* PUF function - device-specific and unclonable *)
fun puf(principal, puf_challenge): puf_response [private].

(* Concatenation *)
fun concat(bitstring, bitstring): bitstring.
fun concat3(bitstring, bitstring, bitstring): bitstring.

(* XOR operation *)
fun xor(bitstring, bitstring): bitstring.

(* Session key derivation from PUF responses *)
fun derive_session_key(puf_response, nonce): session_key [private].

(* Type conversions *)
fun challenge_to_bits(puf_challenge): bitstring [data, typeConverter].
fun response_to_bits(puf_response): bitstring [data, typeConverter].
fun nonce_to_bits(nonce): bitstring [data, typeConverter].
fun key_to_bits(session_key): bitstring [data, typeConverter].
fun principal_to_bits(principal): bitstring [data, typeConverter].
fun tid_to_bits(temp_identity): bitstring [data, typeConverter].

(* ===== EQUATIONS ===== *)
equation forall x: bitstring, y: bitstring; xor(xor(x,y),y) = x.
equation forall x: bitstring; xor(x,x) = x.

(* ===== CONSTANTS ===== *)
const Q_GW: bitstring.  (* Registration request *)
const legitimate_user: principal.    (* Pre-defined legitimate user *)
const legitimate_sensor: principal.  (* Pre-defined legitimate sensor *)

(* ===== CHANNELS ===== *)
free c: channel.                    (* Public channel *)
free secure_reg: channel [private]. (* Secure registration channel *)

(* ===== TABLES FOR PERSISTENT STORAGE ===== *)
```

```
table sensor_registry(principal, puf_challenge, puf_response).
table user_registry(principal, temp_identity, puf_challenge, puf_response).

(* ===== EVENTS ===== *)

(* Core security events *)
event sensor_controls_challenge(principal, puf_challenge).
event sensor_registered_securely(principal, puf_challenge, puf_response).
event user_registered_securely(principal, temp_identity, puf_challenge, puf_response).

(* Authentication events *)
event user_auth_initiated(principal, temp_identity, nonce).
event gateway_authenticated_user(principal, principal, nonce).
event sensor_authenticated_gateway(principal, principal, puf_response).
event session_key_established(principal, principal, principal, session_key).

(* Security events *)
event unauthorized_puf_blocked(principal, puf_challenge).

(* ===== QUERIES ===== *)

(* 1. Session key secrecy - MOST IMPORTANT *)
query secret SK.

(* 2. Authentication ordering - proper protocol flow *)
query u: principal, gw: principal, sn: principal, sk: session_key,
      tid: temp_identity, nu: nonce, resp: puf_response;
  event(session_key_established(u, gw, sn, sk)) ==>
  (event(user_auth_initiated(u, tid, nu)) &&
   event(sensor_authenticated_gateway(sn, gw, resp))).

(* 3. Strong Registration prerequisite - authentication requires prior registration *)
query u: principal, sn: principal, sk: session_key,
      c_sn: puf_challenge, r_sn: puf_response,
      c_u: puf_challenge, r_u: puf_response, tid: temp_identity;
  event(session_key_established(u, u, sn, sk)) ==>
  (event(sensor_registered_securely(sn, c_sn, r_sn)) &&
   event(user_registered_securely(u, tid, c_u, r_u))).

(* 4. Legitimate entities only - restrict to pre-defined entities *)
query u: principal, sn: principal, sk: session_key;
  event(session_key_established(u, u, sn, sk)) ==>
  (u = legitimate_user && sn = legitimate_sensor).

(* 5. Sensor challenge control *)
query sn: principal, c: puf_challenge;
  event(sensor_controls_challenge(sn, c)) ==> (sn = legitimate_sensor).

(* ===== PROTOCOL PROCESSES ===== *)

```

```
(* ENHANCED SENSOR REGISTRATION - Core Innovation *)
let enhanced_sensor_registration(sn: principal) =
  (* Restrict registration to legitimate sensor only *)
  if sn = legitimate_sensor then
  (
    (* Wait for legitimate registration request *)
    in(c, =Q_GW);

    (* KEY ENHANCEMENT: Sensor generates its own challenge *)
    new C_sn: puf_challenge;
    event sensor_controls_challenge(sn, C_sn);

    (* Compute PUF response *)
    let R_sn = puf(sn, C_sn) in

    (* Send via secure channel to prevent interception *)
    out(secure_reg, (sn, C_sn, R_sn));

    (* Store in registry for authentication *)
    insert sensor_registry(sn, C_sn, R_sn);
    event sensor_registered_securely(sn, C_sn, R_sn);
    0
  ).

(* USER REGISTRATION *)
let user_registration(u: principal) =
  (* Restrict registration to legitimate user only *)
  if u = legitimate_user then
  (
    (* Send identity to gateway *)
    out(c, u);

    (* Receive challenge from gateway *)
    in(c, C_u: puf_challenge);

    (* Compute PUF response *)
    let R_u = puf(u, C_u) in
    out(c, R_u);

    (* Receive temporary identity *)
    in(c, TID_u: temp_identity);

    (* Store registration data *)
    insert user_registry(u, TID_u, C_u, R_u);
    event user_registered_securely(u, TID_u, C_u, R_u);
    0
  ).

(* GATEWAY REGISTRATION HANDLER *)
let gateway_registration() =
```

```
  (* Handle sensor registration - only legitimate sensor *)
  (
    out(c, Q_GW);
    in(secure_reg, (sn: principal, C_sn: puf_challenge, R_sn: puf_response));
    (* Verify it's the legitimate sensor *)
    if sn = legitimate_sensor then
    0
  )
  |
  (* Handle user registration - only legitimate user *)
  (
    in(c, u: principal);
    if u = legitimate_user then
    (
      new C_u: puf_challenge;
      out(c, C_u);
      in(c, R_u: puf_response);
      new TID_u: temp_identity;
      out(c, TID_u);
      0
    )
  ).

(* ENHANCED AUTHENTICATION PROTOCOL *)
let enhanced_authentication(u: principal, sn: principal) =
  (* Ensure only legitimate entities can authenticate *)
  if u = legitimate_user && sn = legitimate_sensor then
  (
    (* Lookup user registration *)
    get user_registry(=u, TID_u, C_u, R_u) in

    (* Lookup sensor registration - uses sensor-controlled challenge *)
    get sensor_registry(=sn, C_sn, R_sn) in

    (* CRITICAL: Ensure registration events happened first *)
    event user_registered_securely(u, TID_u, C_u, R_u);
    event sensor_registered_securely(sn, C_sn, R_sn);

    (* Step 1: User initiates authentication *)
    new N_u: nonce;
    out(c, (TID_u, N_u));
    event user_auth_initiated(u, TID_u, N_u);

    (* Step 2: Gateway responds with user's registered challenge *)
    in(c, (recv_C_u: puf_challenge, N_g: nonce, V_0: bitstring));

    (* Verify received challenge matches registered challenge *)
    if recv_C_u = C_u then
    (
      (* Step 3: User computes fresh PUF response and verifies gateway *)
```

```
      let R_u_fresh = puf(u, C_u) in
      let V_0_check = h(concat3(
        nonce_to_bits(N_g),
        response_to_bits(R_u_fresh),
        nonce_to_bits(N_u)
      )) in

      if V_0 = V_0_check then
      (
        (* Gateway authenticated - send proof to gateway *)
        let user_proof = h(concat(
          principal_to_bits(u),
          response_to_bits(R_u_fresh)
        )) in
        out(c, (user_proof, principal_to_bits(sn)));

        (* Step 4: Gateway initiates sensor authentication *)
        in(c, (n_1_enc: bitstring, C_sn_recv: puf_challenge, V_2: bitstring));

        (* Verify sensor receives its own registered challenge *)
        if C_sn_recv = C_sn then
        (
          (* Step 5: Sensor authenticates gateway using registered challenge *)
          let R_sn_fresh = puf(sn, C_sn) in
          let n_1 = xor(n_1_enc, response_to_bits(R_sn_fresh)) in
          let V_2_check = h(concat(
            response_to_bits(R_sn_fresh),
            n_1
          )) in

          if V_2 = V_2_check then
          (
            (* Gateway authenticated by sensor *)
            event sensor_authenticated_gateway(sn, u, R_sn_fresh);

            (* Step 6: Establish session key using both PUF responses *)
            let SK = derive_session_key(R_sn_fresh, N_u) in
            event session_key_established(u, u, sn, SK);

            (* Send sensor confirmation *)
            let sensor_proof = h(concat(
              principal_to_bits(sn),
              response_to_bits(R_sn_fresh)
            )) in
            out(c, sensor_proof);
            0
          )
        )
      )
    )
```

```
  ).

(* GATEWAY PROCESS *)
let gateway_process() =
  (* Step 2: Handle user authentication request *)
  in(c, (TID_u: temp_identity, N_u: nonce));

  (* Lookup user from temporary identity - ensure it's legitimate user *)
  get user_registry(u, =TID_u, C_u, R_u) in
  if u = legitimate_user then
  (
    new N_g: nonce;
    let V_0 = h(concat3(
      nonce_to_bits(N_g),
      response_to_bits(R_u),
      nonce_to_bits(N_u)
    )) in
    out(c, (C_u, N_g, V_0));
    event gateway_authenticated_user(u, u, N_g);

    (* Step 4: Handle user proof and initiate sensor auth *)
    in(c, (user_proof: bitstring, sn_bits: bitstring));

    (* Lookup sensor from proof - ensure it's legitimate sensor *)
    get sensor_registry(sn, C_sn, R_sn) in
    if sn = legitimate_sensor then
    (
      new n_1: nonce;
      let n_1_enc = xor(nonce_to_bits(n_1), response_to_bits(R_sn)) in
      let V_2 = h(concat(
        response_to_bits(R_sn),
        nonce_to_bits(n_1)
      )) in
      out(c, (n_1_enc, C_sn, V_2));

      (* Step 6: Receive sensor confirmation *)
      in(c, sensor_proof: bitstring);
      0
    )
  ).

(* DOS PROTECTION - Blocks unauthorized PUF queries *)
let dos_protection(sn: principal) =
  in(c, (challenge: puf_challenge, source: bitstring));
  if source <> Q_GW then
  (
    event unauthorized_puf_blocked(sn, challenge);
    0
  ).

```

```
(* ===== MAIN PROCESS ===== *)

process
  (* Create legitimate principals using pre-defined constants *)
  (
    (* Registration phase - establishes sensor-controlled challenges *)
    enhanced_sensor_registration(legitimate_sensor) |
    user_registration(legitimate_user) |
    gateway_registration() |

    (* Authentication phase - uses registered challenges *)
    enhanced_authentication(legitimate_user, legitimate_sensor) |
    gateway_process() |

    (* Security mechanisms *)
    dos_protection(legitimate_sensor)
  )
```