Modeling and Verification of the Worth-One-Minute Security Protocols

Giorgia Remedi, Alessandro Aldini, Alessandro Bogliolo, Saverio Delpriori, and Lorenz Cuno Klopfenstein

University of Urbino Carlo Bo, Urbino, Italy
name.surname@uniurb.it

Abstract

Worth One Minute (WOM) is a general-purpose rewarding system based on the exchange of anonymous vouchers. It is designed to support crowd-sensing applications that rely on the willingness of users to participate and invest in common causes. The platform rewards the effort of users toward such causes, thus triggering virtuous circles contributing to the expected social value. The system's reliability depends on the security conditions of the voucher issuing and spending processes. These processes are based on two cryptographic protocols, which are discussed in this paper and formally validated through the automated verification tool ProVerif.

1 Introduction

The evolution and pervasiveness of network security protocols for multi-agent, distributed systems are one of the consequences of the development of user-centric and data-driven Web applications that manage sensitive information. Very often, how reliable these applications appear when dealing with sensitive information in respect of security-related issues represents one of the key elements of their success and widespread usage. Also, cryptographic protocols are an essential branch of cybersecurity that, nowadays, must offer a wide range of guarantees about privacy, authentication, confidentiality, and integrity, among others. When these conditions are met, they contribute to the trustworthiness of the networks and the distributed applications on which sensitive information is exchanged.

The complexity of these systems, their security requirements, and the adversarial conditions make the application of rigorous verification techniques fundamental. They have indeed demonstrated to play a crucial role in the analysis of apparently trivial security protocols developed over 30 years ago [2]. Fortunately, nowadays it is becoming common practice to accompany the design of new security architectures with the early formal verification of their robustness with respect to several types of cyber attacks [3, 5, 10]. The mathematical foundations behind formal verification often represent an obstacle to the wide dissemination of these methodologies in non-academic environments, so that their applicability to real-world case studies is just as important as their formal correctness and completeness.

1.1 Crowdsourcing and incentives

Many different kinds of systems rely on the participation of volunteers in order to complete tasks and achieve their objectives. Tasks that are highly parallel in nature are especially well-suited to being carried out by a distributed work force. Some of these tasks may require volunteers to solve problems, collect data, perform measurements, or to provide creative input. This approach, also known as "crowdsourcing", has been widely seen as a radical new paradigm of problem-solving that harnesses the contributions of a distributed network of individuals [8].

In recent years, modern technology and the Web have been adopted to harvest distributed intellect in order to collect huge amounts of knowledge (such as in Wikipedia), to solve creative problems, or to perform menial tasks (see Amazon's 'Mechanical Turk'). Other examples of this approach include "volunteer computing", where volunteers offer part of the hardware resources of their electronic devices in order to solve difficult problems [4], or "crowdsensing", whereby volunteers participate in large-scale sensing initiatives by sharing local information about phenomena—such as traffic, air quality, or road surface quality—that they can either perceive directly or measure through their mobile devices [15, 16]. Distributed computing and sensing tasks have caught on over the last years thanks to the ubiquitous presence of devices such as smartphones, which are capable of providing notable computational power and sensing capabilities.

Participants in a 'crowd'-based system may devote significant efforts towards the shared task, in the form of time, attention, hardware resources, creativity, or even giving access to private information. Individuals may not be willing to contribute to the common task without strong incentives for cooperation: thus, a primary aspect of crowdsourced systems is that of providing incentives and reward schemes for participants, in order to ensure a number of users that is sufficient to guarantee coverage and reliability [12].

In this setting, a new platform, called "Worth One Minute" (WOM)[1], has been proposed recently to provide a general-purpose incentives system based on the exchange of anonymous vouchers, simply called WOMs, each of which represents a reward for 'one minute' of effort towards a common social goal. The platform allows any volunteer-based initiative to reward the efforts of their participants, endowing them with vouchers that can be collected and spent within a community that promotes and encourages volunteer activities aimed at the common good [13].

The critical security aspects of such a platform are concerned with the privacy of individuals (no personal, sensitive information shall be disclosed as a consequence of the contributions provided and of the vouchers gained and spent) and the reliability of the vouchers management system (no vouchers shall be forged, stolen, double spent, revealed to unauthorized parties).

1.2 Contribution

The security conditions of the WOM platform are guaranteed by means of ad-hoc cryptographic protocols governing the interactions among the various parties, the complexity of which requires strong verification. In this paper, we show how to formally verify some authentication and confidentiality properties of these protocols, by using the proof-theoretic approach implemented in the software tool ProVerif [7].

In the following, we discuss the main features of the Worth One Minute infrastructure (Section 2) and of the underlying cryptographic protocols. Then, we present the formal specifications and verification of these protocols through ProVerif (Section 3). Some remarks and feedback about such a modeling and analysis experience are finally reported (Section 4).

2 The WOM Platform

The WOM platform is designed to provide an open and participative infrastructure, which is not bound to a single crowd-based initiative and leverages platform economy mechanics in order to multiply its impact. By decoupling the crowdsourcing system from its incentive scheme, the

[1] Official Web site: https://wom.social.

(a) $CT \rightarrow AS : (c_1, c_2, \ldots, c_n, Pwd)_{SK_{ctas}}$

(b) $AS \rightarrow REG : (c'_1, c'_2, \ldots, c'_n, Pwd, N, ID_{instr})_{PK_{reg}}$

(c) $REG \rightarrow AS : (N, OTC_{gen}, ID_{reg})_{PK_{instr}}$

(d) $AS \rightarrow REG : (OTC_{gen})_{PK_{reg}}$

(e) $AS \rightarrow CT : (OTC_{gen})_{SK_{ctas}}$

(f) $CT \rightarrow PCK : (OTC_{gen})$

(g) $PCK \rightarrow REG : (OTC_{gen}, Pwd, K_s)_{PK_{reg}}$

(h) $REG \rightarrow PCK : (v_1, v_2, \ldots, v_n)_{K_s}$

(i) $REG \rightarrow AS : (N, confirm)_{PK_{instr}}$

Table 1: Communication protocol for vouchers generation.

platform serves as a link between volunteers (who may contribute to one or more common good initiatives) and stakeholders (who wish to support the efforts of volunteers).

The platform is intended to attract volunteer-based initiatives on one side, allowing users to gain vouchers, and third-party subsidizers on the other one, allowing users to spend vouchers. In the context of the WOM platform, the crowd-based tools are called "Instruments" and subsidizers are known as "Merchants". While the platform is open to any volunteer and any Merchant, the addition of a new Instrument to the platform is carefully evaluated by a transparent ethical committee that evaluates whether the Instrument is compatible with the purpose of the platform and the shared goals of subsidizers.

2.1 WOM Generation and Spending

The main operations of the WOM platform are concerned with voucher generation and transfer to the intended user, in response to a legitimate request, as well as voucher verification and spending during a transaction between a user and a Merchant. These operations rely on the execution of two protocols [14], which are detailed in Tables 1 and 2, respectively.

The protocols require the interaction among multiple participants. In particular: the Instrument's client (CT), which allows the user to perform the intended task and collects related information, and the Instrument's aggregation service (AS), which gathers volunteer contributions and validates them in order to enable the generation of the related vouchers as a reward. The Registry (REG) is the central trusted authority of the WOM platform, which manages vouchers and payments. The Pocket (PCK) is a client-side software controlled by the volunteer, that allows him/her to collect and spend vouchers independently of the specific Instruments that have been used previously to gain them. Finally, the Point of Sale (POS) allows Merchants to request new payments, which can then be performed by volunteers by spending vouchers stored in their Pocket.

Table 1 describes the protocol that allows Instruments to gain new vouchers. In steps (a) and (b) the two Instrument-side components CT and AS exchange and transmit the user's contributions (represented by the abstract terms c_1, c_2, …, c_n) and a fresh password Pwd (known by the user) to the Registry. Even if they do not include any user identity related

(j) $POS \rightarrow REG : (f, k, ACK_{pck}, Pwd, N, ID_{pos})_{PK_{reg}}$

(k) $REG \rightarrow POS : (N, OTC_{pay}, ID_{reg})_{PK_{pos}}$

(l) $POS \rightarrow REG : (OTC_{pay})_{PK_{reg}}$

(m) $POS \rightarrow PCK : (OTC_{pay})$

(n) $PCK \rightarrow REG : (OTC_{pay}, Pwd, K_{s_1})_{PK_{reg}}$

(o) $REG \rightarrow PCK : (f, k, ID_{pos})_{K_{s_1}}$

(p) $PCK \rightarrow REG : (OTC_{pay}, Pwd, v_1, v_2, \ldots, v_k, h(S), K_{s_2})_{PK_{reg}}$

(q) $REG \rightarrow POS : (N)_{PK_{pos}}$

(r) $REG \rightarrow PCK : (ACK_{pck})_{K_{s_2}}$

Table 2: Communication protocol for spending vouchers.

information, contributions refer to a specific task (*what* common cause was pursued) and *where*/*when* they have been generated. Hence, in order to trade for user's privacy, they may be (partially) obfuscated, turning them into c'_1, c'_2 ...,c'_n, before their transfer to the central authority. Notice that communications between CT and AS are secured by the Instrument, by encrypting them with the shared, secret key SK_{ctas} of a symmetric key algorithm. At step (b) a unique *nonce* N is generated and the identity of the Instrument is added to the message that is sent to the central authority, encrypted with the public key PK_{reg} of the Registry[2]. Steps (c) to (e) are needed to transmit a unique one-time code (OTC_{gen}) back to the Instrument. In essence, this code identifies a pending voucher request and takes the form of an URL facilitating the interactions between mobile agents and the central authority. Notice that message (c) includes the nonce N, the identity of the Registry, and is encrypted with the public key PK_{instr} of the Instrument. After the handshake phase between Instrument and Registry, the Instrument transmits the code as a clear text to the user's Pocket, see step (f), which will use it to enable the subsequent voucher claim. On step (g), the Pocket asks the user for the password Pwd again, and then provides the couple OTC_{gen} and Pwd to the Registry. Message (g) includes a fresh key K_s, chosen by PCK. The Registry checks the validity of the pair (OTC_{gen}, Pwd) and, if the check is passed, sends back the set of vouchers v_1, v_2, ..., v_n (corresponding to rewards for c'_1, c'_2, ..., c'_n, respectively), encrypted with the key K_s. Finally, on step (i) the Registry confirms to AS that the vouchers have been generated and collected.

With respect to the security conditions of interest, it is worth verifying whether the whole process guarantees the confidentiality of the vouchers generated by the Registry, which shall be claimed by and made available to the legitimate user only, without any chance for the adversary to even distinguish between the issuance of different vouchers. Moreover, to ensure both the user's privacy and reliability of the protocol phase during which the user's contributions are exchanged, the authentication between Instrument and Registry must be guaranteed. During the protocol execution, it can be assumed that the unique trustworthy party is the Registry central authority, which cannot be impersonated by an adversary.

The second protocol, shown in Table 2, describes the mechanism by which Merchants can

[2]With the usual notation PK_a, SK_a we mean the pair of public and secret keys of an asymmetric encryption algorithm, respectively, for agent a.

request new payments for services delivered to the users. The protocol bears many similarities to the one in Table 1. On steps (j) to (l), the Merchant's Point of Sale (POS) and the Registry agree on a new payment request, which is specified in terms of how many (parameter k) and which type (parameter f) of vouchers are required. The type can be related to the *what/where/when* dimensions associated with the user's contributions from which the vouchers derive. The POS component also supplies a code ACK_{pck} which represents the value that will be disclosed to the Pocket on a successful payment. Similarly as in steps (a) and (b) of the previous protocol, in step (j) the POS component also sends a user-specified password Pwd, a fresh *nonce* N, and her identity ID_{pos}. In response to the request, the Registry issues a unique one-time code OTC_{pay}, which is then sent on step (m) as a clear text to the user's Pocket after the handshake phase between POS and Registry. The Pocket component employs the one-time code to request detailed information about the payment on step (n), by exposing also the password specified by the user, and a fresh session key K_{s_1}. In the following step (o) the Registry checks the pair (OTC_{pay}, Pwd) and, if the check is passed, returns to the Pocket the parameters f and k, along with the Points of Sale identity ID_{pos}, encrypted with the session key K_{s_1}. On step (p) OTC_{pay}, Pwd, a set of vouchers v_1, v_2, ..., v_k fulfilling the requirements, the hash $h(S)$ of a random value S chosen by the Pocket, and a fresh session key K_{s_2} are sent by the Pocket to the Registry, which then confirms the payment to the POS on step (q), provided that the vouchers are valid and fulfil the requirements. At the end of the exchange, on step (r) the PCK component obtains ACK_{pck} as confirmation of the payment and stores privately the secret S in case of future disputes with the merchant.

The security properties that require validation are similar to the previous case. Indeed, it is worth verifying the authentication between Merchant's POS and Registry, as well as the reliability of the payment, which must be completed by using legitimate vouchers that can be neither stolen nor identified by the adversary. Again, it is worth assuming that the Registry is the unique trustworthy agent involved in the protocol execution.

3 Formal Analysis

The formal verification of the security properties of the two WOM protocols has been conducted through ProVerif [7], a software tool for the specification and analysis of cryptographic protocols, encompassing features like, e.g., modeling of several cryptographic primitives (symmetric and asymmetric encryption, digital signatures, hash functions, nonces), expressing families of secrecy and authentication properties [18], modeling of unbounded, even concurrent, number of protocol sessions. Formally, ProVerif specifications, given in a typed version of the Applied Pi calculus [1], are turned into Horn clauses, while the validity of the security properties is demonstrated through logical derivation by applying resolution techniques, as in logic programming [6]. The adversary model relies on the Dolev and Yao assumptions [11], meaning that:

- the attacker controls the communication means and can read, drop, modify, create, and transmit packets by acting as a man-in-the-middle;
- cryptographic operations represent perfect black boxes, i.e., the attacker cannot access nor modify the content of encrypted messages (or encrypt new messages) without knowing the related key and, in general, cannot perform cryptanalysis attacks.

In the following section, we specify and analyze in detail the voucher generation protocol of Table 1. The reader interested also in the spending protocol of Table 2 can refer to the official website [17].

3.1 ProVerif Verification: Voucher Generation Protocol

ProVerif specifications include three sections: declarations, process macros, and the main process. Declarations specify names and types:

```
free ch: channel. (* Public channel of communication *)
(* Types *)
type nonce.       (* Nonce *)
type pkey.        (* Public key *)
type skey.        (* Secret key *)
(* Names and costants *)
free cn: bitstring [private]. (* contributions *)
free vn: bitstring [private]. (* vouchers *)
free confirm: bool.           (* final confirmation *)
(* Type converter *)
fun nonce_to_bitstring(nonce): bitstring [data,typeConverter].
```

Notice that both user's contributions and vouchers are abstracted by terms `cn` and `vn`, respectively. The keyword `[private]` excludes the name to which it is applied from the adversary's knowledge. The declaration section includes also the definition of the functions formalizing the behavior of the cryptographic primitives:

```
(* Public key encryption *)
fun pk(skey): pkey.
fun aencrypt(bitstring, pkey): bitstring.
reduc forall x: bitstring, y: skey; adecrypt(aencrypt(x, pk(y)), y) = x.
(* Shared key encryption *)
fun sencrypt(bitstring, skey): bitstring.
reduc forall x: bitstring, y: skey; sdecrypt(sencrypt(x, y), y) = x.
```

Secrecy assumptions are then added to specify initial limitations of the adversary, who cannot know the secret key `SKctas` shared by *CT* and *AS*, the secret keys for asymmetric encryption possessed by *AS* (`SKinstr`) and *REG* (`SKreg`), and the password `Pwd` known by the user. Notice that the use of such assumptions preserves soundness of the verification, because the tool also checks that these secrets cannot be derived at any time.

```
(* Secrecy assumptions *)
not attacker(new SKctas).
not attacker(new SKinstr).
not attacker(new SKreg).
not attacker(new Pwd).
```

Then, the declarations section terminates with the specification of the security properties of interest, which are defined as queries. The first set of properties is related to the secrecy of critical information:

```
(* Secrecy queries *)
(* To test secrecy of critical information *)
query secret Pwdreg;    (* Pwd chosen at the Instrument side *)
      secret Nreg;      (* Nonce exchanged by AS and REG *)
      attacker(vn).
noninterf vn. (* To test strong secrecy of vouchers *)
```

The keyword `secret` expresses a query testing the secrecy of bound variables inside the related process, which are, in our case, the password shared by the Instrument components and known to the agent (see name `Pwdreg`), and the nonce exchanged by Instrument and Registry (see name `Nreg`). The meaning and scope of such names will be clear in the next section of the specification dedicated to the formal description of the various agents' behavior. An alternative way to test secrecy is through the command `attacker`, which in our example is used to check whether the vouchers `vn` can be derived by the attacker. In such a case, we also model a strong secrecy property based on a noninterference condition, i.e., we check (through observational equivalence [9]) whether the attacker can even distinguish two sessions differing for the vouchers issued, see the `noninterf` command.

Mutual authentication (between AS and REG) is captured by asserting a one-to-one correspondence between specific events in order to express conditions of the form "*if event* e_2 *occurs, then event* e_1 *has been observed previously*".

```
(* Authentication queries *)
event beginREGAuthparam(pkey).
event endREGAuthparam(pkey).
event beginASAuthparam(pkey).
event endASAuthparam(pkey).
query x: pkey; inj-event(endREGAuthparam(x)) ==> inj-event(beginREGAuthparam(x)).
query x: pkey; inj-event(endASAuthparam(x)) ==> inj-event(beginASAuthparam(x)).
```

The specific nature of the events will be determined within the description of the agents involved in the authentication.

In the second section of the ProVerif specification, process macros are used to define the agents and their behavior in the protocol. First of all, we have the CT agent, described by the following process term `processCollectionTool`:

```
(* Collection Tool *)
let processCollectionTool(SKctas: skey, Pwd: nonce) =
                  (* Message (a): transfer the contributions and the password to AS *)
  out(ch, sencrypt((cn, Pwd), SKctas));
  in(ch, (message_e: bitstring));
                  (* Symmetric decryption with the shared key SKctas *)
  let (OTCgenX: bitstring) = sdecrypt(message_e, SKctas) in
                  (* Message (f): send OTCgenX to PCK in clear *)
  out(ch, OTCgenX).
```

The process has two arguments, which are the secret key `SKctas` of type `skey` and the password `Pwd` of type `nonce`, and is defined in terms of the sequence of message exchanges in which it is involved, either as sender or as receiver (see Table 1). Firstly, the process sends to the public channel `ch` message (a) of the protocol, encrypted with the shared, secret key `SKctas` (see the `out` command), and then waits for the message (e) containing the encrypted one-time code (see the `in` command), which in turn is decrypted and finally sent to PCK in message (f), see the `let` and the subsequent `out` commands. Analogously, we have the AS agent:

```
(* Aggregation Service *)
let processAggregationService(PKreg: pkey, SKinstr: skey, SKctas: skey) =
  in(ch, message_a: bitstring);
                  (* Decrypt message (a) with the shared key SKctas *)
  let (c: bitstring, PwdX: nonce) = sdecrypt(message_a, SKctas) in
                  (* BEGIN AUTHENTICATION WITH REG *)
```

```
                    (* Await to read pkX on channel ch *)
  in(ch, pkX: pkey);
  if pkX = PKreg then
                    (* Start the run of the protocol with PKreg *)
  event beginREGAuthparam(pkX);
                    (* Generate a random nonce N *)
  new N: nonce;
                    (* Message (b): request vouchers generation *)
  out(ch, aencrypt((c, PwdX, N, pk(SKinstr)), pkX));
  in(ch, message_c: bitstring);
                    (* Asymmetric decryption of message (c) with secret key SKinstr
                       Check N and pkX using pattern matching. *)
  let (=N, OTCgenX: nonce, =pkX) = adecrypt(message_c, SKinstr) in
                    (* Message (d): send OTCgenX as confirmation *)
  out(ch, aencrypt(nonce_to_bitstring(OTCgenX), pkX));
                    (* Record the end of the protocol with pkX *)
  event endASAuthparam(pk(SKinstr));
                    (* END AUTHENTICATION WITH REG *)
                    (* Message (e): send OTCgenX to CT *)
  out(ch, (sencrypt(nonce_to_bitstring(OTCgenX), SKctas)));
  in(ch, message_i: bitstring);
                    (* Decrypt message (i) and check the content via pattern matching *)
  let (=N, =confirm) = adecrypt(message_i, SKinstr) in 0.
```

The specification describes the phases of the protocol and the message exchanges in which *AS* is involved, as given in Table 1. For instance, the first message, received from *CT*, is decrypted with `SKctas` and shall include the contributions and the password, represented by names `c` and `PwdX`. Then, the subsequent authenticated handshake is executed with the Registry agent, which is identified by the public key read from the channel, see name `pkX` and the equality check `pkX = PKreg`. Two events denote the beginning (with *REG*) and the end (by *AS*) of such a handshake. The correctness of the content of received messages is verified through pattern matching, e.g., the incoming bitstring `message_c` is correct if it can be decrypted with `SKinstr`, and the resulting three elements are the nonce `N` previously generated and sent, a nonce assigned to name `OTCgenX`, and `pkX` (see the `let` command managing the bitstring `message_c`). In a similar way, we have the description of the agent *REG*:

```
(* Registry *)
let processRegistry(PKinstr: pkey, SKreg: skey) =
                    (* BEGIN AUTHENTICATION WITH AS *)
  in(ch, message_b: bitstring);
                    (* Asymmetric decryption of message (b) with secret key SKreg *)
  let (c: bitstring, PwdY: nonce, NY: nonce, pkY: pkey) = adecrypt(message_b, SKreg) in
                    (* Start the run of the protocol with pkY *)
  event beginASAuthparam(pkY);
                    (* Generate one-time code OTCgen *)
  new OTCgen: nonce;
                    (* Message (c): issue OTGgen needed to claim the new vouchers *)
  out(ch, aencrypt((NY, OTCgen, pk(SKreg)), pkY));
  in(ch, message_d: bitstring);
                    (* Check OTCgen *)
  if nonce_to_bitstring(OTCgen) = adecrypt(message_d, SKreg) then
  if pkY = PKinstr then
                    (* Record the end of the protocol with pkY *)
```

```
  event endREGAuthparam(pk(SKreg));
                    (* END AUTHENTICATION WITH AS *)
                    (* Assign NY and PwdY to new names to test secrecy queries *)
  let Nreg = NY in
  let Pwdreg = PwdY in
  in(ch, (message_g: bitstring));
                    (* Check correctess of OTCgen and PwdY using pattern matching *)
  let (=OTCgen, =PwdY, KsY: skey) = adecrypt(message_g, SKreg) in
                    (* Message (h): vouchers sent to PCK using session key KsY *)
  out(ch, (sencrypt((vn), KsY)));
                    (* Message (i): notify the transfer of vouchers to AS *)
  out(ch, (aencrypt((NY, confirm), PKinstr))).
```

Notice that the Registry is available to start an authenticated session with the agent identified by the public key `pkY` received in message (b). As observed above, the handshake is delimited by the beginning (with AS) and the end (by REG) events, which together with the two events of agent AS are used to define the mutual authentication queries. At the end of the handshake we can find the specification of the names subject to the secrecy queries, i.e., the nonce and the password exchanged with the Instrument. The agent PCK is defined as follows:

```
(* Pocket *)
let processPocket(PKreg: pkey, Pwd: nonce) =
                    (* Await to receive one-time code OTCgen *)
  in(ch, (OTCgenZ: nonce));
  new Ks : skey;    (* Generate a new session key Ks *)
                    (* Message (g): reedem request *)
  out(ch, (aencrypt((OTCgenZ, Pwd, Ks), PKreg)));
  in(ch, (message_h: bitstring));
                    (* Symmetric decryption of vouchers with secret key Ks *)
  let (=vn) = sdecrypt(message_h, Ks) in 0.
```

Finally, the main process describes the whole architecture of the system by declaring the instances of the agents involved and the knowledge possessed by such instances. In particular, the pairs of keys of asymmetric encryption must be declared and the public keys broadcast via the public channel. Moreover, the password `Pwd` must be in the initial knowledge of CT and PCK, while the secret key `SKctas` must be shared by CT and AS.

```
                    (*** MAIN PROCESS ***)
process
                    (* Create the key shared between CT and AS *)
  new SKctas: skey;
                    (* Create the user password used at the client side *)
  new Pwd: nonce;
                    (* Create the secret key of Instrument *)
  new SKinstr: skey;
                    (* Create and transmit on channel ch the public key of Instrument *)
  let PKinstr = pk(SKinstr) in out(ch, PKinstr);
                    (* Create the secret key of Registry *)
  new SKreg: skey;
                    (* Create and transmit on channel ch the public key of Registry *)
  let PKreg = pk(SKreg) in out(ch, PKreg);
  (    (* Launch an unbounded number of sessions of the Collection Tool,
          Aggregation Service, Registry, and Pocket *)
```

```
    (!processCollectionTool(SKctas, Pwd)) |
    (!processAggregationService(PKreg, SKinstr, SKctas)) |
    (!processRegistry(PKinstr, SKreg)) |
    (!processPocket(PKreg, Pwd))
  )
```

The feedback provided by the automated verification reveals a vulnerability because of which the attacker may obtain the secret password `Pwd`. Indeed, in the last outgoing message (f), the CT agent simply forwards, without any check, the one-time code extracted from message (e). However, the attacker may drop message (e) and replay message (a), thus inducing CT to inadvertently disclose the password in message (f), as reconstructed by ProVerif in the form of an execution trace. Checking the content of message (f) is somehow implicit in the description of Table 1, but it cannot be given for granted in the protocol implementation, which could skip such a check because of some form of code optimization. Hence, to avoid the replay attack it is worth verifying that the bitstring in message (f) is not a tuple, by changing the last line of the CT component as follows:

```
   if OTCgenX <> (cn, Pwd) then out(ch, OTCgenX).
```

The automated verification with such a modification demonstrates the satisfaction of the security queries:

```
C:\...\proverif2.00>proverif WOM_vouchersGeneration.pv
RESULT secret Pwdreg is true.
RESULT secret Nreg is true.
RESULT not attacker(vn[]) is true.
RESULT Non-interference vn is true (bad not derivable).
RESULT inj-event(endREGAuthparam(x_25)) ==> inj-event(beginREGAuthparam(x_25)) is true.
RESULT inj-event(endASAuthparam(x_26)) ==> inj-event(beginASAuthparam(x_26)) is true.
```

Therefore, ProVerif has been useful to reveal and avoid a potential attack, thus confirming that it is worth paying attention to the way in which even simple message exchanges are implemented.

4 Conclusion

The complexity of the two WOM cryptographic protocols—in terms of number of agents and communications involved—and the variety of security conditions to verify—from authentication, to secrecy and noninterference—demand a tool-supported formal analysis, which has been conducted in this paper by using ProVerif. The experimental results revealed that the two vouchers management protocols are secure with respect to the properties of interest. Also, by analyzing the voucher generation protocol some potential subtleties emerged to which software developers should pay attention during implementation.

In general, the success of such a kind of verification process strictly depends on the usability and expressiveness of the analysis tool at hand. To give some useful feedback about our experience, the case study presented in this paper has been conducted with the effort of a BSc student in Computer Science with some experience in concurrency theory, logic programming, formal verification, and model checking. The training period required to gain experience and familiarity with ProVerif was about one month, after which the specification of the WOM cryptographic protocols required three more weeks, the first one of which was necessary for the study of the WOM platform. Hence, we believe that this experience shall convince the security protocol/system developer that formal verification is not only mandatory but also feasible with a reasonable effort even in the case of complex systems.

References

[1] Martín Abadi, Bruno Blanchet, and Cédric Fournet. The applied Pi calculus: Mobile values, new names, and secure communication. *Journal of the ACM*, 65(1):1:1–1:41, 2017.

[2] Matteo Avalle, Alfredo Pironti, and Riccardo Sisto. Formal verification of security protocol implementations: a survey. *Formal Aspects of Computing*, 26(1):99–123, 2014.

[3] David Basin, Jannik Dreier, Lucca Hirschi, Saša Radomirovic, Ralf Sasse, and Vincent Stettler. A formal analysis of 5G authentication. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS'18, pages 1383–1396. ACM, 2018.

[4] Adam L. Beberg, Daniel L. Ensign, Guha Jayachandran, Siraj Khaliq, and Vijay S. Pande. Folding@home: Lessons from eight years of volunteer distributed computing. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–8. IEEE, 2009.

[5] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 483–502, 2017.

[6] Bruno Blanchet. Using Horn clauses for analyzing security protocols. In Véronique Cortier and Steve Kremer, editors, *Formal Models and Techniques for Analyzing Security Protocols*, volume 5 of *Cryptology and Information Security Series*, pages 86–111. IOS Press, 2011.

[7] Bruno Blanchet. Modeling and verifying security protocols with the applied Pi calculus and ProVerif. *Foundations and Trends in Privacy and Security*, 1(1-2):1–135, 2016.

[8] Daren C. Brabham. Crowdsourcing as a model for problem solving: An introduction and cases. *Convergence*, 14(1):75–90, 2008.

[9] Vincent Cheval and Bruno Blanchet. Proving more observational equivalences with Proverif. In *Procs of the Second Int. Conf. on Principles of Security and Trust*, POST'13, pages 226–246. Springer-Verlag, 2013.

[10] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A comprehensive symbolic analysis of TLS 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS'17, pages 1773–1788. ACM, 2017.

[11] D. Dolev and A. C. Yao. On the security of public key protocols. In *Procs. of the 22Nd Annual Symposium on Foundations of Computer Science*, SFCS'81, pages 350–357. IEEE, 1981.

[12] Luis G. Jaimes, Idalides J. Vergara-Laurens, and Andrew Raij. A Survey of Incentive Techniques for Mobile Crowd Sensing. *IEEE Internet of Things Journal*, 2(5):370–380, 2015.

[13] Lorenz Cuno Klopfenstein, Saverio Delpriori, Alessandro Aldini, and Alessandro Bogliolo. Introducing a flexible rewarding platform for mobile crowd-sensing applications. In *2018 IEEE International Conference on Pervasive Computing and Communications Workshops*, pages 728–733, Athens, 2018. IEEE.

[14] Lorenz Cuno Klopfenstein, Saverio Delpriori, Alessandro Aldini, and Alessandro Bogliolo. "Worth one minute": An anonymous rewarding platform for crowd-sensing systems. *Journal of Communications and Networks*, 21(5):509–520, October 2019.

[15] Lorenz Cuno Klopfenstein, Saverio Delpriori, Paolo Polidori, Andrea Sergiacomi, Marina Marcozzi, Donna Boardman, Peter Parfitt, and Alessandro Bogliolo. Mobile crowdsensing for road sustainability: exploitability of publicly-sourced data. *International Review of Applied Economics*, pages 1–22, July 2019.

[16] Evangelos Kosmidis et al. hackAIR: Towards Raising Awareness about Air Quality in Europe by Developing a Collective Online Platform. *ISPRS International Journal of Geo-Information*, 7(5), 2018.

[17] DIGIT srl. *Modeling and Verification of the WOM Security Protocols*, 2020. https://digit.srl/modeling-and-verification-of-the-worth-one-minute-security-protocols/.

[18] Thomas Y. C. Woo and Simon S. Lam. Authentication for distributed systems. *Computer*, 25(1):39–52, 1992.

A Specification of the payment protocol

```
(* Communication protocol for spending vouchers *)

(****************************** DECLARATIONS ******************************)

free ch: channel. (* Public channel of communication *)
(* Types *)
type nonce.   (* Nonce *)
type pkey.    (* Public key *)
type skey.    (* Secret key *)
(* Names and costants *)
free f, k: bitstring.         (* payment parameters *)
free vk: bitstring [private]. (* vouchers *)
free ACKpck: bitstring.       (* acknowledgement *)
(* Hash function *)
fun hash(nonce): bitstring.
(* Type converter *)
fun nonce_to_bitstring(nonce): bitstring [data,typeConverter].

(* Public key encryption *)
fun pk(skey): pkey.
fun aencrypt(bitstring, pkey): bitstring.
reduc forall x: bitstring, y: skey; adecrypt(aencrypt(x,pk(y)), y) = x.
(* Shared key encryption *)
fun sencrypt(bitstring, skey): bitstring.
reduc forall x: bitstring, y: skey; sdecrypt(sencrypt(x, y), y) = x.

(* Secrecy assumptions *)
not attacker(new SKpos).
not attacker(new SKreg).
not attacker(new Pwd).

(************************** QUERIES AND EVENTS **************************)

(* Secrecy queries *)
noninterf vk. (* To test strong secrecy of vouchers *)
(* To test secrecy of some terms *)
query attacker(vk).
query secret Pwdreg;   (* Pwd sent to REG *)
      secret Npos.     (* Nonce exchanged by POS and REG *)

(* Authentication queries *)
event beginPOSAuthparam(pkey).
event endPOSAuthparam(pkey).
event beginREGAuthparam(pkey).
event endREGAuthparam(pkey).

query x: pkey; inj-event(endREGAuthparam(x)) ==> inj-event(beginREGAuthparam(x)).
query x: pkey; inj-event(endPOSAuthparam(x)) ==> inj-event(beginPOSAuthparam(x)).

(********************************* MACROS *********************************)
```

```
(* Point of Sale *)
let processPointOfSale(PKreg: pkey, SKpos: skey, Pwd: nonce) =
     (* BEGIN AUTHENTICATION WITH REG *)
     (* Await to read pkX on channel ch *)
  in (ch, pkX: pkey);
  if pkX = PKreg then
     (* Start to run the protocol with pkX *)
  event beginREGAuthparam(pkX);
     (* Generate a random nonce N *)
  new N: nonce;
     (* Message (j): create new payment istance *)
  out(ch, (aencrypt((f, k, ACKpck, Pwd, N, pk(SKpos)), pkX)));
  in(ch, message_k: bitstring);
     (* Asymmetric decryption of message (k) with secret key SKpos.
        Check N and pkX using pattern matching. *)
  let(=N, OTCpayX: nonce, =pkX) = adecrypt(message_k, SKpos) in
     (* Message (l): send OTCpay as confirmation *)
  out(ch, (aencrypt(nonce_to_bitstring(OTCpayX), pkX)));
     (* Record the end of the protocol with pkX *)
  event endPOSAuthparam(pk(SKpos));
     (* END AUTHENTICATION WITH REG *)
     (* Assign N to a new name to test secrecy query *)
  let Npos = N in
     (* Message (m): send OTCpay to PCK in clear *)
  out(ch, OTCpayX);
  in(ch, message_q: bitstring);
     (* Check payment confirmation by REG *)
  let(=N, =PKreg) = adecrypt(message_q, SKpos) in
  0.


(* Registry *)
let processRegistry(PKpos: pkey, SKreg: skey) =
     (* BEGIN AUTHENTICATION WITH POS *)
  in(ch, message_j: bitstring);
  let(=f, =k, =ACKpck, PwdY: bitstring, NY: nonce, pkY: pkey) = adecrypt(message_j, SKreg) in
     (* Start to run the protocol with pkY *)
  event beginPOSAuthparam(pkY);
     (* Generates one-time code OTCpay *)
  new OTCpay: nonce;
     (* Message (k): sends OTCpay to POS *)
  out(ch, (aencrypt((NY, OTCpay, pk(SKreg)), pkY)));
  in(ch, message_l: bitstring);
     (* Check OTCpay *)
  if nonce_to_bitstring(OTCpay) = adecrypt(message_l, SKreg) then
  if pkY = PKpos then
     (* Record the end of the protocol with pkY *)
  event endREGAuthparam(pk(SKreg));
     (* END AUTHENTICATION WITH POS *)
     (* Assign PwdY to a new name to test secrecy query *)
  let Pwdreg = PwdY in
  in(ch, message_n: bitstring);
     (* Check correctness of OTCpay and PwdY using pattern matching *)
```

```
  let (=OTCpay, =PwdY, Ks1Y: skey) = adecrypt(message_n, SKreg) in
     (* Message (o): transfer parameters of payment *)
  out(ch, (sencrypt((f, k, PKpos), Ks1Y)));
  in(ch, message_p: bitstring);
     (* Verify payment *)
  let (=OTCpay, =PwdY, =vk, md: bitstring, Ks2Y: skey) = adecrypt(message_p, SKreg) in
     (* Message (q): notify successful payment to the Merchant *)
  out(ch, (aencrypt(nonce_to_bitstring(NY), PKpos)));
  (* Message (r): confirm payment to the Pocket *)
  out(ch, (sencrypt((ACKpck), Ks2Y))).


(* Pocket *)
let processPocket(PKreg: pkey, PKpos: skey, Pwd: nonce) =
     (* Await on the channel OTCpay *)
  in(ch, (OTCpayZ: nonce));
     (* Generate a fresh session key Ks1 *)
  new Ks1: skey;
     (* Message (n): new request of payment *)
  out(ch, (aencrypt((OTCpayZ, Pwd, Ks1), PKreg)));
  in(ch, message_o: bitstring);
     (* Check payment parameters *)
  let (=f, =k, =PKpos) = sdecrypt(message_o, Ks1) in
     (* Generate a nonce *)
  new S: nonce;
     (* Generate a fresh session key Ks2 *)
  new Ks2: skey;
     (* Message (p): transfer of vouchers *)
  out(ch, (aencrypt((OTCpayZ, Pwd, vk, hash(S), Ks2), PKreg)));
  in(ch, message_r: bitstring);
     (* Symmetric decryption of message (r) with Ks2 *)
  let (=ACKpck) = sdecrypt(message_r, SKreg) in
  0.

(****************************** MAIN PROCESS ******************************)

process
     (* Create the user password used at the client side *)
  new Pwd: nonce;
     (* Create the secret key of Merchant *)
  new SKpos: skey;
     (* Create the public key of Merchant *)
  let PKpos = pk(SKpos) in
     (* Transmit the public key on channel ch *)
  out(ch, PKpos);

     (* Create the secret key of Registry *)
  new SKreg: skey;
     (* Create the public key of Registry *)
  let PKreg = pk(SKreg) in
     (* Transmit the public key on channel ch *)
  out(ch, PKreg);
```

```
  (* Launch an unbounded number of sessions of the Point of Sale *)
  (!processPointOfSale(PKreg, SKpos, Pwd)) |
  (* Launch an unbounded number of sessions of the Registry *)
  (!processRegistry(PKpos, SKreg)) |
  (* Launch an unbounded number of sessions of the Pocket *)
  (!processPocket(PKreg, PKpos, Pwd))


(* Verification results

C:\...\proverif2.00>proverif WOM_payment.pv
RESULT Non-interference vk is true (bad not derivable).
RESULT not attacker(vk[]) is true.
RESULT secret Pwdreg is true.
RESULT secret Npos is true.
RESULT inj-event(endREGAuthparam(x_24)) ==> inj-event(beginREGAuthparam(x_24)) is true.
RESULT inj-event(endPOSAuthparam(x_25)) ==> inj-event(beginPOSAuthparam(x_25)) is true.
*)
```