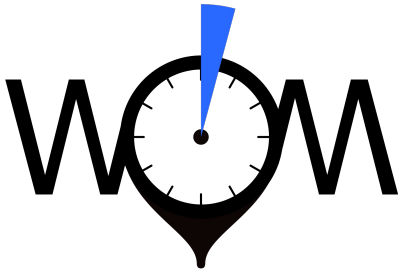
Modeling and Verification of the Worth-One-Minute Security Protocols

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

1506
UNIVERSITÀ
DEGLI STUDI
DI URBINO
CARLO BO



Objectives

1. Verification of the security architecture for a multi-agent, distributed platform

2. Experiment about the usability of tool-supported formal methods in real practice

Agenda

- Introducing the platform: Worth-One-Minute (WOM)
- Introducing the modeling and verification framework: ProVerif
- WOM verification, formally through ProVerif
- Lessons learnt

Worth-One-Minute (WOM)

- Crowdsourced systems based on volunteer contributions may require incentives and reward schemes for participants to incentivize cooperation
- "***Worth One Minute***" is a general-purpose incentives system based on the exchange of anonymous vouchers - called WOMs - each one representing a reward for *one minute* of effort towards a common (social) goal

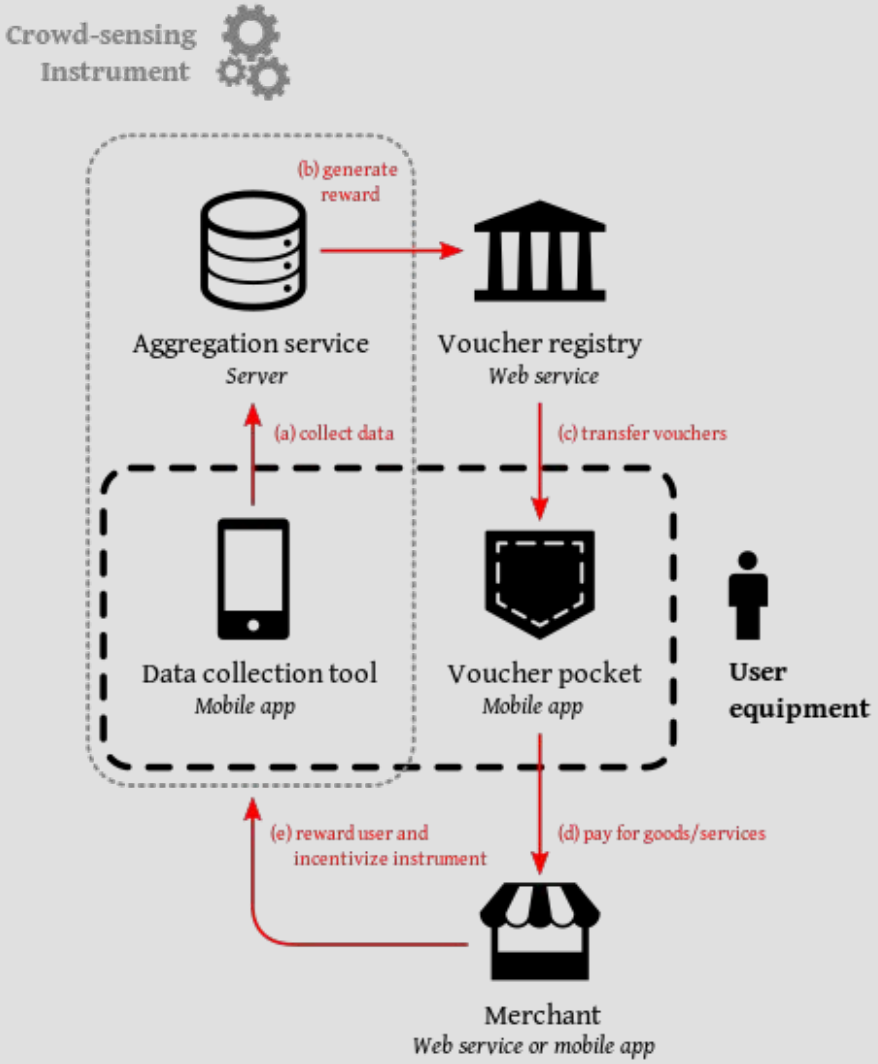
Web site: https://wom.social

WOM flow:

Volunteers participate in initiatives supported by **Instruments**, through which contributions can be rewarded in terms of vouchers, while **stakeholders** supporting the initiatives allow users to spend vouchers

Security issues:

- about the vouchers management system
- about user privacy



Proverif

Modeling functionalities

- Verbose dialect of Applied Pi Calculus
- Crypto primitives (symmetric and asymmetric encryption, digital signatures, hash functions, nonces, ...)
- Security properties (secrecy, authentication, ...)
- Unbounded number of protocol sessions

Verification

- Proof theoretic approach (from specification to Horn clauses and then resolution-based logical derivation)

Proverif

Adversary model à la Dolev-Yao

- the attacker controls the communication channel
- cryptographic operations are perfect black boxes

Proverif model structure

- Declaration of types, names and related visibility, crypto functions

Example:

```
type nonce.
type pkey.
type skey.
free ch: channel.
free voucher: bitstring [private].

fun pk(skey): pkey.

fun aencrypt(bitstring, pkey): bitstring.
reduc forall x: bitstring, y: skey;
  adecrypt(aencrypt(x, pk(y)), y) = x.

fun sencrypt(bitstring, skey): bitstring.
reduc forall x: bitstring, y: skey;
  sdecrypt(sencrypt(x, y), y) = x.
```

Proverif model structure

- Declaration of secrecy assumptions and security properties
- Secrecy: sensitive information cannot be derived by the attacker
- Authentication: if event A occurs, then event B has been observed previously (1-1 correspondence between specific events)
- Noninterference

Example:

```
not attacker(new SKctas).
not attacker(new SKinstr).
not attacker(new SKreg).
not attacker(new Pwd).

query secret Pwdreg;
        secret Nreg;
        attacker(vn).
noninterf vn.

event beginREGAuthparam(pkey).
event endREGAuthparam(pkey).

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

Proverif model structure

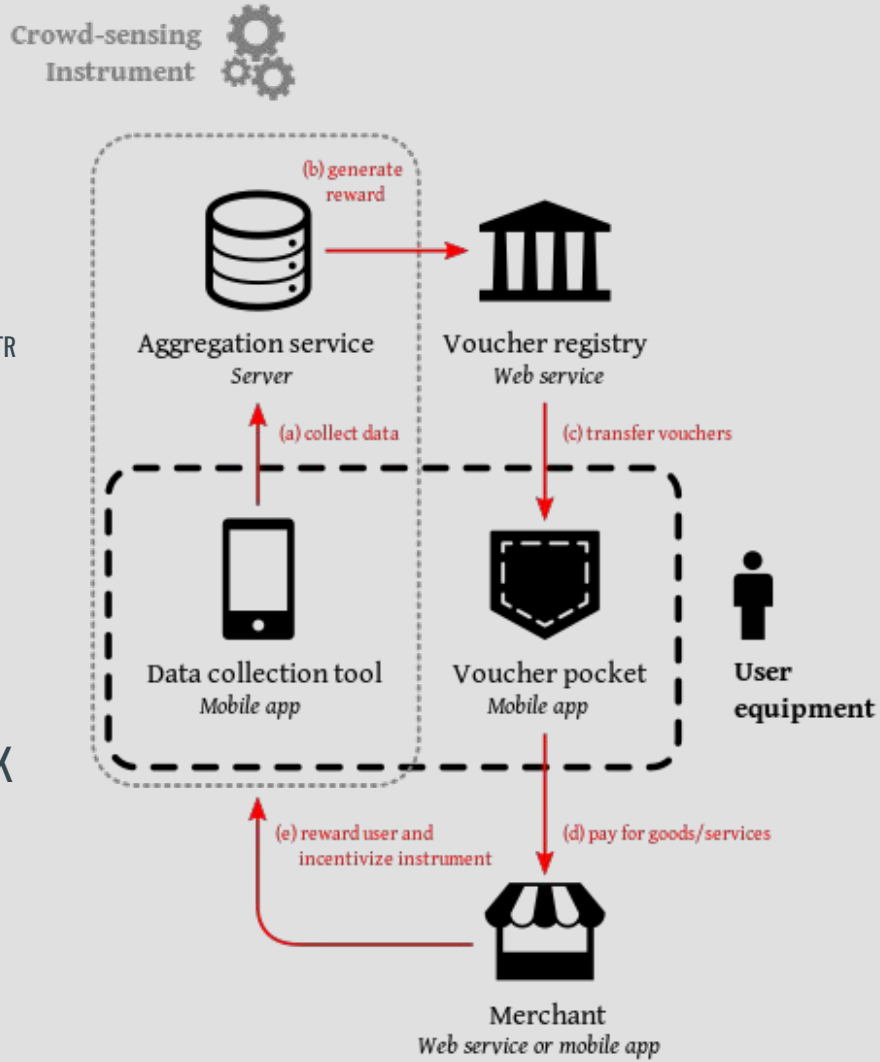
- Declaration of process macros, instances and overall architecture

Agents:

- Instrument side:
  - User Collection Tool (CT)
  - Aggregation Service (AS) with public key PK_{INSTR}
  - CT and AS share a secret key SK_{CTAS}
- Registry (REG) with public key PK_{REG}
- User Pocket (PCK)
- Merchant (POS) with public key PK_{POS}

Two crypto-protocols:

- Voucher generation - involving CT, AS, REG, PCK
- Voucher spending - involving POS, REG, PCK



Vouchers generation protocol

(a) $CT \rightarrow AS : (contributions, Pwd)SK_{CTAS}$

(b) $AS \rightarrow REG : (contributions, Pwd, N, ID_{INSTR})PK_{REG}$

(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) $REG \rightarrow PCK : (vouchers)K_s$

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

Vouchers generation protocol

(a) $CT \rightarrow AS : (contributions, Pwd)SK_{CTAS}$

(b) $AS \rightarrow REG : (contributions, Pwd, N, ID_{INSTR})PK_{REG}$

(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) $REG \rightarrow PCK : (vouchers)K_s$

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

(* **Collection Tool** *)

```
let processCollectionTool(SKctas: skey, Pwd: nonce) =
out(ch, sencrypt((cn, Pwd), SKctas));
in(ch, (message_e: bitstring));
let (OTCgen: bitstring) = sdecrypt(message_e, SKctas) in
out(ch, OTCgen).
```

Vouchers generation protocol

(a) $CT \rightarrow AS : (contributions, Pwd)SK_{CTAS}$

(b) $AS \rightarrow REG : (contributions, Pwd, N, ID_{INSTR})PK_{REG}$

(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) $REG \rightarrow PCK : (vouchers)K_s$

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

(* **Aggregation Service** *)

```
let processAggregationService(PKreg: pkey, SKinstr: skey, SKctas: skey) =

in(ch, message_a: bitstring);

let (c: bitstring, PwdX: nonce) = sdecrypt(message_a, SKctas) in

in(ch, pkX: pkey);

if pkX = PKreg then event beginREGAuthparam(pkX);

new N: nonce;

out(ch, aencrypt((c, PwdX, N, pk(SKinstr)), pkX));

...
```

Vouchers generation protocol

(a) $CT \rightarrow AS : (contributions, Pwd)SK_{CTAS}$

(b) $AS \rightarrow REG : (contributions, Pwd, N, ID_{INSTR})PK_{REG}$

(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) $REG \rightarrow PCK : (vouchers)K_s$

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

(* **Aggregation Service** *)

...

in(ch, message_c: bitstring);

let (=N, OTCgenX: nonce, =pkX) = adecrypt(message_c, SKinstr) in

out(ch, aencrypt(nonce_to_bitstring(OTCgenX), pkX));

event endASAuthparam(pk(SKinstr));

out(ch, (sencrypt(nonce_to_bitstring(OTCgenX), SKctas)));

in(ch, message_i: bitstring);

let (=N, =confirm) = adecrypt(message_i, SKinstr) in 0.

Vouchers generation protocol

(a) $CT \rightarrow AS : (contributions, Pwd)SK_{CTAS}$

(b) $AS \rightarrow REG : (contributions, Pwd, N, ID_{INSTR})PK_{REG}$

(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) $REG \rightarrow PCK : (vouchers)K_s$

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

```
(* Registry *)

let processRegistry(PKinstr: pkey, SKreg: skey) =

in(ch, message_b: bitstring);

let (c: bitstring, PwdY: nonce, NY: nonce, pkY: pkey) =
adecrypt(message_b, SKreg) in

event beginASAuthparam(pkY);

new OTCgen: nonce;

out(ch, aencrypt((NY, OTCgen, pk(SKreg)), pkY));

in(ch, message_d: bitstring);

if nonce_to_bitstring(OTCgen) = adecrypt(message_d,
SKreg) then

if pkY = PKinstr then

event endREGAuthparam(pk(SKreg));
```

Vouchers generation protocol

(a) $CT \rightarrow AS : (contributions, Pwd)SK_{CTAS}$

(b) $AS \rightarrow REG : (contributions, Pwd, N, ID_{INSTR})PK_{REG}$

(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) $REG \rightarrow PCK : (vouchers)K_s$

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

(* **Registry** *)

...

```
in(ch, (message_g: bitstring));

let (=OTCgen, =PwdY, KsY: skey) = adecrypt(message_g,
SKreg) in

out(ch, (sencrypt((vn), KsY)));

out(ch, (aencrypt((NY, confirm), PKinstr))).
```

Vouchers generation protocol

(a) $CT \rightarrow AS : (contributions, Pwd)SK_{CTAS}$

(b) $AS \rightarrow REG : (contributions, Pwd, N, ID_{INSTR})PK_{REG}$

(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) $REG \rightarrow PCK : (vouchers)K_s$

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

```
(* Pocket *)
let processPocket(PKreg: pkey, Pwd: nonce) =
in(ch, (OTCgenZ: nonce));
new Ks : skey;
out(ch, (aencrypt((OTCgenZ, Pwd, Ks), PKreg)));
in(ch, (message_h: bitstring));
let (=vn) = sdecrypt(message_h, Ks) in 0.
```

Vouchers generation protocol

(a) $CT \rightarrow AS : (contributions, Pwd)SK_{CTAS}$

(b) $AS \rightarrow REG : (contributions, Pwd, N, ID_{INSTR})PK_{REG}$

(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) $REG \rightarrow PCK : (vouchers)K_s$

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

```
process
new SKctas: skey;
new Pwd: nonce;
new SKinstr: skey;
let PKinstr = pk(SKinstr) in out(ch, PKinstr);
new SKreg: skey;
let PKreg = pk(SKreg) in out(ch, PKreg);
((!processCollectionTool(SKctas, Pwd)) |
(!processAggregationService(PKreg, SKinstr, SKctas)) |
(!processRegistry(PKinstr, SKreg)) |
(!processPocket(PKreg, Pwd)))
```

Verification of the protocol: an attack

(a) $CT \rightarrow AS : (\text{contributions}, Pwd)SK_{CTAS}$

...

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

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

...

(* **Collection Tool** *)

```
let processCollectionTool(SKctas: skey, Pwd: nonce) =
out(ch, sencrypt((cn, Pwd), SKctas));
in(ch, (message_e: bitstring));
let (OTCgen: bitstring) = sdecrypt(message_e, SKctas) in
out(ch, OTCgen).
```

goal reachable: attacker(Pwd[])

1. The attacker has some term @sid_3951.
attacker(@sid_3951).

2. The attacker has some term @sid_3949.
attacker(@sid_3949).

3. By 2, the attacker may have the session identifier @sid_3949.
So the message sencrypt((cn[],Pwd[]),SKctas[]) may be sent to the attacker at output {10}.
attacker(sencrypt((cn[],Pwd[]),SKctas[])).

4. By 1, the attacker may have the session identifier @sid_3951.
The message sencrypt((cn[],Pwd[]),SKctas[]) that the attacker may have by 3 may be received at input {11}. So the message (cn[],Pwd[]) may be sent to the attacker at output {13}.
attacker((cn[],Pwd[])).

5. By 4, the attacker may know (cn[],Pwd[]).
Using the function 2-proj-2-tuple the attacker may obtain Pwd[].
attacker(Pwd[]).

Verification of the protocol: a solution

(a) $CT \rightarrow AS : (\text{contributions}, Pwd)SK_{CTAS}$

...

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

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

...

(* **Collection Tool** *)

```
let processCollectionTool(SKctas: skey, Pwd: nonce) =
out(ch, sencrypt((cn, Pwd), SKctas));
in(ch, (message_e: bitstring));
let (OTCgen: bitstring) = sdecrypt(message_e, SKctas) in
if OTCgen <> (cn, Pwd) then out(ch, OTCgen).
```

RESULT Non-interference vn is true (bad not derivable).

RESULT secret Pwdreg is true.

RESULT secret Nreg is true.

RESULT not attacker(vn[]) is true.

RESULT inj-event(endREGAuthparam(x_30)) ==> inj-event(beginREGAuthparam(x_30)) is true.

RESULT inj-event(endASAuthparam(x_31)) ==> inj-event(beginASAuthparam(x_31)) is true.

Vouchers spending protocol

(j) POS → REG : $(f, k, ACK_{pck}, Pwd, N, ID_{pos})PK_{REG}$

(k) REG → POS : $(N, OTC_{pay}, ID_{reg})PK_{POS}$

(l) POS → REG : $(OTC_{pay})PK_{REG}$

(m) POS → PCK : (OTC_{pay})

(n) PCK → REG : $(OTC_{pay}, Pwd, K_{s1})PK_{REG}$

(o) REG → PCK : $(f, k, ID_{pos})\ K_{s1}$

(p) PCK → REG : $(OTC_{pay}, Pwd, vouchers, h(S), K_{s2})PK_{REG}$

(q) REG → POS : $(N)PK_{POS}$

(r) REG → PCK : $(ACK_{pck})K_{s2}$

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.

Conclusions

- Goals achieved
  - Proof of **security** of a real-world case study: DONE!
  - Proof of **usability** for formal methods tools like ProVerif:
    - ➢ **Modeler**: BSc grad student with some background on concurrency theory, logic programming, formal verification, but never exposed to ProVerif
    - ➢ **Training time**: 4 weeks to learn how to properly use ProVerif, 1 week to study and understand the WOM platform, 2 weeks to specify and check the two crypto-protocols
- Lessons learnt:
  - Tool-supported **formal verification** can become **common practice** in non-academic environments - it is only a matter of education
  - Software designers (including verification experts) and software developers must **work closely**