Analysis of an Electronic Voting Protocol in the Applied Pi Calculus

Steve Kremer[1] and Mark Ryan[2]

[1] Laboratoire Spécification et Vérification
CNRS UMR 8643 & INRIA Futurs projet SECSI & ENS Cachan, France
kremer@lsv.ens-cachan.fr
[2] School of Computer Science
University of Birmingham, UK
M.D.Ryan@cs.bham.ac.uk

Abstract. Electronic voting promises the possibility of a convenient, efficient and secure facility for recording and tallying votes in an election. Recently highlighted inadequacies of implemented systems have demonstrated the importance of formally verifying the underlying voting protocols. The applied pi calculus is a formalism for modelling such protocols, and allows us to verify properties by using automatic tools, and to rely on manual proof techniques for cases that automatic tools are unable to handle. We model a known protocol for elections known as FOO 92 in the applied pi calculus, and we formalise three of its expected properties, namely fairness, eligibility, and privacy. We use the ProVerif tool to prove that the first two properties are satisfied. In the case of the third property, ProVerif is unable to prove it directly, because its ability to prove observational equivalence between processes is not complete. We provide a manual proof of the required equivalence.

1 Introduction

Electronic voting promises the possibility of a convenient, efficient and secure facility for recording and tallying votes. It can be used for a variety of types of elections, from small committees or on-line communities through to full-scale national elections. However, the electronic voting machines used in recent US elections have been fraught with problems. Recent work [18] has analysed the source code of the machines sold by the second largest and fastest-growing vendor, which are in use in 37 US states. This analysis has produced a catalogue of vulnerabilities and possible attacks.

A potentially much more secure system could be implemented, based on formal protocols that specify the messages sent between the voters and administrators. Such protocols have been studied for several decades. They offer the possibility of abstract analysis of the protocol against formally-stated properties. There are two main kinds of protocol proposed for electronic voting. In blind signature schemes [10, 15, 17], the voter first obtains a token, which is a message blindly signed by the administrator and known only to the voter herself. She later sends her vote anonymously, with this token as proof of eligibility. In schemes using homomorphic encryption [4, 16], the voter cooperates with the administrator in order to construct an encryption of her vote. The admin-

istrator then exploits homomorphic properties of the encryption algorithm to compute the encrypted tally directly from the encrypted votes.

Among the properties which electronic voting protocols may satisfy are the following:

Fairness: no early results can be obtained which could influence the remaining voters.
Eligibility: only legitimate voters can vote, and only once.
Privacy: the fact that a particular voted in a particular way is not revealed to anyone.
Individual verifiability: a voter can verify that her vote was really counted.
Universal verifiability: the published outcome really is the sum of all the votes.
Receipt-freeness: a voter cannot prove that she voted in a certain way (this is important to protect voters from coercion).

In this paper, we study a protocol commonly known as the FOO 92 scheme [15], which works with blind signatures. By informal analysis (e.g., [21]), it has been concluded that FOO 92 satisfies the first four properties in the list above.

Because security protocols are notoriously difficult to design and analyse, formal verification techniques are particularly important. In several cases, protocols which were thought to be correct for several years have, by means of formal verification techniques, been discovered to have major flaws [19, 7]. Our aim in this paper is to use verification techniques to analyse the FOO 92 protocol. We model it in the applied pi calculus [3], which has the advantages of being based on well-understood concepts. The applied pi calculus has a family of proof techniques which we can use, is supported by the ProVerif tool [5], and has been used to analyse a variety of security protocols [1, 14].

Formalising the properties of electronic voting protocols mentioned above can be subtle. For example, as stated above, the privacy property as stated above is in general false, because if all the voters vote unanimously then everyone will get to know how everyone else voted. In our formalisation, we model two voters, and consider whether the attacker could detect if they swap their votes.

Part of our verification is performed using the ProVerif tool [5]. One kind of property which ProVerif can check is *reachability*. This is used for modelling the eligibility property, and also part of the fairness property. Another kind of check which can be made with ProVerif is *observational equivalence* between processes [6]. We use this to model another part of fairness, and also for the privacy property. ProVerif's ability to check observational equivalence is not complete: there may be cases in which processes are observationally equivalent but ProVerif cannot prove it. This is indeed the case for the equivalence required for checking privacy, and we provide a manual proof.

The paper is structured as follows. The voting protocol FOO 92 is described in the next section, and we review the applied pi calculus in section 3. In section 4, we model FOO 92 in the applied pi calculus, and analyse it in section 5. Our conclusions are in section 6.

2 The FOO 92 protocol

In this section we give an informal description of the FOO 92 voting protocol [15]. In section 4 we show how this protocol can be modeled in the applied pi calculus.

The protocol involves voters, an administrator, verifying that only eligible voters can cast votes, and a collector, collecting and publishing the votes. In comparison with authentication protocols, the protocol also uses some unusual cryptographic primitives, such as secure bit-commitment and blind signatures. Moreover, it relies on anonymous channels. We deliberately do not specify the way these channels are handled as any anonymizer mechanism could be suitable depending on the precise context the protocol is used in. One possible implementation is to use MIX-nets introduced by Chaum in [8] or one of its successors e.g. onion routing [22].

In a first phase, the voter gets a signature on a commitment to his vote from the administrator. To ensure privacy, blind signatures [9] are used, i.e. the administrator does not learn the commitment of the vote.

- Voter V selects a vote v and computes the commitment $x = \xi(v,r)$ using the commitment scheme ξ and a random key r;
- V computes the message $e = \chi(x, b)$ using a blinding function χ and a random blinding factor b;
- V digitally signs e and sends his signature $\sigma_V(e)$ to the administrator A together with his identity;
- A verifies that V has the right to vote, has not voted yet and that the signature is valid; if all these tests hold, A digitally signs e and sends his signature $\sigma_A(e)$ to V;
- V now *unblinds* $\sigma_A(e)$ and obtains $y = \sigma_A(x)$, i.e. a signed commitment to V's vote.

The second phase of the protocol is the actual voting phase.

- V sends y, A's signature on the commitment to V's vote, to the collector C using an anonymous channel;
- C checks correctness of the signature y and, if the test succeeds, enters (ℓ, x, y) onto a list as an ℓ-th item.

The last phase of the voting protocol starts, once the collector decides that he received all votes, e. g. after a fixed deadline. In this phase the voters reveal the random key r which allows C to open the votes and publish them.

- C publishes the list (ℓ_i, x_i, y_i) of commitments he obtained;
- V verifies that his commitment is in the list and sends ℓ, r to C via an anonymous channel;
- C opens the ℓ-th ballot using the random r and publishes the vote v.

Note that we need to separate the voting phase into a commitment phase and an opening phase to avoid releasing partial results of the election. The protocol is summarized in Fig. 1.

3 The applied pi calculus

The applied pi calculus [3] is a language for describing concurrent processes and their interactions. It is based on the pi calculus, but is intended to be less pure and therefore

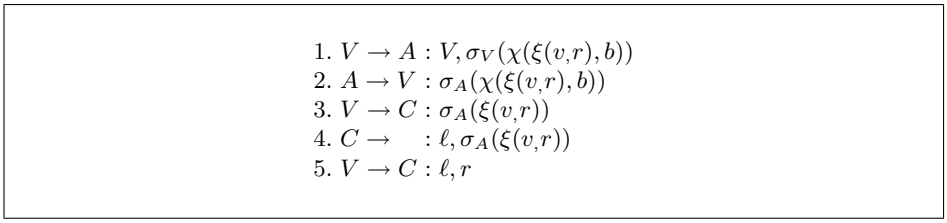

Fig. 1: FOO 92 protocol

more convenient to use. Properties of processes described in the applied pi calculus can be proved by employing manual techniques [3], or by automated tools such as ProVerif [5]. As well as reachability properties which are typical of model checking tools, ProVerif can in some cases prove that processes are observationally equivalent [6]. This capability is important for privacy-type properties such as those we study here. The applied pi calculus has been used to study a variety of security protocols, such as those for private authentication [14] and for fast key establishment [1].

To describe processes in the applied pi calculus, one starts with a set of *names* (which are used to name communication channels or other constants), a set of *variables*, and a *signature* Σ which consists of the function symbols which will be used to define terms. In the case of security protocols, typical function symbols will include enc for encryption, which takes plaintext and a key and returns the corresponding cipher text, and dec for decryption, taking cipher text and a key and returning the plaintext. One can also describe the equations which hold on terms constructed from the signature, such as

$$\mathsf{dec}(\mathsf{enc}(x, k), k) = x.$$

Terms are defined as names, variables, and function symbols applied to other terms. Terms and function symbols are sorted, and of course function symbol application must respect sorts and arities. Modelling bit commitment and blind signatures may also be done by choosing function symbols and defining appropriate equations.

In the applied pi calculus, one has (plain) processes and extended processes. Plain processes are built up in a similar way to processes in the pi calculus, except that messages can contain terms (rather than just names).

$P, Q, R :=$	plain processes
0	null process
$P \mid Q$	parallel composition
$!P$	replication
$\nu n.P$	name restriction (“new”)
if $M = N$ then P else Q	conditional
$u(x).P$	message input
$\bar{u}\langle N\rangle.P$	message output

Extended processes can also be *active substitutions*:

$A, B, C :=$	extended processes

P	plain process
$A \mid B$	parallel composition
$\nu n.A$	name restriction
$\nu x.A$	variable restriction
$\{^M/_x\}$	active substitution

$\{^M/_x\}$ is the substitution that replaces the variable x with the term M. Active substitutions generalise "let". The process $\nu x.(\{^M/_x\} \mid P)$ corresponds exactly to "let $x = M$ in P".

Active substitutions are useful because they allow us to map an extended process A to its *frame* $\phi(A)$ by replacing every plain processes in A with 0. A frame is an extended process built up from 0 and active substitutions by parallel composition and restriction. The frame $\phi(A)$ can be viewed as an approximation of A that accounts for the static knowledge A exposes to its environment, but not A's dynamic behaviour.

The operational semantics of processes in the applied pi calculus is defined by structural rules defining two relations: *structural equivalence*, noted $\equiv$, and *internal reduction*, noted $\rightarrow$. A context $C[\cdot]$ is a process with a hole; an evaluation context is a context whose hole is not under a replication, a conditional, an input, or an output. Structural equivalence is is the smallest equivalence relation on extended processes that is closed under α-conversion on names and variables, by application of evaluation contexts, and satisfying some further basic structural rules such as $A \mid 0 \equiv A$, associativity and commutativity of $|$, binding-operator-like behaviour of ν, and when $\Sigma \vdash M = N$ the equivalences:

$$\nu x.\{^M/_x\} \equiv 0 \qquad \{^M/_x\} \mid A \equiv \{^M/_x\} \mid A\{^M/_x\} \qquad \{^M/_x\} \equiv \{^N/_x\}$$

Internal reduction $\rightarrow$ is the smallest relation on extended processes closed under structural equivalence such that $\bar{a}\langle x\rangle.P \mid a(x).Q \rightarrow P \mid Q$ and whenever $\Sigma \nvdash M = N$,

$$\text{if } M = M \text{ then } P \text{ else } Q \rightarrow P \qquad \text{if } M = N \text{ then } P \text{ else } Q \rightarrow Q.$$

Many properties of security protocols (including some of the properties we study in this paper) are formalised in terms of *observational equivalence* between processes. To define this, we write $A \Downarrow a$ when A can send a message on a, that is, when $A \rightarrow^* C[\bar{a}\langle M\rangle.P]$ for some evaluation context C that does not bind a.

Definition 1. *Observational equivalence ($\approx$) is the largest symmetric relation R between closed extended processes with the same domain such that $A\ R\ B$ implies:*

1. *if $A \Downarrow a$ then $B \Downarrow a$.*
2. *if $A \rightarrow^* A'$ then $B \rightarrow^* B'$ and $A'\ R\ B'$ for some B'.*
3. *$C[A]\ R\ C[B]$ for closing evaluation contexts C.*

In cases in which the two processes differ only by the terms they contain, if they are also observationally equivalent then ProVerif may be able to prove it directly. However, ProVerif's ability to prove observational equivalence is incomplete, and therefore

sometimes one has to resort to manual methods, whose justifications are contained in [3].

The method we use in this paper relies on two further notions: *static equivalence* ($\approx_s$), and *labelled bisimilarity* ($\approx_l$). Static equivalence just compares the static knowledge processes expose to their environment. Two frames are statically equivalent if, when considered as substitutions, they agree on the distinguishability of terms. For frames, static equivalence agrees with observational equivalence, while for general extended processes, observational equivalence is finer.

The definition of *labelled* bisimilarity is like the usual definition of bisimilarity, except that at each step in the unravelled definition one additionally requires that the processes are statically equivalent. Labelled bisimilarity and observational equivalence coincide [3]. Therefore, to prove observational equivalence, it is sufficient to prove bisimilarity and static equivalence at each step. This is what we do to prove the privacy property.

4 Modelling FOO 92 in the applied pi calculus

4.1 Model

We use the applied pi calculus to model the FOO 92 protocol. The advantage is that we can combine powerful (hand) proof techniques from the applied pi calculus with automated proofs provided by Blanchet's ProVerif tool. Moreover, the verification is not restricted to a bounded number of sessions and we do not need to explicitly define the adversary. We only give the equational theory describing the intruder theory. Generally, the intruder has access to any message sent on a public, i.e. unrestricted, channel. These public channels model the network. Note that all channels are anonymous in the applied pi calculus. Unless the identity or something like the IP address is specified explicitly in the conveyed message, the origin of a message is unknown. This abstraction of a real network is very appealing, as it avoids having us to model explicitly an anonymiser service. However, we stress that a real implementation needs to treat anonymous channels with care.

Most of our proofs rely directly on Blanchet's ProVerif tool. The input for the tool is given in an ascii version of the applied pi calculus. To be as precise as possible, the processes described below are directly extracted out of the input files and are given in a pretty-printed version of the ascii input. The minor changes with the usual applied pi calculus notation should be clear.

4.2 Signature and equational theory

The signature and equational theory are represented in Process 1. We model cryptography in a Dolev-Yao style as being perfect. In this model we can note that bit commitment (modeled by the functions commit and open) is identical to classical symmetric-key encryption. The functions and equations that handle public keys and hostnames should be clear. Digital signatures are modeled as being signatures with message recovery, i.e. the signature itself contains the signed message which can be extracted using the checksign function. To model blind signatures we add a pair of functions blind and unblind.

```
(* Signature *)
fun commit/2    (* bit commitment *)
fun open/2      (* open bit commitment *)
fun sign/2      (* digital signature *)
fun checksign/2 (* open digital signature *)
fun pk/1        (* get public key from private key *)
fun host/1      (* get host from public key *)
fun getpk/1     (* get public key from host *)
fun blind/2     (* blinding *)
fun unblind/2   (* undo blinding *)

(* Equational theory *)
equation open(commit(m,r),r) = m
equation getpk(host(pubkey))=pubkey
equation checksign(sign(m,sk),pk(sk)) = m
equation unblind(blind(m,r),r) = m
equation unblind(sign(blind(m,r),sk),r) = sign(m,sk)
```

Process 1. signature and equational theory

These functions are again similar to perfect symmetric key encryption and bit commitment. However, we add a second equation which permits us to extract a signature out of a blinded signature, when the blinding factor is known. The ProVerif tool also implicitly handles pairing: pair(x,y) is abbreviated as (x,y). We also consider the functions fst and snd to extract the first, respectively second element of a pair. Note that because of the property that unblind(sign(blind(m,r),sk),r) = sign(unblind(blind(m,r),r),sk) = sign(m,sk), our theory is not a subterm theory. Therefore the results for deciding static equivalence from [2] do not apply. However, an extension of [2] presents new results that seem to cover a more general family of theories, including the one considered here [12].

4.3 The environment process

The main process is specified in Process 2. Here we model the environment and specify how the other processes (detailed below) are combined. First, fresh secret keys for the voters and the administrator are generated using the restriction operator. For simplicity, all legitimate voters share the same secret key in our model (and therefore the same public key). The public keys and hostnames corresponding to the secret keys are then sent on a public channels, i.e. they are made available to the intruder. The list of legitimate voters is modeled by sending the public key of the voters to the administrator on a private communication channel. We also register the intruder as being a legitimate voter by sending his public key pk(ski) where ski is a free variable: this enables the intruder to introduce votes of his choice and models that some voters may be corrupted. Then we combine an unbounded number of each of the processes (voter, administrator and collector). An unbounded number of administrators and collectors models that

```
process
    ν ska. ν skv. (* private keys *)
    ν privCh. (* channel for registering legimitate voters *)
    let pka=pk(ska) in
    let hosta = host(pka) in
    let pkv=pk(skv) in
    let hostv=host(pkv) in
    (* publish host names and public keys *)
    out(ch,pka). out(ch,hosta).
    out(ch,pkv). out(ch,hostv).
    (* register legimitate voters *)
    ((out(privCh,pkv). out(privCh,pk(ski))) |
    (!processV)|(!processA)|(!processC))
```

Process 2. environment process

```
let processV =
    ν blinder. ν r.
    let blindedcommitedvote=blind(commit(v,r),blinder) in
    out(ch,(hostv,sign(blindedcommitedvote,skv))).
    in(ch,m2).
    let blindedcommitedvote0=checksign(m2,pka) in
    if blindedcommitedvote0=blindedcommitedvote then
    let signedcommitedvote=unblind(m2,blinder) in
    phase 1.
    out(ch,signedcommitedvote).
    in(ch,(l,=signedcommitedvote)).
    phase 2.
    out(ch,(l,r))
```

Process 3. voter process

these processes are servers, creating a separate instance of the server process (e.g. by "forking") for each client.

4.4 The voter process

The voter process given in Process 3 models the role of a voter. At the beginning two fresh random numbers are generated for blinding, respectively bit commitment of the vote. Note that the vote is not modeled as a fresh nonce. This is because generally the domain of values of the votes are known. For instance this domain could be $\{yes, no\}$, a finite number of candidates, etc. Hence, vulnerability to guessing attacks is an important topic. We will discuss this issue in more detail in section 5. The remainder of the specification follows directly the informal description given in section 2. The command in(ch,(l,=s)) means the process inputs not any pair but a pair whose second argument is s. Note that we use phase separation commands, introduced by the ProVerif tool as

```
let processA =
    in(privCh,pubkv). (* register legimitate voters *)
    in(ch,m1).
    let (hv,sig)=m1 in
    let pubkeyv=getpk(hv) in
    if pubkeyv = pubkv then
    out(ch,sign(checksign(sig,pubkeyv),ska))
```

Process 4. administrator process

```
let processC =
    phase 1.
    in(ch,m3).
    ν l. out(ch,(l,m3)).
    phase 2.
    in (ch,(=l,rand)).
    let voteV=open(checksign(m3,pka),rand) in
    out(ch,voteV)
```

Process 5. collector process

global synchronization commands. The process first executes all instructions of a given phase before moving to the next phase. The separation of the protocol in phases is useful when analyzing fairness and the synchronization is even crucial for privacy to hold.

4.5 The administrator process

The administrator is modeled by the process represented in Process 4. In order to verify that a voter is a legitimate voter, the administrator first receives a public key on a private channel. Legitimate voters have been registered on this private channel in the environment process described above. The received public key has to match the voter who is trying to get a signed ballot from the administrator. If the public key indeed matches, then the administrator signs the received message which he supposes to be a blinded ballot.

4.6 The collector process

In Process 5 we model the collector. When the collector receives a committed vote, he associates a fresh label 'l' with this vote. Publishing the list of votes and labels is modeled by sending those values on a public channel. Then the voter can send back the random number which served as a key in the commitment scheme together with the label. The collector receives the key matching the label and opens the vote which he then publishes. Note that in this model the collector immediately publishes the vote without waiting that all voters have committed to their vote. In order to verify in section 5 that no early votes can be revealed we simply omit the last steps in the voter and collector process corresponding to the opening and publishing of the results.

5 Analysis

We have analysed three major properties of electronic voting protocols: fairness, eligibility and privacy. Most of the properties can be directly verified using ProVerif. The tool allows us to verify standard secrecy properties as well as resistance against guessing attacks, defined in terms of equivalences. For all but one property, privacy, the tool directly succeeds its proofs. When analysing privacy, we need to rely on the proof techniques introduced in [3]. Although the results are positive results, we believe that the way we verify the properties increases the understanding of the properties themselves and also the way to model them.

5.1 Fairness

Fairness is the property that ensures that no early results can be obtained and influence the vote. Of course, when we state that no early results can be obtained, we mean that the protocol does not leak any votes before the opening phase. It is impossible to prevent "exit polls", i.e. people revealing their vote when asked.

We model fairness as a secrecy property: it should be impossible for an attacker to learn a vote before the opening phase, i.e. before the beginning of phase 2.

Standard secrecy. Checking *standard* secrecy, i.e. secrecy based on reachability, is the most basic property ProVerif can check. We request ProVerif to check that the private free variable v representing the vote cannot be deduced by the attacker. ProVerif directly succeeds to prove this result.

Resistance against guessing attacks. In the previous paragraph we deduce that a *standard* attacker cannot learn a legitimate voter's vote. However, voting protocols are particularly vulnerable to *guessing attacks* because the values of the votes are taken from a small domain of possible values. Intuitively, in a guessing attack, an attacker *guesses* a possible value for the secret vote and then tries to verify his guess. A trivial example of a guessing attack is when the voter encrypts his vote with the collector's public key (using deterministic encryption). Then the attacker just needs to encrypt his guess and compare the result with the observed encrypted vote. Guessing attacks have been formalized by Lowe [20] and later by Delaune and Jacquemard [13]. A definition in terms of equivalences has been proposed by Corin et al. in [11]:

Definition 2. *Let ϕ be a frame in which v is free. Then we say that ϕ verifies a guess of v if $\phi \not\approx_s \nu v.\phi$. Conversely, we say that ϕ is secure wrt v if $\phi \approx_s \nu v.\phi$.*

Intuitively, if ϕ and $\nu v.\phi$ can be distinguished then an adversary can verify his guess using ϕ. This is also the definition checked by ProVerif. ProVerif succeeds in proving this stronger version of secrecy for the commitment phase of the FOO 92 protocol. Note that verification of guessing attacks does not support considering the protocol up to a given phase. Therefore, we slightly change the processes presented in section 4: we omit the last sending of the voter process which allows the opening of the commitment.

```
let processC =
    phase 1.
    in(ch,m3).
    ν l. out(ch,(l,m3)).
    phase 2.
    in (ch,(=l,rand)).
    let voteV=open(checksign(m3,pka),rand) in
    ν attack.
        if voteV=challengeVote then
                out(ch, attack)
        else
                out(ch, voteV)
```

Process 6. modified collector process for checking the eligibility properties

Strong secrecy. We also verified *strong secrecy* in the sense of [6]. Intuitively, strong secrecy is verified if the intruder cannot distinguish between two processes where the secret changes. For the precise definition, we refer the reader to [6]. The main difference with guessing attacks is that strong secrecy relies on observational equivalence rather than static equivalence. ProVerif directly succeeds to prove strong secrecy.

Corrupt administrator. We have also verified standard secrecy, resistance against guessing attacks and strong secrecy in the presence of a corrupt administrator. A corrupt administrator is modeled by outputting the administrator's secret key on a public channel. Hence, the intruder can perform any actions the administrator could have done. Again, the result is positive: the administrator cannot learn the votes of a honest voter, before the committed votes are opened. Note that we do not need to model a corrupt collector, as the collector never uses his secret key, i.e. the collector could anyway be replaced by the attacker.

5.2 Eligibility

Eligibility is the property verifying that only legitimate voters can vote, and only once. The way we verify the first part of this property is by giving the attacker a *challenge vote*. We modify the processes in two ways: (i) the attacker is not registered as a legitimate voter; (ii) the collector tests whether the received vote is the challenge vote and outputs the restricted name attack if the test succeeds. The modified collector process is given in Process 6. Verifying eligibility is now reduced to secrecy of the name attack. ProVerif succeeds in proving that attack cannot be deduced by the attacker.

If we register the attacker as a legitimate voter, the tool finds the trivial attack, where the intruder votes *challenge vote*. Similarly, if a corrupt administrator is modeled then the intruder can generate a signed commitment to the challenge vote and insert it.

The second part of the eligibility property (that a voter can vote only once) cannot be verified in our model, because of our simplifying assumption that all voters share the same key.

```
process
    let x=choice[v1,v2] in
    let y=choice[v2,v1] in
    ( (out(ch,x)) | (out(ch,y)) )
```

Process 7. limitation of the ProVerif tool to prove observational equivalence

5.3 Privacy

The privacy property aims to guarantee that the link between a given voter V and his vote v remains hidden. Anonymity and privacy properties have been successfully studied using equivalences. However, the definition of privacy in the context of voting protocols is rather subtle. While generally most security properties should hold against an arbitrary number of dishonest participants, arbitrary coalitions do not make sense here. Consider for instance the case where all but one voter are dishonest: as the results of the vote are published at the end, the dishonest voter can collude and determine the vote of the honest voter. A classical trick for modeling anonymity is to ask whether two processes, one in which V_1 votes and one in which V_2 votes, are equivalent. However, such an equivalence does not hold here as the voters' identities are revealed (and they need to be revealed at least to the administrator to verify eligibility). In a similar way, an equivalence of two processes where only the vote is changed does not hold, because the votes are published at the end of the protocol. To ensure privacy we need to hide the *link* between the voter and the vote and not the voter or the vote itself.

In order to give a reasonable definition of privacy, we need to suppose that at least two voters are honest. We denote the voters V_1 and V_2 and their votes $vote_1$, respectively $vote_2$. We say that a voting protocol respects privacy whenever a process where V_1 votes $vote_1$ and V_2 votes $vote_2$ is observationally equivalent to a process where V_1 votes $vote_2$ and V_2 votes $vote_1$.

With respect to the modeling given in section 4 we explicitly add a second voter. However, the equivalence that is checked by ProVerif is strictly finer than observational equivalence. Therefore the tool does not succeed in proving the above given privacy property. In Process 7, we illustrate a simple process that is observationally equivalent (it is actually structurally equivalent), but cannot be proven so by ProVerif. This example also illustrates ProVerif's choice operator used to define two processes that should be proven observationally equivalent. The choice operator is a binary operator that defines two processes P_1 and P_2 such that choice(x_1,x_2) evaluates to x_1 in P_1 and to x_2 in P_2. Although the two processes are structurally equivalent, the current version of ProVerif does not succeed in proving observational equivalence.

As ProVerif takes as input processes in the applied pi calculus, we can rely on hand proof techniques to show privacy. The processes modeling the two voters are shown in Process 8. The main process is adapted accordingly to publish public keys and host names.

Proposition 1. *The FOO 92 protocol respects privacy, i.e.* $P[vote_1/v_1, vote_2/v_2] \approx P[vote_2/v_1, vote_1/v_2]$*, where* P *is given in Process 9.*

```
(* Voter1 *)
let processV1 =
    ν blinder1. ν r1.
    let blindedcommitedvote1=blind(commit(v1,r1),blinder1) in
    out(ch,(hostv1,sign(blindedcommitedvote1,skv1))).
    in(ch,m21).
    let blindedcommitedvote01=checksign(m21,pka) in
    if blindedcommitedvote01=blindedcommitedvote1 then
    let signedcommitedvote1=unblind(m21,blinder1) in
    phase 1.
    out(ch,signedcommitedvote1).
    in(ch,(l1,=signedcommitedvote1)).
    phase 2.
    out(ch,(l1,r1))

(* Voter2 *)
let processV2 =
    ν blinder2. ν r2.
    let blindedcommitedvote2=blind(commit(v2,r2),blinder2) in
    out(ch,(hostv2,sign(blindedcommitedvote2,skv2))).
    in(ch,m22).
    let blindedcommitedvote02=checksign(m22,pka) in
    if blindedcommitedvote02=blindedcommitedvote2 then
    let signedcommitedvote2=unblind(m22,blinder2) in
    phase 1.
    out(ch,signedcommitedvote2).
    in(ch,(l2,=signedcommitedvote2)).
    phase 2.
    out(ch,(l2,r2))
```

Process 8. two voters for checking the privacy property

The proof can be sketched as follows. First note that the only difference between $P[vote_1/v_1, vote_2/v_2]$ and $P[vote_2/v_1, vote_1/v_2]$ lies in the two voter processes. We therefore first show that

$$(processV1|processV2)[vote_1/v_1, vote_2/v_2] \approx (processV1|processV2)[vote_2/v_1, vote_1/v_2].$$

To prove this we show labelled bisimilarity. We denote the left-hand process as P_1 and the right-hand process as P_2. The labelled transition of P_1

$$\begin{aligned} P_1 \xrightarrow{\nu x1.\bar{ch}\langle x1\rangle} & \nu blinder1.\nu r1.\nu blinder2.\nu r2. \\ & (P_1'|\{^{(hostv1,sign(blind(commit(v1,r1),blinder1),skv1)}/_{x1}\}) \\ \xrightarrow{\nu x2.\bar{ch}\langle x2\rangle} & \nu blinder1.\nu r1.\nu blinder2.\nu r2. \\ & (P_1''|\{^{(hostv1,sign(blind(commit(v1,r1),blinder1),skv1)}/_{x1}\} \\ & |\{^{(hostv2,sign(blind(commit(v2,r2),blinder1),skv2)}/_{x2}\}) \end{aligned}$$

```
process
    ν ska. ν skv1. ν skv2. (* private keys *)
    ν privCh. (* channel for registrating legimitate voters *)
    let pka=pk(ska) in
    let hosta = host(pka) in
    let pkv1=pk(skv1) in
    let hostv1=host(pkv1) in
    let pkv2=pk(skv2) in
    let hostv2=host(pkv2) in
    (* publish host names and public keys *)
    out(ch,pka). out(ch,hosta).
    out(ch,pkv1). out(ch,hostv1).
    out(ch,pkv2). out(ch,hostv2).
    let v1=choice[vote1,vote2] in
    let v2=choice[vote2,vote1] in
    ((out(privCh,pkv1). out(privCh,pkv2). out(privCh,pk(ski))) |
    (processV1)|(processV2)|(!processA)|(!processC))
```

Process 9. main process with two voters

can be simulated by P_2 as

$$\begin{aligned} P_2 \xrightarrow{\nu x1.\bar{ch}\langle x1\rangle} & \nu blinder1.\nu r1.\nu blinder2.\nu r2. \\ & (P_2' | \{^{(hostv1, sign(blind(commit(v2,r1),blinder1),skv1)}/_{x1}\}) \\ \xrightarrow{\nu x2.\bar{ch}\langle x2\rangle} & \nu blinder1.\nu r1.\nu blinder2.\nu r2. \\ & (P_2'' | \{^{(hostv1, sign(blind(commit(v2,r1),blinder1),skv1)}/_{x1}\} \\ & | \{^{(hostv2, sign(blind(commit(v1,r2),blinder1),skv2)}/_{x2}\}) \end{aligned}$$

For the first input of both voters, we need to consider two cases: either the input of both voters corresponds to the expected messages from the administrator or any other message has been introduced by the attacker. In the first case, both voters synchronize at phase 1 and the frames of P_1, respectively P_2 are

$$\begin{aligned} \phi_1 = \nu blinder1.\nu r1.\nu blinder2.\nu r2. \\ & \{^{(hostv1, sign(blind(commit(v1,r1),blinder1),v1))}/_{x1}, \\ & \{^{(hostv2, sign(blind(commit(v2,r2),blinder2),v2))}/_{x2}, \\ & \{^{sign(blind(commit(v1,r1),blinder1),skva)}/_{x3}, \\ & \{^{sign(blind(commit(v2,r2),blinder2),skva)}/_{x4}\} \\ \phi_2 = \nu blinder1.\nu r1.\nu blinder2.\nu r2. \\ & \{^{(hostv1, sign(blind(commit(v2,r1),blinder1),v1))}/_{x1}, \\ & \{^{(hostv2, sign(blind(commit(v1,r2),blinder2),v2))}/_{x2}, \\ & \{^{sign(blind(commit(v2,r1),blinder1),skva)}/_{x3}, \\ & \{^{sign(blind(commit(v1,r2),blinder2),skva)}/_{x4}\} \end{aligned}$$

Given our equational theory and the fact that the blinding factors are restricted, these frames are statically equivalent. In the second case, if at least one input does not corre-

spond to the correct administrator's signature, both voter processes will block, as testing correctness of the message fails and hence they cannot synchronize.

After the synchronization at phase 1, the remaining of the voter processes are structurally equivalent: the remaining of the first voter's process of P_1 is equivalent to the remaining of the second voter's process of P_2 and vice-versa. Due to this structural equivalence, P_2 can always simulate P_1 (and vice-versa). Moreover static equivalence will be ensured: with respect to frames ϕ_1 and ϕ_2 no other difference will be introduced and the blinding factors are never divulged.

Given observational equivalence of the voter processes, we can conclude observational equivalence of the the whole process, as observational equivalence is closed under application of closed evaluation contexts.

Note also that the use of phases is crucial for privacy to be respected. Surprisingly, when we omit the synchronization after the registration phase with the administrator, privacy is violated. Consider the following scenario. Voter 1 contacts the administrator. As no synchronization is considered, voter 1 can send his commited vote to the collector before voter 2 contacts the administrator. As voter 2 could not have submitted the commited vote, the attacker can link this commitment to the first voter's identity. This problem was found during a first attempt to prove the protocol where the phase instructions were omitted. The original paper divides the protocol into three phases but does not explain the crucial importance of the synchronization after the first phase. Our analysis emphasizes this need and we believe that it increases the understanding of some subtle details of the privacy property in this protocol.

6 Conclusion

We have modelled the FOO 92 electronic voting scheme in the applied pi calculus, and proved three kinds of property. Each property is checked either by reachability analysis or by checking observational equivalence:

Fairness. F1: the vote of a particular voter is not leaked to an attacker (reachability). F2: a guess of a vote cannot be verified by the attacker and strong secrecy is guaranteed (observational equivalence). These properties are also proved in the presence of a corrupt administrator.

Eligibility. E1: an attacker cannot trick the system into accepting his vote (reachability).

Privacy. P1: the attacker cannot distinguish the actual situation from one in which two voters have swapped their votes (observational equivalence).

The reachability properties (F1, E1) and the first observational equivalence property (F2) can be proved by ProVerif. The other observational equivalence property (P1) is more delecate, both in the way it is formulated and in the way that it is proved. ProVerif cannot prove this observational equivalence automatically. Therefore we proved it manually, by showing that the two processes are labelled-bisimilar.

In proving P1 manually, we noticed a feature of the protocol which is not much stressed in the descriptions (e.g. [15, 21]) but is vital for the proof: every participant

must finish the registration stage before proceeding to the voting stage, and every participant must finish the voting stage before the collector can begin opening the votes. Otherwise, some attacks are possible. For example, if voting could begin before everyone has registered, the attacker could break privacy by temporarily blocking all registrations but V's. If V then votes, the attacker can establish a link between V and V's vote. We used the phase construct of ProVerif to prevent this.

Acknowledgments. Many thanks to Bruno Blanchet for suggestions about using ProVerif, as well as to Mathieu Baudet and Stéphanie Delaune for interesting discussions and comments. Thanks also to anonymous reviewers for helpful comments. Steve Kremer's work was partially supported by the RNTL project PROUVÉ and the ACI-SI Rossignol.

References

1. Martín Abadi, Bruno Blanchet, and Cedric Fournet. Just fast keying in the pi calculus. In David Schmidt, editor, *13th European Symposium on Programming (ESOP'04)*, volume 2986 of *Lecture Notes in Computer Science*, pages 340–354, Barcelona, Spain, March 2004. Springer.
2. Martín Abadi and Véronique Cortier. Deciding knowledge in security protocols under equational theories. In Josep Diaz, Juhani Karhumäki, Arto Lepistö, and Don Sannella, editors, *31st Int. Coll. Automata, Languages, and Programming (ICALP'04)*, volume 3142 of *Lecture Notes in Computer Science*, pages 46–58, Turku, Finland, July 2004. Springer.
3. Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In Hanne Riis Nielson, editor, *Proceedings of the 28th ACM Symposium on Principles of Programming Languages*, pages 104–115, London, UK, January 2001. ACM.
4. Josh G. Beneloh. *Verifiable Secret Ballot Elections*. PhD thesis, Yale University, 1987.
5. Bruno Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In Steve Schneider, editor, *14th IEEE Computer Security Foundations Workshop*, pages 82–96, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society Press.
6. Bruno Blanchet. Automatic Proof of Strong Secrecy for Security Protocols. In *IEEE Symposium on Security and Privacy*, pages 86–100, Oakland, California, May 2004.
7. Rohit Chadha, Steve Kremer, and Andre Scedrov. Formal analysis of multi-party contract signing. In Riccardo Focardi, editor, *17th IEEE Computer Security Foundations Workshop*, pages 266–279, Asilomar, CA, USA, June 2004. IEEE Computer Society Press.
8. David Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–88, February 1981.
9. David Chaum. Blind signatures for untraceable payments. In *Advances in Cryptology, Proceedings of CRYPTO'82*, pages 199–203. Plenum Press, 1983.
10. David Chaum. Elections with unconditionally-secret ballots and disruption equivalent to breaking RSA. In *Advances in Cryptology – Eurocrypt'88*, volume 330 of *LNCS*, pages 177–182. Springer, 1988.
11. Ricardo Corin, Jeroen Doumen, and Sandro Etalle. Analysing password protocol security against off-line dictionary attacks. In *2nd International Workshop on Security Issues with Petri Nets and other Computational Models (WISP'04)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2004. To appear.
12. Véronique Cortier. Personal communication, 2004.
13. Stéphanie Delaune and Florent Jacquemard. A theory of dictionary attacks and its complexity. In Riccardo Focardi, editor, *17th IEEE Computer Security Foundations Workshop*, pages 2–15, Asilomar, Pacific Grove, CA, USA, June 2004. IEEE Computer Society Press.

14. Cedric Fournet and Martin Abadi. Hiding names: Private authentication in the applied pi calculus. In *International Symposium on Software Security (ISSS'02)*, pages 317–338. Springer, 2003.
15. Atsushi Fujioka, Tatsuaki Okamoto, and Kazui Ohta. A practical secret voting scheme for large scale elections. In J. Seberry and Y. Zheng, editors, *Advances in Cryptology — AUSCRYPT '92*, volume 718 of *Lecture Notes in Computer Science*, pages 244–251. Springer, 1992.
16. Martin Hirt and Kazue Sako. Efficient receipt-free voting based on homomorphic encryption. In Bart Preneel, editor, *Advances in Cryptography – Eurocrypt'00*, volume 1807 of *Lecture Notes in Computer Science*, pages 539–556, Bruges, Belgium, may 2000. Springer.
17. Wen-Shenq Juang and Chin-Laung Lei. A secure and practical electronic voting scheme for real world environments. *IEICE Transaction on Fundamentals of Electronics, Communications and Computer Science, E80A*, 1:64–71, January 1997.
18. Tadayoshi Kohno, Adam Stubblefield, Aviel D. Rubin, and Dan S. Wallach. Analysis of an electronic voting system. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 2004.
19. Gavin Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, 56:131–133, 1995.
20. Gavin Lowe. Analysing protocols subject to guessing attacks. In Joshua Guttman, editor, *In Proceedings of the Workshop on Issues in the Theory of Security (WITS'02)*, Portland, Oregon, USA, January 2002.
21. Zuzana Rjaskova. Electronic voting schemes. Master's thesis, Comenius University, 2002. www.tcs.hut.fi/ helger/crypto/link/protocols/voting.html.
22. Paul F. Syverson, David M. Goldschlag, and Michael G. Reed. Anonymous connections and onion routing. In *IEEE Symposium on Security and Privacy, May 4–7, 1997, Oakland, California*, pages 44–54. IEEE Computer Society Press, 1997.