Computer-Assisted Verification of a Protocol for Certified Email

Martín Abadi[1] and Bruno Blanchet[2]

[1] Computer Science Department, University of California, Santa Cruz
abadi@cs.ucsc.edu
[2] Département d'Informatique, École Normale Supérieure, Paris
and Max-Planck-Institut für Informatik, Saarbrücken
Bruno.Blanchet@ens.fr

Abstract. We present the formalization and verification of a recent cryptographic protocol for certified email. Relying on a tool for automatic protocol analysis, we establish the key security properties of the protocol. This case study explores the use of general correspondence assertions in automatic proofs, and aims to demonstrate the considerable power of the tool and its applicability to non-trivial, interesting protocols.

1 Introduction

A great deal of effort has been invested in the development of techniques for specifying and verifying security protocols in recent years. This effort is justified, in particular, by the seriousness of security flaws and the relative simplicity of security protocols. It has produced a number of interesting methods and effective tools. These range from mathematical frameworks for manual proofs to fully automatic model-checkers. The former are fundamentally constrained by the unreliability and time-scarcity of human provers. The latter tend to be limited to basic properties of small systems, such as the secrecy of session keys in finite-state simplifications of protocols; they may be viewed as useful but ultimately limited automatic testers. The development of automatic or semi-automatic tools that overcome these limitations is an important problem and the subject of active research.

In previous work, we have developed a protocol checker [1, 6, 7] that can establish secrecy and authenticity properties of protocols represented directly as programs in a minimal programming notation (an extension of the pi calculus). The protocols need not be finite-state; the tool can deal with an unbounded number of protocol sessions, even executed in parallel. Nevertheless, the proofs are fully automatic and often fast.

This paper reports on a fairly ambitious application of this tool in the verification of a recently published protocol for certified email [2]. The protocol allows a sender to send an email message to a receiver, in such a way that the receiver gets the message if and only if the sender obtains an unforgeable receipt for the

message. The protocol is non-trivial, partly because of a number of real-world constraints. The verification yields assurance about the soundness of the protocol. It also suggests a promising method for reasoning about other, related protocols.

This case study aims to demonstrate the considerable power of the tool and its applicability to interesting protocols. It has also served in guiding certain improvements to the tool. Specifically, formalizing the main properties of the protocol has lead us to a generalization of the tool to handle a large class of correspondence assertions [17]. The bulk of the proofs remains fully automatic; for the code presented in this paper, the automatic proofs take only 80 ms on an Intel Xeon 1.7 GHz processor. Easy manual arguments show that the correspondence assertions capture the expected security guarantees for the protocol. Because the protocol is expressed directly in a programming notation, without limitation to finite-state instances, the need for additional arguments to justify the protocol representation is, if not eliminated, drastically reduced.

Outline We review the description of the certified email protocol in Section 2. We also review our verification technique, in Section 3, and show how to extend it so as to handle the correspondence assertions on which we rely here. We explain our formal specification of the protocol in Section 4, then prove its security properties in Section 5. We conclude in Section 6, mentioning our work on the analysis of more elaborate variants of the protocol.

Related work It is fairly common to reason informally about security protocols, with various degrees of thoroughness and rigor. For instance, Krawczyk gave some informal arguments about the properties of the Skeme protocol (a variant of the core of IPsec) when he introduced Skeme [10]. Similarly, the presentation of the protocol that we treat in this paper included informal proof sketches for some of its central properties [2]. Generally, such proofs are informative, but far from complete and fully reliable.

It has been widely argued that formal proofs are particularly important for security protocols, because of the seriousness of security flaws. Nevertheless, formal proofs for substantial, practical protocols remain relatively rare. Next we mention what seem to be the most relevant results in this area.

The theorem prover Isabelle has been used for verifying (fragments of) several significant protocols with an inductive method, in particular Kerberos [4, 5], TLS (a descendant of SSL 3.0) [14], and the e-commerce protocol SET [3]. Following the same general approach, G. Bella and C. Longo are currently working on the verification the certified email protocol that we treat in this paper. It will be interesting to compare definitions and proofs.

Meadows has used the NRL protocol verifier for reasoning about the Internet Key Exchange protocol, a component of proposals for IP security [12]. The reasoning, although enlightening in some respects, was not a full verification.

The finite-state model checker Murphi has served for the verification of SSL 3.0 [13] and of contract-signing protocols [16]. Somewhat similarly, Mocha has been used for the verification of contract-signing protocols within a game

model [11]. (Contract-signing protocols have some high-level similarities to protocols for certified email.) Largely because of tool characteristics, the proofs in Murphi and Mocha require non-trivial encodings and simplifications of the protocols under consideration, and of their properties.

Schneider has studied a non-repudiation protocol in a CSP-based model, with manual proofs [15]. That protocol, which is due to Zhou and Gollmann, has commonalities with protocols for certified email.

Gordon and Jeffrey have been developing attractive type-based techniques for proving correspondence assertions of protocols [8, 9]. To date, they have had to support only limited forms of correspondence assertions, and they have included a limited repertoire of cryptographic primitives. In these respects, their system is insufficient for the protocol that we treat in this paper, and weaker than the tool that we use. On the other hand, those limitations are probably not intrinsic.

2 The Protocol

This section recalls the description of the protocol for certified email. This section is self-contained, but we refer the reader to the original description [2] for additional details and context.

Protocols for certified email aim to allow a sender, S, to send an email message to a receiver, R, so that R gets the message if and only if S gets a corresponding return receipt. Some protocols additionally aim to ensure the confidentiality of the message.

This protocol, like several others, relies on an on-line trusted third party, TTP. For simplicity, the channels between TTP and the other parties are assumed to guarantee reliable message delivery. Furthermore, the channel between R and TTP should provide secrecy and authentication of TTP to R. (These properties are needed when R gives a password or some other secret to TTP in order to prove its identity.) In practice such a channel might be an SSL connection or, more generally, a channel protected with symmetric keys established via a suitable protocol.

The protocol supports several options for authenticating R. For each email, S picks one of the options; the choice is denoted by authoption. We have done the proofs for all options. For the sake of brevity, however, we only show our results for the mode called BothAuth, in which both TTP and S authenticate R.

- TTP authenticates R using a shared secret RPwd—a password that identifies R to TTP.
- S authenticates R using a query/response mechanism. R is given a query q by the receiver software and r is the response that S expects R to give.

The protocol relies on a number of cryptographic primitives. The corresponding notation is as follows. $\mathsf{E}(\mathsf{k}, \mathsf{m})$ is an encryption of m using key k under some symmetric encryption algorithm. $\mathsf{H}(\mathsf{m})$ is the hash of m in some collision-resistant hashing scheme. $\mathsf{A}(\mathsf{k}, \mathsf{m})$ is an encryption of m using key k under some public-key encryption algorithm. $\mathsf{S}(\mathsf{k}, \mathsf{m})$ is a signature of m using key k under a public-key

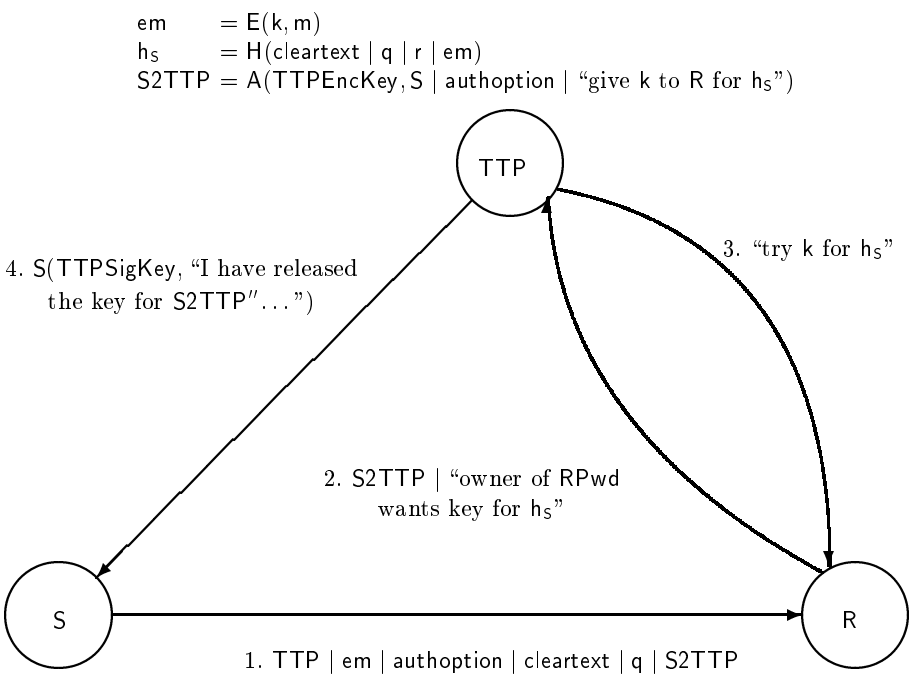

Fig. 1. Protocol sketch

signature algorithm. Finally $\mathsf{m}_1 \mid \cdots \mid \mathsf{m}_n$ denotes the unambiguous concatenation of the m_is.

TTP has a public key TTPEncKey that S can use for encrypting messages for TTP, and a corresponding secret key TTPDecKey. TTP also has a secret key TTPSigKey that it can use for signing messages and a public key TTPVerKey that S can use for verifying these signatures.

In the first step of the protocol, S encrypts its message under a freshly generated symmetric key, encrypts this key under TTPEncKey, and mails this and the encrypted message to R. Then R forwards the encrypted key to TTP. After authenticating R appropriately, TTP releases the key to R (so R can decrypt and read the message) and sends a receipt to S. In more detail, the exchange of messages goes as follows. (Figure 1, adapted from [2], shows some of this detail.)
Step 1: When S wishes to send a message m to R:

1.1. S generates a key k. S also picks authoption (BothAuth in this paper). S knows or generates a query q to which R should respond r.
1.2. S encrypts m with k, letting em = E(k, m).
1.3. S then computes $\mathsf{h_S}$ = H(cleartext | q | r | em). This hash will both identify the message to TTP and serve for authenticating R. The part cleartext is simply a header.
1.4. S computes S2TTP = A(TTPEncKey, S | authoption | "give k to R for $\mathsf{h_S}$").

1.5. S sends Message 1:

MESSAGE 1, S to R: TTP | em | authoption | cleartext | q | S2TTP

Step 2: When R receives a message of the form: TTP | em′ | authoption′ | cleartext′ | q′ | S2TTP′:

2.1. R reads cleartext′, decides whether it wants to read the message with the assistance of TTP, and checks the authentication option (to be BothAuth, in this paper). If R decides to proceed, it constructs a response r′ to query q′, and recalls its password RPwd for TTP.
2.2. R computes $h_R = H(\text{cleartext}' \mid q' \mid r' \mid \text{em}')$.
2.3. R sends Message 2:

MESSAGE 2, R to TTP: S2TTP′ | "owner of RPwd wants key for h_R"

This message and the next one are transmitted on the secure channel that links R and TTP.

Steps 3 and 4: When TTP receives of a message of the form S2TTP″ | "owner of RPwd′ wants key for h'_R":

3.1. TTP tries to decrypt S2TTP″ using TTPDecKey. The cleartext should be of the form S | authoption″ | "give k′ to R′ for h'_S".
3.2. TTP checks that RPwd′ is the password for R′ and that h'_S equals h'_R. If TTP's checks succeed, it proceeds with Messages 3 and 4.
3.3. TTP sends Message 3:

MESSAGE 3, TTP to R: "try k′ for h'_R"

Upon receipt of such a message R uses k′ to decrypt em′, obtaining m.
4.1. TTP sends Message 4:

MESSAGE 4, TTP to S:
S(TTPSigKey, "I have released the key for S2TTP″ to R′")

4.2. When S receives Message 4, it checks this receipt. Later on, if S ever wants to prove that R has received m to a judge, S can provide this message, em, k, cleartext, q, and r, and the judge should check that these values and TTP's public key match.

3 The Verification Tool

In this section we review the verification tool that we employ for our analysis (see [1, 6, 7] for further information). We also explain how we extend this tool.

The tool requires expressing protocols in a formal language, which we describe below. The semantics of this language is the point of reference for our proofs. The tool is sound, with respect to this semantics. (So proofs with the tool can guarantee the absence of attacks captured in this semantics, but not necessarily of other attacks.) On the other hand, the tool is not complete; however, it is successful in substantial proofs, as we demonstrate.

3.1 The Input Language

The verifier takes as input the description of a protocol in a little programming language, an extension of the pi calculus. This calculus represents messages by terms $M, N, \ldots$, and programs by processes $P, Q, \ldots$. Identifiers are partitioned into names, variables, constructors, and destructors. We often use a, b, and c for names, x for a variable, f for a constructor, and g for a destructor.

Constructors are functions that serve for building terms. Thus, the terms are variables, names, and constructor applications of the form $f(M_1, \ldots, M_n)$. A constructor f of arity n is introduced with the declaration **fun** f/n. On the other hand, destructors do not appear in terms, but only manipulate terms in processes. They are partial functions on terms that processes can apply. The process **let** $x = g(M_1, \ldots, M_n)$ **in** P **else** Q tries to evaluate $g(M_1, \ldots, M_n)$; if this succeeds, then x is bound to the result and P is run, else Q is run. More precisely, a destructor g of arity n is described with a set of reduction rules of the form $g(M_1, \ldots, M_n) \to M$ where $M_1, \ldots, M_n, M$ are terms without free names. These reduction rules are specified in a **reduc** declaration. We extend these rules by $g(M'_1, \ldots, M'_n) \to M'$ if and only if there exists a substitution σ and a reduction rule $g(M_1, \ldots, M_n) \to M$ in the declaration of g such that $M'_i = \sigma M_i$ for all $i \in \{1, \ldots, n\}$, and $M' = \sigma M$. Pairing and encryption are typical constructors; projections and decryption are typical destructors. More generally, we can represent data structures and cryptographic operations using constructors and destructors, as can be seen below in our coding of the protocol for certified email.

The process calculus includes auxiliary events that are useful in specifying security properties. The process **begin**$(M).P$ executes the event **begin**(M), then P. The process **end**$(M).P$ executes the event **end**(M), then P. We prove security properties of the form "if a certain **end** event has been executed, then certain **begin** events have been executed".

Most other constructs of the language come from the pi calculus. The input process $\mathbf{in}(M, x); P$ inputs a message on channel M, then runs P with the variable x bound to the input message. The output process $\mathbf{out}(M, N); P$ outputs the message N on the channel M, then runs P. The nil process $\mathbf{0}$ does nothing. The process $P \mid Q$ is the parallel composition of P and Q. The replication $!P$ represents an unbounded number of copies of P in parallel. The restriction **new** $a; P$ creates a new name a, then executes P. The let definition **let** $x = M$ **in** P runs P with x bound to M, and **if** $M = N$ **then** P **else** Q runs P when M equals N, otherwise it runs Q. As usual, we may omit an **else** clause when it consists of $\mathbf{0}$.

The name a is bound in the process **new** $a; P$. The variable x is bound in P in the processes $\mathbf{in}(M, x); P$, **let** $x = g(M_1, \ldots, M_n)$ **in** P **else** Q, and **let** $x = M$ **in** P. We write $fn(P)$ and $fv(P)$ for the sets of names and variables free in P, respectively. A process is closed if it has no free variables; it may have free names. Processes that represent complete protocols are always closed.

The formal semantics of this language can be defined by a reduction relation on configurations, as explained in the appendix. (This semantics, as well as the

proof method, have evolved in minor ways since previous publications [7].) A reduction trace is a finite sequence of reduction steps.

We generally assume that processes execute in the presence of an adversary, which is itself a process in the same calculus. The adversary need not be programmed explicitly; we usually establish results with respect to all adversaries. We need only constrain the initial knowledge of the adversary, which we represent with a set of names *Init*, and restrict the adversary not to use auxiliary events:

Definition 1. *Let Init be a finite set of names. The closed process Q is an* *Init*-adversary *if and only if $fn(Q) \subseteq Init$ and Q does not contain* **begin** *or* **end** *events.*

3.2 The Internal Representation and the Proof Engine

Given a protocol expressed as a process in the input language, the verifier first translates it, automatically, into a set of Horn clauses (logic programming rules).

In the rules, messages are represented by patterns, which are expressions similar to terms except that names a are replaced with functions $a[\ldots]$. A free name a is replaced with the function without parameter $a[]$ (or simply a), while a bound name is replaced with a function of inputs above the restriction that creates the name. The rules are written in terms of four kinds of facts:

- $attacker(p)$ means that the adversary may have the message p;
- $mess(p, p')$ means that the message p' may be sent on channel p;
- $begin(p)$ means that the event $\mathbf{begin}(p)$ may have been executed;
- $end(p)$ means that the event $\mathbf{end}(p)$ may have been executed.

The verifier uses a resolution-based solving algorithm in order to determine properties of the protocol. Specifically, it implements a function $solve_{P,Init}(F)$ that takes as parameters the protocol P, the initial knowledge of the adversary *Init*, and a fact F, and returns a set of Horn clauses. This function first translates the protocol into a set of Horn clauses $\mathcal{C}$, then saturates this set using a resolution-based algorithm [7, Sections 4.2 and 4.3]. Finally, this function determines what is derivable. More precisely, let F' be an instance of F. Let $\mathcal{C}_b$ be any set of closed facts $begin(p)$. We can show that the fact F' is derivable from $\mathcal{C} \cup \mathcal{C}_b$ if and only if there exist a clause $F_1 \wedge \ldots \wedge F_n \rightarrow F_0$ in $solve_{P,Init}(F)$ and a substitution σ such that $F' = \sigma F_0$ and $\sigma F_1, \ldots, \sigma F_n$ are derivable from $\mathcal{C} \cup \mathcal{C}_b$. In particular, when $solve_{P,Init}(F) = \emptyset$, no instance of F is derivable. Other values of $solve_{P,Init}(F)$ give information on which instances of F are derivable, and under which conditions. In particular, the *begin* facts in the hypotheses of the clauses in $solve_{P,Init}(F)$ indicate which *begin* facts must be in $\mathcal{C}_b$ in order to prove F', that is, which **begin** events must be executed.

3.3 Secrecy

In the input language, we define secrecy in terms of the communications of a process that executes in parallel with an arbitrary attacker. This treatment of

secrecy is a fairly direct adaptation of our earlier one [1], with a generalization from free names to terms.

Definition 2 (Secrecy). *Let P be a closed process and M a term such that $fn(M) \subseteq fn(P)$. The process P* preserves the secrecy of all instances of *M from Init if and only if for any Init-adversary Q, any $c \in fn(Q)$, and any substitution σ, no reduction trace of $P \mid Q$ executes* $\mathbf{out}(c, \sigma M)$.

The following result provides a method for proving secrecy properties:

Theorem 1 (Secrecy). *Let P be a closed process. Let M be a term such that $fn(M) \subseteq fn(P)$. Let p be the pattern obtained by replacing names a with patterns $a[]$ in the term M. Assume that $solve_{P,Init}(attacker(p)) = \emptyset$. Then P preserves the secrecy of all instances of M from Init.*

Basically, this result says that if the fact $attacker(p)$ is not derivable then the adversary cannot obtain the term M that corresponds to p.

3.4 Correspondence Assertions

As shown in [7], the verifier can serve for establishing correspondence assertions [17] of the restricted form "if $\mathbf{end}(M)$ has been executed, then $\mathbf{begin}(M)$ must have been executed". Here, we extend this technique so as to prove specifications of the more general form "if $\mathbf{end}(N)$ has been executed, then $\mathbf{begin}(M_1)$, ..., $\mathbf{begin}(M_n)$ must have been executed". Deemphasizing technical differences with Woo's and Lam's definitions, we refer to these specifications as correspondence assertions. Below, we use correspondence assertions for establishing that R gets S's message if and only if S gets a corresponding receipt.

We define the meaning of these specifications as follows:

Definition 3 (Correspondence). *Let P be a closed process and $N, M_1, \ldots, M_l$ be terms such that $fn(N) \cup fn(M_1) \cup \ldots \cup fn(M_l) \subseteq fn(P)$. The process P* satisfies the correspondence assertion $\mathbf{end}(N) \rightsquigarrow \mathbf{begin}(M_1), \ldots, \mathbf{begin}(M_l)$ *with respect to Init-adversaries if and only if, for any Init-adversary Q, for any σ defined on the variables of N, if $\mathbf{end}(\sigma N)$ is executed in some reduction trace of $P \mid Q$, then we can extend σ so that for $k \in \{1, \ldots, l\}$, $\mathbf{begin}(\sigma M_k)$ is executed in this trace as well.*

Analogously to Theorem 1, the next theorem provides a method for proving these correspondence assertions with our verifier.

Theorem 2 (Correspondence). *Let P be a closed process and $N, M_1, \ldots, M_l$ be terms such that $fn(N) \cup fn(M_1) \cup \ldots \cup fn(M_l) \subseteq fn(P)$. Let $p, p_1, \ldots, p_l$ be the patterns obtained by replacing names a with patterns $a[]$ in the terms N, M_1, ..., M_l, respectively. Assume that, for all rules R in $solve_{P,Init}(end(p))$, there exist σ' and H such that $R = H \wedge begin(\sigma' p_1) \wedge \ldots \wedge begin(\sigma' p_l) \rightarrow end(\sigma' p)$. Then P satisfies the correspondence assertion $\mathbf{end}(N) \rightsquigarrow \mathbf{begin}(M_1), \ldots, \mathbf{begin}(M_l)$ with respect to Init-adversaries.*

Intuitively, the condition on R means that, for the fact $end(\sigma' p)$ to be derivable, $begin(\sigma' p_1)$, ..., $begin(\sigma' p_l)$ must be derivable. The conclusion of the theorem is the corresponding statement on events: if $\mathbf{end}(\sigma N)$ has been executed, then $\mathbf{begin}(\sigma M_1)$, ..., $\mathbf{begin}(\sigma M_l)$ must have been executed as well.

4 Formalizing the Protocol

In order to analyze the protocol for certified email, we program it in the verifier's input language, following the informal specification rather closely. In the code below, comments such as "Step 1.1" refer to corresponding steps of the informal specification.

The code represents the situation in which all principals proceed honestly. In Section 5, when we consider situations in which S or R are adversarial and may therefore not execute this code, we simplify the specification accordingly. In addition, in order to specify and prove security properties, we add events (at the program points marked *Event* S, *Event* R, and *Event* TTP).

```
(* Public-key cryptography *)
fun pk/1. fun A/2.
reduce decA(y, A(pk(y), x)) = x.

(* Signatures *)
fun S/2. fun Spk/1.
reduce checkS(Spk(y), S(y, x)) = x.
reduce getS(S(y, x)) = x.

(* Shared-key cryptography *)
fun E/2.
reduce decE(y, E(y, x)) = x.

(* Hash function *)
fun H/1.

(* Constants to identify messages *)
fun Give/0. fun Wants/0. fun Try/0. fun Released/0.
(* Constant authentication modes. We consider only BothAuth here *)
fun BothAuth/0.

(* Function from R's password to R's name *)
fun PasswdTable/1.

(* It is assumed that an attacker cannot relate q and r = Reply(h, q) except for
   the hosts h it creates itself *)
private fun Reply/2.
reduce ReplyOwnHost(x, q) = Reply(PasswdTable(x), q).

(* Build a message *)
private fun Message/2.

(* Secrecy assumptions *)
```

```
not TTPDecKey. not TTPSigKey.

(* Free names (public and private constants) *)
free c, cleartext, Sname, TTPname.
private free TTPDecKey, TTPSigKey, RPwd.

let processS =
  (* The attacker chooses possible recipients of the message *)
  in(c, recipient);

  (* Build the message to send *)
  new msgid; let m = Message(recipient, msgid) in

  (* Step 1 *)
  new k; new q; let r = Reply(recipient, q) in                    (* Step 1.1 *)
  let em = E(k, m) in                                             (* Step 1.2 *)
  let hs = H((cleartext, q, r, em)) in                            (* Step 1.3 *)
                                                                  (* Step 1.4 *)
  let S2TTP = A(TTPEncKey, (Sname, BothAuth, (Give, k, recipient, hs))) in
  (* Event S [to be added later] *)
  out(recipient, (TTPname, em, BothAuth, cleartext, q, S2TTP)));  (* Step 1.5 *)

  (* Step 4.2 *)
  !
  in(Sname, inmess4);
  let (= Released, = S2TTP, = recipient) = checkS(TTPVerKey, inmess4) in
      (* S knows that the recipient has read the message *)
      0
  else out(Sname, inmess4).

let processR =
  (* Step 2 *)
  in(Rname, (= TTPname, em2, = BothAuth, cleartext2, q2, S2TTP2));
  let r2 = Reply(Rname, q2) in                                    (* Step 2.1 *)
  let hr = H((cleartext2, q2, r2, em2)) in                        (* Step 2.2 *)
  (* Establish the secure channel R-TTP *)
  new secchannel;
  out(ChannelToTTP, Rname);
  out(ChannelToTTP, secchannel);
  let outchannel = (TTPname, secchannel) in
  let inchannel = (Rname, secchannel) in
  (* Event R [to be added later] *)
  out(outchannel, (S2TTP2, (Wants, RPwd, hr)));                   (* Step 2.3 *)

  (* Step 3.3 *)
  !
  in(inchannel, (= Try, k3, = hr));
  let m3 = decE(k3, em2) in
```

```
  (* R has obtained the message m3 = m *)
  end(Rreceived(m3)).

let processTTP =
  (* Establish the secure channel R-TTP *)
  in(ChannelToTTP, receivername);
  in(ChannelToTTP, secchannel);
  let inchannel = (TTPname, secchannel) in
  let outchannel = (receivername, secchannel) in

  (* Step 3 *)
  in(inchannel, (S2TTP3, (= Wants, RPwd3, hr3)));
  let (Sname3, authoption3, (= Give, k3, R3, = hr3)) =          (* Step 3.1 *)
      decA(TTPDecKey, S2TTP3) in
  if R3 = PasswdTable(RPwd3) then                               (* Step 3.2 *)
  if R3 = receivername then
  (* Event TTP [to be added later] *)
  out(outchannel, (Try, k3, hr3));                              (* Step 3.3 *)

  (* Step 4.1 *)
  out(Sname3, S(TTPSigKey, (Released, S2TTP3, R3))).

process
  let TTPEncKey = pk(TTPDecKey) in out(c, TTPEncKey);
  let TTPVerKey = Spk(TTPSigKey) in out(c, TTPVerKey);
  let Rname = PasswdTable(RPwd) in out(c, Rname);
  new ChannelToTTP;
  ((!processS) | (!processR) | (!processTTP)
      | (!in(c, m); out(ChannelToTTP, m)))
```

This code first declares cryptographic primitives. For instance, the constructor A is the public-key encryption function, which takes two parameters, a public key and a cleartext, and returns a ciphertext. The constructor pk computes a public key from a secret key. The destructor decA is the corresponding decryption function. From a ciphertext $\mathsf{A}(\mathsf{pk}(y), x)$ and the corresponding secret key y, it returns the cleartext x. Hence we give the rule $\mathsf{decA}(y, \mathsf{A}(\mathsf{pk}(y), x)) = x$. We assume perfect cryptography, so the cleartext can be obtained from the ciphertext only when one has the decryption key. We define signatures, shared-key encryption, and a hash function analogously. Note that the constructor Spk that builds a public key for signatures from a secret key is different from the constructor pk that builds a public key for encryptions. The destructor checkS checks the signature, while getS returns the cleartext message without checking the signature. (In particular, the adversary may use getS in order to obtain message contents from signed messages.) Concatenation is represented by tuples, which are pre-declared by default. We also declare a number of constants that appear in messages.

The constructor PasswdTable computes the name of a receiver from its password. This constructor represents the password table (*host name*, *host password*). Since all host names are public but some passwords are secret, the adversary must not be able to compute the appropriate password from a host name, so we define a function that maps passwords to host names but not the converse: *host name* = PasswdTable(*host password*). One advantage of this encoding is that we can compactly model systems with an unbounded number of hosts.

The challenge-response authentication of R by S goes as follows. S creates an arbitrary query q, and the reply r to this query is computed by the constructor Reply, so $r = \mathsf{Reply}(h, q)$ where h is the recipient host name. Both S and R can use the constructor Reply. However, this constructor is declared **private**, that is, the adversary cannot apply it. (Otherwise, it could impersonate R.) The adversary must be able to compute replies for hosts that it creates, that is, when it has the password of the host. Therefore, we define a public destructor ReplyOwnHost that computes a reply from the query and the password of the host.

The constructor Message builds the messages that S sends to R. We assume that these messages are initially secret, so we make the constructor private. We also assume that S sends different messages to different recipients, so let a message be a function of the recipient and of a message identifier.

Secrecy assumptions correspond to an optimization of our verifier. The declaration **not** M indicates to the verifier that M is secret. The verifier can then use this information in order to speed up the solving process. At the end of solving, the verifier checks that the secrecy assumption is actually true, so that a wrong secrecy assumption leads to an error message but not to an incorrect result.

The declaration **free** declares public free names. c is a public channel, cleartext is the header of the messages sent by S, and Sname and TTPname are the names of S and TTP, respectively. R's name is Rname = PasswdTable(RPwd) so it not a free name. (It is declared at the end of the protocol.) The declaration **private free** declares private free names (not known by the adversary); TTPDecKey and TTPSigKey are TTP's secret keys, and RPwd is R's password.

The processes *processS*, *processR*, and *processTTP* represent S, R, and TTP, respectively. These processes are composed in the last part of the protocol specification. This part computes TTP's public encryption key from its secret key by the constructor pk: TTPEncKey = pk(TTPDecKey). The public key TTPEncKey is output on the public channel c so that the adversary can have TTPEncKey. We proceed similarly for the key pair (TTPSigKey, TTPVerKey). At last, we compute R's name from its password: Rname = PasswdTable(RPwd). This name is public, so we send it on channel c, so that the adversary can have it. In the following, we use Rname as an abbreviation for the term PasswdTable(RPwd). The role of ChannelToTTP and of the last element of the parallel composition is explained below in the description of *processR*.

The process *processS* first receives the name of the host to which S is going to send its message, on the public channel c. Thus, the adversary can choose that host. This conservative assumption implies that S can send its message to any host. Then S builds the message: it creates a new message id *msgid*,

and builds the message m by calling the constructor Message. Then it executes the steps of the protocol description. For instance, in step 1.1, it creates a new key k by **new** k, a new query q by **new** q, and computes the corresponding reply r. In step 1.4, the sentence "give k to *recipient* for *hs*" is represented by a tuple containing the constant Give and the parameters k, *recipient*, and *hs*. Other sentences are represented analogously. Note that, at step 1.5, we send the message to the recipient on channel *recipient*. In our coding of the protocol, the channel always indicates the destination of the message. This indication makes it easier to define the meaning of "a message reaches its destination", but it is only an indication: when the channel is public, the adversary may still obtain the message or send its own messages on the channel. In the destructor application of step 4.2, we use a pattern-matching construct:

$$\textbf{let } (= \mathsf{Released}, = \mathit{S2TTP}, = \mathit{recipient}) = \ldots \textbf{ in } \ldots$$

A pattern $(p_1, \ldots, p_n)$ matches a tuple of arity n, when $p_1, \ldots, p_n$ match the components of the tuple. A pattern x matches any term, and binds x to this term. A pattern $= M$ matches only the term M. So the destructor application of step 4.2 succeeds if and only if $\mathit{inmess4} = \mathsf{S}(\mathsf{TTPSigKey}, (\mathsf{Released}, \mathit{S2TTP}, \mathit{recipient}))$. The same pattern-matching construct is used for message input. When the check of *inmess4* fails, the incoming message *inmess4* is returned on the channel Sname (by the else clause of the destructor application), so that another session of S can get it. We assume that the execution is fair, so that all sessions of S get a chance to have the receipt *inmess4*. Moreover, because of the replication at the beginning of step 4.2, S still waits for a receipt from TTP even after receiving a wrong receipt. In an actual implementation, S would store a set of the messages it has sent recently and for which it has not yet obtained a receipt. When obtaining a receipt, it would look for the corresponding message in this set. Our coding represents this lookup by returning the receipt on Sname until it is consumed by the right session of S.

The process *processR* first executes steps 2.1 and 2.2, then it establishes a secure connection with TTP. The informal specification does not spell out the details related to this connection, so we need to pick them. Several reasonable choices are available; we explore one here and mention others in Section 6. In order to establish the connection with TTP, R employs an asymmetric channel ChannelToTTP (created at the end of the protocol description) on which anybody can write but only TTP can read. For starting a connection with TTP, one sends its own name *receivername* (here Rname) and a new name *secchannel* on ChannelToTTP. Further exchanges between R and TTP are then done on channels (TTPname, *secchannel*) from R to TTP and (*receivername*, *secchannel*) from TTP to R. We use pairs for channels so as to mention explicitly the destination host in the channel name. One might see some similarity with TCP connections, in which packets contain destination addresses. Since the name *secchannel* created by R is secret, only R and TTP will be able to send or receive messages on (TTPname, *secchannel*) and (*receivername*, *secchannel*), so the channel R-TTP is indeed secure. This channel provides authentication of TTP, since only TTP

can read on ChannelToTTP. Any host can send messages on ChannelToTTP, and thus start a connection with TTP. So the authentication of R is not provided by the channel but by the password check that TTP performs (in step 3.2). R writes on that channel, TTP reads on it. In order to allow the adversary to write on that channel, we use a relay process ($!\mathbf{in}(\mathsf{c}, m); \mathbf{out}(\mathsf{ChannelToTTP}, m)$) (last line of the protocol description) that gets a message on c and resends it on ChannelToTTP. Thus, by sending a message on c, the adversary can send it on ChannelToTTP. After establishing the connection with TTP, R continues the execution of steps 2.3 and 3.3. In the end, R emits the event *Rreceived*(*m3*), to note that R has correctly received the message *m3*. Below, this event is useful in the security proofs.

The process *processTTP* first establishes a secure channel with a message recipient, as explained above. Then it executes step 3. Note that, between steps 3.2 and 3.3, it checks that its interlocutor in the connection, *receivername*, actually corresponds to the expected receiver of the message, *R3*. This check ensures that the message on *outchannel* goes to the expected receiver of the message. Finally, TTP sends the key *k3* to the receiver of the message (step 3.3) and the receipt to the sender (step 4.1).

5 Results

In this section we present the proofs of the main security properties of the protocol. We heavily rely on the verifier for these proofs.

5.1 Secrecy

Let P_0 be the process that represents the protocol. The verifier can prove automatically that this process preserves the secrecy of the message m sent by S to R.

Proposition 1. *Let* $Init = \{\mathsf{Sname}, \mathsf{TTPname}, \mathsf{c}, \mathsf{cleartext}\}$. *The process* P_0 *preserves the secrecy of all instances of* Message(Rname, i) *from* $Init$.

Automatic proof: We give the appropriate query *attacker*(Message(Rname, i)). The tool computes $solve_{P_0, Init}(attacker(\mathsf{Message}(\mathsf{Rname}, i))) = \emptyset$. Hence, by Theorem 1, the process P_0 preserves the secrecy of Message(Rname, i) from $Init$. □

5.2 Receipt

The main correctness property of the protocol is the following: R receives the message m if and only if S gets a proof that R has received the message. This proof should be such that, if S goes to a judge with it, the judge can definitely say that R has received the message.

This property holds only when the delivery of messages is guaranteed on the channels from TTP to R, from TTP to S, and from S to the judge, hence the following definition.

Definition 4. *We say that a message m sent on channel c* reaches its destination *if and only if it is eventually received by an input on channel c in the initial process P_0 or a process derived from P_0. If the adversary receives the message, it reemits the message on channel c.*

Furthermore, we use the following fairness hypotheses:

- If infinitely often a reduction step can be executed, then it will eventually be executed.
- If a message m is sent on channel c, and some inputs on channel c reemit it, that is, they execute $\mathbf{in}(c, m) \ldots \mathbf{out}(c, m)$, and some do not reemit m on c, then m will eventually be received by an input that does not reemit it.

Although this definition and these hypotheses are stated somewhat informally, they can be made precise in terms of the semantics of the language. Several variants are possible.

The fact that messages reach their destination and the fairness hypotheses cannot be taken into account by our verifier, so it cannot prove the required properties in a fully automatic way. Still, the verifier can prove a correspondence assertion that constitutes the most important part of the proof. Indeed, we have to show properties of the form: if some event e_1 has been executed, then some event e_2 has or will be executed. The verifier shows automatically the correspondence assertion: if e_1 has been executed then some events e'_2 have been executed *before* e_1. We show manually that if the events e'_2 have been executed, then e_2 will be executed *after* e'_2. Thus the correspondence assertion captures the desired security property. The manual proof just consists in following the execution steps of the process after e'_2. It is much simpler than the first part, which should go backward through all possible execution histories leading to e_1. Fortunately, the first part is fully automatic.

We use the following process to represent the judge to which the informal specification of the protocol alludes:

fun Received/0. **free** Judgename.

let *processJudge* =
  (S must send* TTP*'s certificate plus other information *)*
  in(Judgename, (*certif*, *Sname5*, *k5*, *cleartext5*, *q5*, *r5*, *em5*));
  let (= Released, *S2TTP5*, *Rname5*) = checkS(TTPVerKey, *certif*) **in**
  let *m5* = decE(*k5*, *em5*) **in**
  let *hs5* = H((*cleartext5*, *q5*, *r5*, *em5*)) **in**
  if *S2TTP5* =
      A(TTPEncKey, (*Sname5*, BothAuth, (Give, *k5*, *Rname5*, *hs5*))) **then**
  (The judge says that Rname5 has received m5 *)*
  end(*JudgeSays*(Received, *Rname5*, *m5*)).

The judge receives a certificate from S, tries to check it, and if it succeeds, says that the receiver has received the message. This process is executed in parallel with *processR*, *processTTP*, and *processS*. At the end of *processS*, instead of executing **0**, the sender S sends to the judge:

$\mathbf{out}(\mathsf{Judgename}, (\mathit{inmess4}, \mathsf{Sname}, k, \mathsf{cleartext}, q, r, \mathit{em}))$

The result to prove decomposes into two propositions, Propositions 2 and 3.

Proposition 2. *Assume that the messages from* TTP *sent on Sname3 and from* S *sent on* Judgename *reach their destinations. If* R *has received* m*, then the judge says that* R *has received* m*.*

In this proof, R is included in the adversary: R tries to get a message without S having the corresponding receipt. So we need not constrain R to follow the protocol. The process for R becomes:

$\mathbf{out}(\mathsf{c}, \mathsf{ChannelToTTP}); \mathbf{out}(\mathsf{c}, \mathsf{RPwd}) \mid \mathbf{in}(\mathsf{c}, m); \mathbf{end}(\mathit{Rreceived}(m))$

This process reveals all the information that R has. When the adversary obtains some message m, it can send it on c, thus execute the event $\mathbf{end}(\mathit{Rreceived}(m))$. Since R is included in the adversary, the adversary can compute the constructor Reply, so its declaration becomes: **fun** Reply/2. Writing P_0 for the resulting process that represents the whole system, the proposition becomes, more formally:

Proposition 2'. *Assume that the messages from* TTP *sent on Sname3 and from* S *sent on* Judgename *reach their destinations. Let Init* $=$ {Sname, TTPname, Judgename, c, cleartext}*. For any Init-adversary* Q*, if the event* $\mathbf{end}(\mathit{Rreceived}(\mathsf{Message}(M_x, M_i)))$ *is executed in a reduction trace of* $P_0 \mid Q$ *for some terms* M_x *and* M_i*, then* $\mathbf{end}(\mathit{JudgeSays}(\mathsf{Received}, M_x, \mathsf{Message}(M_x, M_i)))$ *is executed in all continuations of this trace.*

At point *Event* TTP, we introduce the event $\mathbf{begin}(\mathit{TTP_send}(\mathit{Sname3}, \mathsf{S}(\mathsf{TTPSigKey}, (\mathsf{Released}, \mathit{S2TTP3}, \mathit{R3}))))$, to note that TTP sends the receipt $\mathsf{S}(\mathsf{TTPSigKey}, (\mathsf{Released}, \mathit{S2TTP3}, \mathit{R3}))$ to S. At point *Event* S, we introduce the event $\mathbf{begin}(\mathit{S_has}(\mathsf{Sname}, k, \mathsf{cleartext}, q, r, m))$, to note that S has all parameters needed to obtain the answer from the judge (except TTP's receipt).

Automatic part of the proof: We invoke our tool with the query $\mathit{end}(\mathit{Rreceived}(\mathsf{Message}(x, i)))$, to determine under which conditions an instance of the corresponding event may be executed. The tool then computes the set of clauses $\mathit{solve}_{P_0, \mathit{Init}}(\mathit{end}(\mathit{Rreceived}(\mathsf{Message}(x, i))))$ and returns a rule of the form:

$$\begin{aligned}&\mathit{begin}(\mathit{TTP_send}(\mathsf{Sname}, \mathsf{S}(\mathsf{TTPSigKey}, (\mathsf{Released}, \mathsf{A}(\mathsf{TTPEncKey}, (\mathsf{Sname},\\ &\quad \mathsf{BothAuth}, (\mathsf{Give}, p_k, p_x, \mathsf{H}((\mathsf{cleartext}, p_q, p_r, \mathsf{E}(p_k, \mathsf{Message}(p_x, p_i)))))))), p_x))))\wedge\\ &\mathit{begin}(\mathit{S_has}(\mathsf{Sname}, p_k, \mathsf{cleartext}, p_q, p_r, \mathsf{Message}(p_x, p_i)))\wedge\\ &H \rightarrow \mathit{end}(\mathit{Rreceived}(\mathsf{Message}(p_x, p_i)))\end{aligned}$$

for some patterns p_x, p_k, p_q, p_r, p_i, and some hypothesis H.

So, by Theorem 2, if $\mathbf{end}(\mathit{Rreceived}(\mathsf{Message}(M_x, M_i)))$ is executed in a trace of $P_0 \mid Q$, then the events

$$\mathbf{begin}(\mathit{TTP_send}(\mathsf{Sname}, \mathit{certificate}))$$
$$\mathbf{begin}(\mathit{S_has}(\mathsf{Sname}, M_k, \mathsf{cleartext}, M_q, M_r, \mathsf{Message}(M_x, M_i)))$$

are executed in this trace for some terms M_k, M_q, and M_r, with *certificate* = $\mathsf{S}(\mathsf{TTPSigKey}, (\mathsf{Released}, \mathsf{A}(\mathsf{TTPEncKey}, (\mathsf{Sname}, \mathsf{BothAuth}, (\mathsf{Give}, M_k, M_x, \mathsf{H}((\mathsf{cleartext}, M_q, M_r, \mathsf{E}(M_k, \mathsf{Message}(M_x, M_i)))))))), M_x))$.

Manual part of the proof: Since TTP executes **begin**(*TTP_send*(Sname, *certificate*)) as proved above, it is then going to execute **out**(Sname, *certificate*). Since this message reaches its destination, it will then be received by an input on Sname from P_0, that is, by the last input of *processS*. Moreover, the session that has executed **begin**(*S_has*(Sname, M_k, cleartext, M_q, M_r, Message(M_x, M_i))) does not reemit this message, so by the fairness hypothesis, this message will be received by a session of S that does not reemit it. Such a session successfully checks the certificate and sends it to the judge on the channel Judgename. Since this message reaches its destination, it will be received by the input on Judgename in *processJudge*. Then the judge also checks successfully the certificate (the check always succeeds when S's check succeeds), so the judge executes **end**(*JudgeSays*(Received, M_x, Message(M_x, M_i))). □

The verifier proves the required correspondence assertion in a fully automatic way. It is then only a few lines of proof to obtain the desired security property. Moreover, we need not even know in advance the exact correspondence assertion to consider: the verifier tells us which correspondence assertion holds for the given **end** event.

Turning to the guarantees for R, we establish:

Proposition 3. *Assume that the message from* TTP *sent on outchannel reaches its destination. If the judge says that* R *has received* m*, then* R *has received* m*.*

In this proof, S is included in the adversary: S may try to fool the judge into saying that R has received a message it does not have. Therefore, we need not be specific on how S behaves, so the process for S is simply **0**. The adversary can compute the constructor Reply, so its declaration becomes: **fun** Reply/2. Writing P_0 for the resulting process that represents the whole system, the proposition becomes, more formally:

Proposition 3'. *Assume that the message from* TTP *sent on outchannel reaches its destination. Let Init* = {Sname, TTPname, Judgename, c, cleartext}*. For any Init-adversary* Q*, if* **end**(*JudgeSays*(Received, Rname, M_m)) *is executed in a reduction trace of* $P_0 \mid Q$ *for some term* M_m*, then the event* **end**(*Rreceived*(M_m)) *is executed in all continuations of this trace.*

At point *Event* R, we introduce the event **begin**(*R_has*(*secchannel*, *em2*, *hr*)), to note that R has received the encrypted message. At point *Event* TTP, we introduce the event **begin**(*TTP_send*(*outchannel*, (Try, *k3*, *hr3*))) to note that TTP sends the key *k3* to R.

Automatic part: We give our verifier the query *end*(*JudgeSays*(Received, Rname, m)). The result returned by the verifier shows, using Theorem 2, that, if the event **end**(*JudgeSays*(Received, Rname, M_m)) is executed in a reduction trace of $P_0 \mid Q$ for some term M_m, then the events **begin**(*R_has*($M_{secchannel}$, E(M_k,

$M_m), M_{hr}))$ and $\mathbf{begin}(TTP_send((\mathsf{Rname}, M_{secchannel}), (\mathsf{Try}, M_k, M_{hr})))$ are executed in this trace for some terms M_k, $M_{secchannel}$, and M_{hr}.

Manual part: TTP is going to send the Message 3 indicated by the event $\mathbf{begin}(TTP_send(\ldots))$ to R. Since this message reaches its destination, it will be received by the session of R that executes the event $\mathbf{begin}(R_has(\ldots))$ mentioned above (since the value of the channel $(\mathsf{Rname}, M_{secchannel})$ must match). This session then executes $\mathbf{end}(Rreceived(M_m))$. □

6 Conclusion

This paper reports on the formal specification of a non-trivial, practical protocol for certified email, and on the verification of its main security properties. Most of the verification work is done with an automatic protocol verifier, which we adapted for the present purposes. The use of this tool significantly reduces the proof burden. It also reduces the risk of human error in proofs. Although the tool itself has not been verified, we believe that its use is quite advantageous.

We have also specified and verified more elaborate variants of the protocol, through similar methods. (We omit details for lack of space.) In particular, we have treated all four authentication modes. We have also treated three ways of establishing the secure channel between R and TTP: the one explained here, one based on a small public-key protocol, and one based on a simplified version of the SSH protocol with a Diffie-Hellman key agreement (challenging in its own right). For these three versions, the automatic parts of the proofs take 2 min 30 s on an Intel Xeon 1.7 Ghz. The manual parts are as simple as the ones shown above. Writing the specifications was more delicate and interesting than constructing the corresponding proofs.

Acknowledgments

Martín Abadi's research was partly supported by faculty research funds granted by the University of California, Santa Cruz, and by the National Science Foundation under Grants CCR-0204162 and CCR-0208800.

References

1. M. Abadi and B. Blanchet. Analyzing security protocols with secrecy types and logic programs. In *29th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL'02)*, pages 33–44, Portland, OR, Jan. 2002. ACM Press.
2. M. Abadi, N. Glew, B. Horne, and B. Pinkas. Certified email with a light on-line trusted third party: Design and implementation. In *11th International World Wide Web Conference (WWW'02)*, Honolulu, Hawaii, USA, May 2002. ACM Press.
3. G. Bella, F. Massacci, and L. C. Paulson. The verification of an industrial payment protocol: The SET purchase phase. In V. Atluri, editor, *9th ACM Conference on Computer and Communications Security (CCS'02)*, pages 12–20, Washington, DC, Nov. 2002. ACM Press.

4. G. Bella and L. C. Paulson. Using Isabelle to prove properties of the Kerberos authentication system. In *DIMACS Workshop on Design and Formal Verification of Security Protocols*, Piscataway, NJ, Sept. 1997.
5. G. Bella and L. C. Paulson. Kerberos version IV: inductive analysis of the secrecy goals. In J.-J. Quisquater et al., editors, *Computer Security - ESORICS 98*, volume 1485 of *Lecture Notes in Computer Science*, pages 361–375, Louvain-la-Neuve, Belgium, Sept. 1998. Springer Verlag.
6. B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society.
7. B. Blanchet. From secrecy to authenticity in security protocols. In M. Hermenegildo and G. Puebla, editors, *9th International Static Analysis Symposium (SAS'02)*, volume 2477 of *Lecture Notes in Computer Science*, pages 342–359, Madrid, Spain, Sept. 2002. Springer Verlag.
8. A. Gordon and A. Jeffrey. Authenticity by typing for security protocols. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 145–159, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society.
9. A. Gordon and A. Jeffrey. Types and effects for asymmetric cryptographic protocols. In *15th IEEE Computer Security Foundations Workshop (CSFW-15)*, pages 77–91, Cape Breton, Nova Scotia, Canada, June 2002. IEEE Computer Society.
10. H. Krawczyk. SKEME: A versatile secure key exchange mechanism for internet. In *Proceedings of the Internet Society Symposium on Network and Distributed Systems Security (NDSS'96)*, San Diego, CA, Feb. 1996. Available at `http://bilbo.isu.edu/sndss/sndss96.html`.
11. S. Kremer and J.-F. Raskin. Game analysis of abuse-free contract signing. In *15th IEEE Computer Security Foundations Workshop (CSFW-15)*, pages 206–222, Cape Breton, Nova Scotia, Canada, June 2002. IEEE Computer Society.
12. C. Meadows. Analysis of the Internet Key Exchange protocol using the NRL protocol analyzer. In *IEEE Symposium on Security and Privacy*, pages 216–231, Oakland, CA, May 1999. IEEE Computer Society.
13. J. C. Mitchell, V. Shmatikov, and U. Stern. Finite-state analysis of SSL 3.0. In *7th USENIX Security Symposium*, pages 201–216, San Antonio, TX, Jan. 1998.
14. L. C. Paulson. Inductive analysis of the Internet protocol TLS. *ACM Transactions on Information and System Security*, 2(3):332–351, Aug. 1999.
15. S. Schneider. Formal analysis of a non-repudiation protocol. In *11th IEEE Computer Security Foundations Workshop (CSFW-11)*, pages 54–65, Rockport, Massachusetts, June 1998. IEEE Computer Society.
16. V. Shmatikov and J. C. Mitchell. Finite-state analysis of two contract signing protocols. *Theoretical Computer Science*, 283(2):419–450, June 2002.
17. T. Y. C. Woo and S. S. Lam. A semantic model for authentication protocols. In *1993 IEEE Symposium on Research on Security and Privacy*, pages 178–194, Oakland, CA, 1993. IEEE Computer Society.

Appendix: Semantics

A semantic configuration is a pair $E, \mathcal{P}$ where the environment E is a finite set of names and $\mathcal{P}$ is a finite multiset of closed processes. The environment E must contain at least all free names of processes in $\mathcal{P}$. The configuration $\{a_1, \ldots, a_n\}, \{P_1, \ldots, P_n\}$ corresponds to the process $\mathbf{new}\ a_1; \ldots \mathbf{new}\ a_n; (P_1 \mid$

$\ldots \mid P_n$). The semantics of the calculus is defined by a reduction relation $\rightarrow$ on semantic configurations as follows:

$$E, \mathcal{P} \cup \{\, \mathbf{0} \,\} \rightarrow E, \mathcal{P} \qquad \text{(Red Nil)}$$

$$E, \mathcal{P} \cup \{\, !P \,\} \rightarrow E, \mathcal{P} \cup \{\, P, !P \,\} \qquad \text{(Red Repl)}$$

$$E, \mathcal{P} \cup \{\, P \mid Q \,\} \rightarrow E, \mathcal{P} \cup \{\, P, Q \,\} \qquad \text{(Red Par)}$$

$$E, \mathcal{P} \cup \{\, \mathbf{new}\ a; P \,\} \rightarrow E \cup \{a'\}, \mathcal{P} \cup \{\, P\{a'/a\} \,\} \qquad \text{(Red Res)}$$

where $a' \notin E$.

$$E, \mathcal{P} \cup \{\, \mathbf{out}(N, M).Q, \mathbf{in}(N, x).P \,\} \rightarrow E, \mathcal{P} \cup \{\, Q, P\{M/x\} \,\} \qquad \text{(Red I/O)}$$

$$E, \mathcal{P} \cup \{\, \mathbf{let}\ x = g(M_1, \ldots, M_n)\ \mathbf{in}\ P\ \mathbf{else}\ Q \,\} \rightarrow E, \mathcal{P} \cup \{\, P\{M'/x\} \,\}$$

if $g(M_1, \ldots, M_n) \rightarrow M'$ (Red Destr 1)

$$E, \mathcal{P} \cup \{\, \mathbf{let}\ x = g(M_1, \ldots, M_n)\ \mathbf{in}\ P\ \mathbf{else}\ Q \,\} \rightarrow E, \mathcal{P} \cup \{\, Q \,\} \qquad \text{(Red Destr 2)}$$

if there exists no M' such that $g(M_1, \ldots, M_n) \rightarrow M'$

$$E, \mathcal{P} \cup \{\, \mathbf{let}\ x = M\ \mathbf{in}\ P \} \rightarrow E, \mathcal{P} \cup \{\, P\{M/x\} \,\} \qquad \text{(Red Let)}$$

$$E, \mathcal{P} \cup \{\, \mathbf{if}\ M = M\ \mathbf{then}\ P\ \mathbf{else}\ Q \,\} \rightarrow E, \mathcal{P} \cup \{\, P \,\} \qquad \text{(Red Cond 1)}$$

$$E, \mathcal{P} \cup \{\, \mathbf{if}\ M = N\ \mathbf{then}\ P\ \mathbf{else}\ Q \,\} \rightarrow E, \mathcal{P} \cup \{\, Q \,\} \qquad \text{(Red Cond 2)}$$

if $M \neq N$

$$E, \mathcal{P} \cup \{\, \mathbf{begin}(M).P \,\} \rightarrow E, \mathcal{P} \cup \{\, P \,\} \qquad \text{(Red Begin)}$$

$$E, \mathcal{P} \cup \{\, \mathbf{end}(M).P \,\} \rightarrow E, \mathcal{P} \cup \{\, P \,\} \qquad \text{(Red End)}$$

A reduction trace $\mathcal{T}$ of a closed process P is a finite sequence of reductions $fn(P), \{P\} \rightarrow \ldots \rightarrow E', \mathcal{P}'$.

The output $\mathbf{out}(M, N)$ *is executed in a trace* $\mathcal{T}$ if and only if this trace contains a reduction $E, \mathcal{P} \cup \{\, \mathbf{out}(N, M).Q, \mathbf{in}(N, x).P \,\} \rightarrow E, \mathcal{P} \cup \{\, Q, P\{M/x\} \,\}$ for some E, $\mathcal{P}$, x, P, Q.

The event $\mathbf{begin}(M)$ *is executed in a trace* $\mathcal{T}$ if and only if this trace contains a reduction $E, \mathcal{P} \cup \{\, \mathbf{begin}(M).P \,\} \rightarrow E, \mathcal{P} \cup \{\, P \,\}$ for some E, $\mathcal{P}$, P.

The event $\mathbf{end}(M)$ *is executed in a trace* $\mathcal{T}$ if and only if this trace contains a reduction $E, \mathcal{P} \cup \{\, \mathbf{end}(M).P \,\} \rightarrow E, \mathcal{P} \cup \{\, P \,\}$ for some E, $\mathcal{P}$, P.