Computationally Sound Mechanized Proofs for Basic and Public-key Kerberos

B. Blanchet [*]
CNRS & École Normale Supérieure & INRIA
blanchet@di.ens.fr

A.D. Jaggard [†]
DIMACS
Rutgers University
adj@dimacs.rutgers.edu

A. Scedrov [‡]
Department of Mathematics
University of Pennsylvania
scedrov@math.upenn.edu

J.-K. Tsay [Σ]
Department of Mathematics
University of Pennsylvania
jetsay@math.upenn.edu

ABSTRACT

We present a computationally sound mechanized analysis of Kerberos 5, both with and without its public-key extension PKINIT. We prove authentication and key secrecy properties using the prover CryptoVerif, which works directly in the computational model; these are the first mechanical proofs of a full industrial protocol at the computational level. We also generalize the notion of key usability and use CryptoVerif to prove that this definition is satisfied by keys in Kerberos.

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols—*Protocol verification*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Mechanical verification*

General Terms

Security, Verification

Keywords

Kerberos, PKINIT, automatic verification, computational model, key usability

1. INTRODUCTION

There are two main approaches to the verification of cryptographic protocols. One approach, known as the cryptographic or computational model, is based on probability and complexity theory and retains the view of messages as bitstrings and encryption/decryption as probabilistic algorithms. Security properties proved in this model give strong security guarantees, since properties are verified against any probabilistic Turing machine attacking the protocol. Another approach, known as the symbolic or Dolev-Yao model, can be viewed as an idealization of the former approach formulated using an algebra of terms. Messages are abstracted as terms in this algebra and, *e.g.*, encryption algorithms are simply function symbols on these terms. This symbolic model has been successfully applied to uncover problems in the design of security protocols [19, 28, 29, 33]. Moreover, verification methods based on the symbolic model have become efficient and robust enough to be deployed for the analysis of even large commercial protocols [4, 17, 22, 29, 33]. Because by-hand proofs in the detailed computational model are prone to human error and are, even in the symbolic model, very time consuming for complex protocols, effort has been put into developing mechanized or fully automated provers. One area of focus has been provers that work in the symbolic model, which facilitates the use of existing theorem provers and model checkers, and some of the resulting tools have been used to analyze commercial protocols [3, 6, 12, 30]. Since the pioneering work by Abadi and Rogaway [1], attention has also been paid [5, 18, 21] to bridging the gap between the symbolic and the computational models; in these frameworks proofs are carried out in a symbolic model, facilitating automation, and the results can be lifted to the computational model under certain conditions. However, provers based on computationally sound symbolic frameworks (*e.g.*, [20, 38]) are currently at most

[*]This research has been done within the INRIA ABSTRACTION project-team (common with the CNRS and the ENS) and was partly supported by the ANR project ARA SSIA FormaCrypt.

[†]Partially supported by NSF Grants DMS-0239996, CNS-0429689, and CNS-0753492, and by ONR Grant N00014-05-1-0818; this work was started while Jaggard was in the Mathematics Department at Tulane University.

[‡]Partially supported by OSD/ONR CIP/SW URI projects through ONR Grants N00014-01-1-0795 and N00014-04-1-0725. Additional support from NSF Grants CNS-0429689 and CNS-0524059 and from ONR Grant N00014-07-1-1039.

[Σ]Partially supported by ONR Grants N00014-01-1-0795 and N00014-07-1-1039, and by NSF Grant CNS-0429689.

ASIACCS '08, March 18-20, Tokyo, Japan

able to cope with academic protocols.

Here we report results on the formalization and analysis of the Kerberos 5 protocol [34], with and without its public-key extension PKINIT [23], using the prover CryptoVerif [13, 14, 15]. Unlike the previously mentioned tools, CryptoVerif can verify protocols directly in the computational model. We note that CryptoVerif is different from ProVerif [12], a well-established tool which verifies protocols in the symbolic model; CryptoVerif is a next-generation prover. CryptoVerif proofs are presented as sequences of games in a probabilistic process calculus inspired by [25, 26, 27, 32]. Previously, CryptoVerif has only been used to analyze academic protocols [13, 14], so this work provides a test case for the suitability of CryptoVerif for analyzing real-world protocols. Kerberos and its public-key extension PKINIT (used in 'public-key mode' as discussed below) provide a particularly good test case because they incorporate many different design elements: symmetric and asymmetric encryption, digital signatures, and keyed hash functions. Using CryptoVerif's interactive mode, we are able to prove authentication and secrecy properties for Kerberos at the computational level. This suggests that CryptoVerif is capable of analyzing large-scale industrial protocols.

Earlier work on analyzing Kerberos includes: analysis of Kerberos 4 (the previous version of Kerberos, which lacked the complexity of Kerberos 5 with PKINIT) using Isabelle [7]; symbolic proofs by hand of authentication and secrecy in basic Kerberos [17]; the discovery of a flaw in a draft version of PKINIT (which led to a Windows Security Bulletin [31]) and the symbolic proof that the fixed version was secure; by-hand computational proofs of the security of Kerberos using the Backes–Pfitzmann–Waidner (BPW) cryptographic library framework [4]; and the mechanical analysis of the PKINIT fragment (without consideration of the later rounds or basic Kerberos without PKINIT) [24]. Our work here extends these earlier analyses of Kerberos to use a mechanized tool on the full Kerberos protocol, with and without its public-key extension PKINIT; this represents the first computationally sound mechanized proof of a full industrial protocol.

In proving confidentiality properties for Kerberos, we consider not only *key indistinguishability* but also the notion of *key usability* introduced in [21] (and which was proved by hand for Kerberos in [36]). This weaker confidentiality property ensures that a key is still 'good' for use in cryptographic operations, even though it might be distinguishable from a random bitstring. This type of property is important for protocols that, like Kerberos, perform operations with a key during a protocol run but then allow for the future use of this key; because the key has been used, it may be distinguishable from random, but that still may not help an attacker learn any information about messages that are later encrypted under that key. Here we define a notion of *strong key usability* that is less restrictive on the adversary's power than the original definition, and we use CryptoVerif to prove that certain keys in Kerberos satisfy this stronger version of key usability.

Using CryptoVerif we are able to prove authentication properties for Kerberos similar to those previously proved in [4]. However, in contrast to proofs in the BPW model, our proofs using CryptoVerif do currently not allow for adaptive corruption; the set of honest protocol participants is determined beforehand and cannot be reduced during the run of the polynomially many protocol sessions.

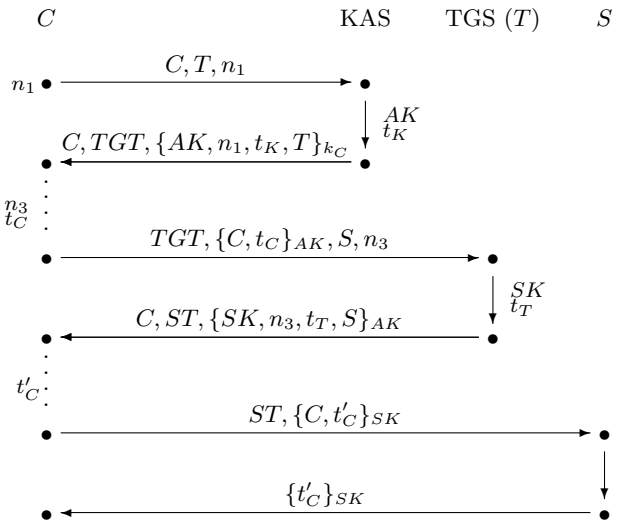

Figure 1: Message flow in basic Kerberos, where $TGT = \{AK, t_K, C\}_{k_T}$ and $ST = \{SK, t_T, C\}_{k_S}$

In Section 2 we review basic and public-key Kerberos 5. Section 3 briefly explains the syntax and semantics of CryptoVerif, including a sample of our formalization of Kerberos, and outlines the authentication and secrecy properties proved by CryptoVerif. Section 4 presents the details of these results and other aspects of our work, while Section 5 provides a summary and surveys areas for future work.

2. KERBEROS AND ITS PROPERTIES

We start with an overview of Kerberos and its public-key extension before discussing (at an informal level) the security properties that we study here. Our description of the protocol here reflects the level of abstraction that we use in our CryptoVerif scripts and which is the level of abstraction that has been used to analyze Kerberos in other frameworks; additional low-level details are described in the protocol specifications [23, 34].

2.1 Basic Kerberos

Kerberos [34] is designed to allow a client to repeatedly authenticate herself to multiple network servers based on a single login. This authentication process can also be used to produce a key shared between the client and end server that can be used for future communications between them. Typically, the human user provides a password at the initial login and a key derived from this password is used in the first round of Kerberos. The credential (or 'ticket') that the client process obtains in this round is then used to obtain other credentials so that the password-derived key is not used again. The client uses this first credential, which might be valid for a single day, to obtain a credential for a particular end server; this second credential might be valid for a few minutes. Finally, the client presents this credential to the end server. Each credential may be used repeatedly as long as it is valid; once the user enters her password, the rest of the protocol can take place in the background.

Figure 1 shows a more detailed view of the message flow in basic Kerberos. The first round, called the *Authentication Service (AS) exchange*, comprises the first two lines of this figure. In it, the client C generates a fresh nonce n_1

and includes it in a message to the Kerberos Authentication Server (KAS) requesting a Ticket Granting Ticket (TGT) for use with the Ticket Granting Server (TGS) named T. The KAS generates a fresh key AK for use between C and T (as well as a timestamp t_K) and sends this key to C. One copy of AK is encrypted under C's long-term key k_C (typically derived from a password), while another is included in the TGT, which is encrypted with a long-term key k_T shared between the KAS and T.

The client then forwards the TGT to T—along with an authenticator encrypted under AK, a fresh nonce n_3, and the name S of an end server—to request a Service Ticket (ST) for S. This message, and the reply from T, form the *Ticket Granting (TG) exchange*. T generates a fresh key SK for use between C and S (as well as a timestamp t_T) and sends this to C. One copy of SK is now encrypted under AK and another is included in the ST, which is encrypted under a long-term key k_S shared between T and S. The TG exchange may be repeated multiple times—to obtain STs for any number of end servers—with a single TGT as long as that ticket is valid.

The last two messages shown in Figure 1 form the *Client/Server (CS) exchange*. In this round of the protocol, C forwards the ST to S along with an authenticator: C's name and a timestamp t'_C encrypted under SK. The last line of the figure shows the optional reply by S, which consists of the timestamp from the authenticator encrypted under SK.

Although not shown in Figure 1, the CS exchange allows C and S to agree on a key for use in future communications between them [34]. The client may send a proposed key in the authenticator she sends to S (encrypted under AK), and S may send a proposed key in the reply to C (encrypted under AK). In our analysis of this key, we assume that exactly one of C and S proposes a key this way.

2.2 The PKINIT Extension

The PKINIT extension [23] to Kerberos replaces the basic AS exchange, allowing the use of a PKI in place of a long-term key shared between the client C and the KAS K. Here we focus exclusively on PKINIT's 'public-key mode,' but PKINIT may also be used in 'Diffie-Hellman (DH) mode' [23]; DH mode, which has recently be studied in [37], appears to have limited implementation, while public-key mode has been implemented for all major operating systems. PKINIT does not change either of the later rounds in the Kerberos protocol. Figure 2 shows the AS exchange when PKINIT is used. In addition to the data sent in the first message of basic Kerberos, C also sends to K her signature over a timestamp t''_C and a second nonce n_2, along with certificates for C's public key. K now generates a key k in addition to AK and t_K as in basic Kerberos. The fresh key k is used in place of k_C to communicate AK to C. k is sent to C encrypted under C's public key pk_C along with certificates for K's public key and a checksum ck taken over C's request; this checksum consists of a keyed hash function (HMAC) using a key derived from k (more precisely, the key is the output of a key derivation function whose input includes k). Furthermore, k and ck are signed by K. (Note that in both of these messages, the signatures $[\_]_{\_}$ are implemented in Kerberos by sending the signed data alongside a signature over the data.)

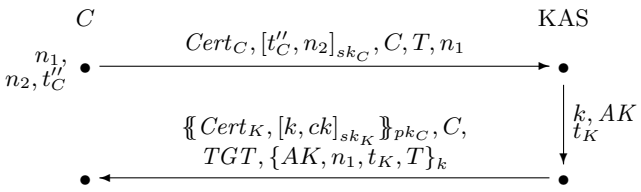

Figure 2: Message flow in the fixed version of PKINIT, where $TGT = \{AK, t_K, C\}_{k_T}$.

2.3 Security Properties

We now informally state security properties of Kerberos concerning entity authentication and key secrecy. In Section 4 we will formalize these security properties in the language of CryptoVerif and present the results in the computational model.

Property 1 (Authentication properties)

a. [Authentication of KAS to client] *If a client receives what appears to be a valid reply from the KAS, then the KAS generated a reply for the client.*

b. [Authentication of request for ST] *If a TGS receives a valid request for a service ticket, then the ticket in the request was generated by the KAS. Furthermore, the authenticator included in the request was generated by the client.*

c. [Authentication of TGS to client] *If a client receives what appears to be a valid reply to a request for a service ticket for server S from a TGS, then the TGS generated a reply for the client.*

d. [Authentication of request to server] *If S receives a valid request, ostensibly from C, containing a service ticket and the session key SK, then some TGS generated the session key SK for C to use with S and also created the service ticket. Furthermore, C created the authenticator.*

e. [Authentication of server to client] *If a client receives a valid reply from server S then this reply was generated by S.*

We consider two different notions of key secrecy. One is the standard notion of cryptographic key secrecy, and the other is the notion of key usability [21] (see below in Section 4.3). We note that the latter notion had not been considered in [4].

Property 2 (Secrecy properties)

a. [Secrecy of AK] *If a client finishes an AS exchange with the KAS, then the authentication key AK is cryptographically secret until the client initiates the TS exchange,* i.e.*, the second round.*

b. [Secrecy of SK] *If a client finishes an TG exchange with a TGS, then the session key SK is cryptographically secret until the client initiates the CS exchange,* i.e.*, the third round.*

c. [Usability of AK] *If a client finishes a TG exchange with a TGS, or if the TGS finishes an exchange with the client, then the authentication key is usable for IND-CCA2-secure encryption.*

d. [Usability of *SK*] *If a client finishes a CS exchange with a server or if the server finishes an CS exchange with the client, then the session key is usable for IND-CCA2-secure encryption.*

3. CRYPTOVERIF

In this section we give a brief overview of CryptoVerif, formalize Kerberos using it, and summarize the authentication and secrecy properties proved by CryptoVerif.

3.1 CryptoVerif Basics

The prover CryptoVerif [11, 13, 14, 15], available at `http://www.cryptoverif.ens.fr`, can directly prove security properties of cryptographic protocols in the computational model. Protocols are formalized using a probabilistic polynomial-time process calculus which is inspired by the pi-calculus and the calculi introduced in [25] and [32]. In this calculus, messages are bitstrings and cryptographic primitives are functions operating on bitstrings. This calculus is illustrated below on a portion of code coming from Kerberos. A more detailed description of the process calculus is given in [11].

The process calculus represents games, and proofs are represented as sequences of games, where the initial game formalizes the protocol for which one wants to prove certain security properties. In a proof sequence, two consecutive games Q and Q' are *observationally equivalent*, meaning that they are computationally indistinguishable for the adversary. CryptoVerif transforms one game into another by applying, *e.g.*, the security definition of a cryptographic primitive or by applying syntactic transformations. In the last game of a proof sequence the desired security properties should be obvious. Given a security parameter η, CryptoVerif proofs are valid for a number of protocol sessions polynomial in η, in the presence of an active adversary.

CryptoVerif operates in two modes: a fully automatic and an interactive mode. The interactive mode, which is best suited for protocols using asymmetric cryptographic primitives, requires a CryptoVerif user to input commands that indicate the main game transformations the tool should perform. CryptoVerif is sound with respect to the security properties it shows in a proof, but properties it cannot prove are not necessarily invalid.

3.2 Modeling Kerberos in CryptoVerif

As an example, we present the client role of the first round of basic Kerberos (AS Exchange) from Figure 1 in the process calculus. (The full CryptoVerif scripts are available at `http://www.cryptoverif.ens.fr/kerberos/`.)

$$
\begin{array}{l}
Q_C = !^{i_C \leq N}\, c_2[i_C](hostT : tgs); \mathsf{new}\ n_1 : nonce; \\
\quad \overline{c_3[i_C]}\langle C, hostT, n_1\rangle; \\
\quad c_4[i_C](=C,\ TGT : bitstring, m_2 : bitstring); \\
\quad \mathsf{let}\ \mathrm{injbot}(\mathrm{concat1}(AK, =n_1, t_K, =hostT)) \\
\qquad = \mathrm{dec}(m_2, k_C)\ \mathsf{in} \\
\quad \mathsf{event}\ fullCK(hostT, n_1,\ TGT, m_2)
\end{array}
$$

Figure 3: CryptoVerif formalization of client's actions in AS exchange.

The replicated process $!^{i_C \leq N} P$ represents N copies of P, available simultaneously, where N is assumed to be polynomial in the security parameter η. These copies are indexed by the integer value $i_C \in [1, N]$. Each copy starts with an input $c_2[i_C](hostT : tgs)$. This input receives a message $hostT$ on channel $c_2[i_C]$. The channel is indexed by i_C, so that, by sending a message on channel $c_2[i]$ for a certain value of i, the adversary can choose which copy of the process receives the message and is then executed. The message $hostT$ is the name of the ticket granting server (TGS) to which the client is going to send his request. The type tgs is a set of bitstrings that contains the representation of all possible names of ticket granting servers. The message is received only if it belongs to this type.

Next, the process chooses a random nonce n_1 uniformly in the type *nonce*, by the construct $\mathsf{new}\ n_1 : nonce$. (A probabilistic Turing machine can choose a number uniformly at random only from sets whose size is a power of 2; to make sure that the choice above is possible, we assume that *nonce* consists of all bitstrings of a certain length.) The process then sends the first message of the protocol $C, hostT, n_1$ on channel $c_3[i_C]$. This message will be received by the adversary; the adversary can do whatever he wants with it, but in order to run a normal session of the protocol, he should send this message to the Kerberos authentication server (KAS).

After sending the message on channel $c_3[i_C]$, the control is returned to the adversary and the process $c_4[i_C](\ldots); \ldots$ is made available. This process waits for the second message of the protocol and will be executed when a message is sent on channel $c_4[i_C]$. The expected message is $C, TGT, m_2 = C, \{AK, t_K, C\}_{k_T}, \{AK, n_1, t_K, hostT\}_{k_C}$. The message received on channel $c_4[i_C]$ then consists of three parts: the client name C, the TGT TGT, and the message $m_2 = \{AK, n_1, hostT\}_{k_C}$. The process checks that the first component of this message is C by using the pattern $=C$; the two other parts are stored in variables.

The process Q_C cannot check the TGT, which is encrypted under a key the client does not have. On the other hand, Q_C can decrypt and check m_2. It decrypts m_2 by $\mathrm{dec}(m_2, k_C)$ and checks that the resulting plaintext matches the expected *nonce* and tgs. If the decryption fails, it returns the special symbol $\perp$. The function injbot is the natural injection from plaintexts to bitstrings and $\perp$, so that, when $\mathrm{injbot}(x) = \mathrm{dec}(m_2, k_C)$, the decryption succeeded and x is the plaintext. Furthermore, the expected plaintext is the concatenation of AK, n_1, t_K, and $hostT$, $\mathrm{concat1}(AK, n_1, t_K, hostT)$. The concatenation function concat1 is assumed to be injective, with inverses computable in polynomial time, so that AK, n_1, t_K, and $hostT$ can be recovered from the plaintext in polynomial time. This assumption is justifiable in view of the Kerberos data structures involved. The let construct in Q_C checks that the plaintext is of the required form, with the already known values of n_1 and $hostT$, and binds the variables AK and t_K to the received values.

When a check fails, the control is returned to the adversary. When all checks succeed, Q_C executes the event $fullCK(hostT, n_1, TGT, m_2)$. Executing this event does not affect the execution of the protocol; it just records that a certain program point is reached with certain values of the variables. Events are used for specifying authentication properties, as explained in Section 3.3. After executing the event, the control is returned to the adversary.

In this calculus, all variables defined under replications are

implicitly arrays, indexed by the indices of these replications. For instance, the variable $hostT$ defined under $!^{i_C \leq N}$ is in fact $hostT[i_C]$, so that each copy of the replicated process stores the value of $hostT$ in a distinct cell of the array. The arrays allow us to keep track of the whole state of the system. In the cryptographic proofs, the arrays used in the calculus of CryptoVerif replace lists often used by cryptographers. As an example of the use of lists, suppose that symmetric encryption satisfies ciphertext integrity (INT-CTXT). This assumption means that, when decryption succeeds, the considered ciphertext has been generated by calling the encryption function with the same secret key (provided the key is not leaked). Then, one usually stores the computed ciphertexts in a list, and upon decryption, one can additionally check that the ciphertext is in the list. In our calculus, the computed ciphertexts are always automatically stored in an array, instead of a list, which avoids having to add explicit list insertion instructions. The calculus provides an array lookup construct, detailed in [11].

3.3 Authentication using CryptoVerif

Authentication in CryptoVerif is modeled by correspondence properties [14]. Events $e(M_1, \ldots, M_m)$ are used in order to record that a certain program point has been reached, with certain values of $M_1, \ldots, M_m$, and the correspondence properties are properties of the form "if some event has been executed, then some other events also have been executed, with overwhelming probability".

More precisely, we distinguish two kinds of correspondences.

- A process Q *satisfies the non-injective correspondence* $\mathsf{event}(e(M_1, \ldots, M_m)) \Rightarrow \bigwedge_{i=1}^{k} \mathsf{event}(e_i(M_{i1}, \ldots, M_{im_i}))$ if and only if, with overwhelming probability, for all values of the variables in $M_1, \ldots, M_m$, if the event $e(M_1, \ldots, M_m)$ has been executed, then the events $e_i(M_{i1}, \ldots, M_{im_i})$ for $i \leq k$ have also been executed for some values of the variables of M_{ij} ($i \leq k$, $j \leq m_i$) not in $M_1, \ldots, M_m$.
- A process Q *satisfies the injective correspondence* $\mathsf{inj\text{-}event}(e(M_1, \ldots, M_m)) \Rightarrow \bigwedge_{i=1}^{k} \mathsf{inj\text{-}event}(e_i(M_{i1}, \ldots, M_{im_i}))$ if and only if, with overwhelming probability, for all values of the variables in $M_1, \ldots, M_m$, for each execution of the event $e(M_1, \ldots, M_m)$, there exist distinct corresponding executions of the events $e_i(M_{i1}, \ldots, M_{im_i})$ for $i \leq k$ for some values of the variables of M_{ij} ($i \leq k$, $j \leq m_i$) not in $M_1, \ldots, M_m$.

(Formal definitions can be found in [14]. CryptoVerif can prove more general correspondences [14], but the correspondences above were sufficient for our study of Kerberos.)

3.4 Secrecy using CryptoVerif

A variable is considered secret when the adversary has no information on it, that is, the adversary cannot distinguish it from a random number. CryptoVerif distinguishes two notions of secrecy.

- A process Q *preserves the one-session secrecy of* x when, with overwhelming probability, the adversary interacting with Q cannot distinguish any element of the array x from a uniformly distributed random number by a single test query. The test query returns either the desired element of x or a freshly generated random number, and the adversary has to distinguish between these two situations. (This notion of secrecy does not guarantee that the random numbers in x are independent.)
- A process Q *preserves the secrecy of* x when, with overwhelming probability, the adversary interacting with Q cannot distinguish the elements of the array x from independent, uniformly distributed random numbers. In this notion of secrecy, the adversary can perform several test queries on the various elements of the array x, which either all return elements of x or all return independent random numbers. This corresponds to the "real-or-random" definition of security [2]. (As shown in [2], this notion is stronger than the more standard approach in which the adversary can perform a single test query and some reveal queries, which always reveal an element of x.)

When the array x contains a single element (that is, x is defined under no replication), the notions of one-session secrecy and of secrecy are equivalent. The one-session secrecy of x is coded in CryptoVerif by the query $\mathsf{secret1}\ x$, while the secrecy of x is coded by $\mathsf{secret}\ x$. The formal definitions of these two notions can be found in [11].

4. RESULTS

We have used CryptoVerif 1.06pl3 to prove secrecy and authentication properties for Kerberos (with and without PKINIT). In the following we will first discuss the assumptions on the cryptographic primitives used in our CryptoVerif proofs, and then present the authentication and secrecy results.

The main challenges we faced in achieving the results below were the following:

- The user needs to know the process calculus well enough to understand how exactly CryptoVerif applies the security of cryptographic primitives and to be able to read the last game of a CryptoVerif proof (which is not trivial and needs some practice). The latter is particularly important for interactive proofs.
- The user must know the underlying cryptography well enough to be able to specify the security of cryptographic primitives through indistinguishable oracles, although many primitives have already been specified in previous examples [11] and the user can copy them from there.

Furthermore, we note that Kerberos is a well-studied protocol and we found the previous work on Kerberos 5 [17, 19, 4] very valuable, as it gave us a good sense for which results we could expect to be verified by CryptoVerif in the computational model. This helped us, in cases in which we initially could not verify an expected property, to narrow down the cause—mostly issues with the underlying cryptography but, in rare cases, also issues with CryptoVerif itself (see also Section 4.5).

4.1 Cryptographic Assumptions

In our analysis, the public-key encryption scheme is assumed to be indistinguishable under adaptive chosen ciphertext attacks (IND-CCA2), and the signature scheme is assumed to be unforgeable under chosen message attacks (UF-CMA). Symmetric encryption is assumed to be indistinguishable under chosen plaintext attacks (IND-CPA) and

to satisfy ciphertext integrity (INT-CTXT). These properties guarantee indistinguishability under adaptive chosen ciphertext attacks (IND-CCA2), as shown in [9]. These assumptions are the same as in [4], and Boldyreva and Kumar showed in [16] that the encryption of the simplified profile of basic Kerberos satisfies these properties for symmetric encryption. They also showed that the general profile encryption is weak, and propose a corrected version of the general profile encryption that satisfies these properties.

The keyed hash function used to compute the checksum in PKINIT is assumed to be a message authentication code, weakly unforgeable under chosen message attacks (UF-CMA), which is in accordance with [8] and which also matches the assumptions in [4]. As it is unwise to use the same key for multiple cryptographic operations, a key derivation function is used to generate multiple keys from a base key; this key derivation function takes as input the base key and publicly known integer called *usage number* [35]. We assume that the key derivation function is a pseudo-random function and use it to derive, from a base key, a key for the message authentication code and another key for the encryption of the message component that includes the authentication key for the client (denoted k in Figure 2). We note that in the specifications [23, 34] a key derivation function is used not only for the key mentioned above but for all symmetric keys (even if they are not used for multiple cryptographic operations); we, however, restrict the use of a key derivation function to the key above. Implementing the use of a key derivation function for all symmetric keys will be part of future work.

For basic Kerberos, we assume that the long-term key k_C shared between the client and the KAS is generated from a random seed, although in practice this key is usually generated from a password and is vulnerable to dictionary attacks [10].

Furthermore, we assume that concatenations of some types of bitstrings (*e.g.*, a key followed by a timestamp followed by a client name) cannot be confused with other such concatenations (*e.g.*, a key followed by a nonce followed by a timestamp followed by a TGS name). The assumptions of this type that we make are justifiable in view of the differences between the various Kerberos data structures.

4.2 Authentication Results

Here we present authentication properties directly proved in the computational model by CryptoVerif 1.06pl3 under the assumptions from Section 4.1.

We formalize Property 1(a) as the following theorem.

Theorem 1 (Authentication of KAS to client) *In basic and public-key Kerberos, for each instance of:*

- *an honest client C completing the AS exchange with KAS K,*
- *in which the client sent the request m_{req} to receive a TGT for the use with honest TGS T,*
- *received what appears to be a valid reply m'_{rep}*

there exists, with overwhelming probability, a distinct corresponding instance of:

- *the KAS completing the AS exchange with C,*
- *in which the KAS received the request m_{req} for a TGT for the use between C and T,*

$$
\begin{aligned}
&!^{i' \leq n''}\, \mathsf{new}\ r : keyseed;\\
&\quad (!^{i \leq n}(x : maxenc) \rightarrow \mathsf{new}\ r' : seed; \mathrm{enc}(x, \mathrm{kgen}(r), r'),\\
&\quad\ !^{i \leq n'}(y : bitstring) \rightarrow \mathrm{dec}(y, \mathrm{kgen}(r)))\\
\approx\ &!^{i' \leq n''}\, \mathsf{new}\ r : keyseed;\\
&\quad (!^{i \leq n}(x : maxenc) \rightarrow \mathsf{new}\ r' : seed;\\
&\qquad \mathsf{let}\ z : bitstring = \mathrm{enc}(x, \mathrm{kgen}'(r), r')\ \mathsf{in}\ z,\\
&\quad\ !^{i \leq n'}(y : bitstring) \rightarrow\\
&\qquad \mathsf{find}\ j \leq n\ \mathsf{suchthat}\ \mathsf{defined}(x[j], z[j]) \wedge z[j] = y\\
&\qquad\quad \mathsf{then}\ \mathrm{injbot}(x[j])\ \mathsf{else}\ \bot
\end{aligned}
\qquad \text{(INT-CTXT)}
$$

$$
\begin{aligned}
&!^{i' \leq n'}\, \mathsf{new}\ r : keyseed; !^{i \leq n}(x : maxenc) \rightarrow\\
&\qquad \mathsf{new}\ r' : seed; \mathrm{enc}(x, \mathrm{kgen}'(r), r')\\
\approx\ &!^{i' \leq n'}\, \mathsf{new}\ r : keyseed; !^{i \leq n}(x : maxenc) \rightarrow\\
&\qquad \mathsf{new}\ r' : seed; \mathrm{enc}'(\mathrm{Z}(x), \mathrm{kgen}'(r), r')
\end{aligned}
\qquad \text{(IND-CPA)}
$$

Figure 4: Definition of INT-CTXT and IND-CPA symmetric encryption in CryptoVerif

- *sent reply m_{rep}, where all message components of m_{rep}, except the TGT, are equal to the corresponding components in m'_{rep}.*

Proof. *Basic Kerberos case*: when the client process completes its participation in an AS exchange, it executes an event $fullCK(hostT, n_1, TGT, m_2)$ that contains the name $hostT$ of the TGS and the nonce n_1 from the client's first message, and the reply from K in TGT and m_2, where m_2 is the reply component encrypted under C's long-term shared key and TGT is assumed to be a ticket granting ticket. When the KAS process completes its participation in an AS exchange, it executes an event $fullKC(hostY, hostW, n'_1, TGT', e4)$ that contains the name $hostY$ of the client, the name $hostW$ of the TGS and the nonce n'_1 listed in the request. Furthermore, it contains ticket granting ticket TGT' generated by the KAS containing the authentication key AK', and the reply component $e4$ encrypted under $hostY$'s long-term shared key. CryptoVerif can then automatically prove the query: $\mathsf{inj\text{-}event}(fullCK(T, n, x, y)) \Rightarrow \mathsf{inj\text{-}event}(fullKC(C, T, n, z, y))$.

The proof done by CryptoVerif consists essentially in applying, after some minor simplifications, the security assumptions on symmetric key encryption for each key k_S, k_T, and k_C. In more detail, CryptoVerif performs the following transformations:

- It removes assignments on k_S, that is, it replaces k_S with its value $\mathrm{kgen}(rKs)$: k_S is generated from a random seed rKs by the key generation algorithm kgen.
- The variable *Pkey* is assigned at two places in the game, either with the key $k_S = \mathrm{kgen}(rKs)$, when T and S are honest, or with a key coming from the adversary. CryptoVerif renames these two assignments to *Pkey* to distinct names *Pkey_88* and *Pkey_87* respectively, which leads to distinguishing two cases, depending on whether *Pkey* is shared between honest T and S or not.
- CryptoVerif removes assignments on *Pkey_88*, that is,

it replaces *Pkey_88* with its value $\mathrm{kgen}(rKs)$.

- CryptoVerif applies the INT-CTXT property of the symmetric encryption on the key $k_S = \mathrm{kgen}(rKs)$. The INT-CTXT property is represented in CryptoVerif by the equivalence (INT-CTXT) of Figure 4. In this equivalence, the left-hand side chooses a random seed r and provides two oracles: the first one encrypts its argument x under key $\mathrm{kgen}(r)$ generated from r, using fresh coins r'; the second one decrypts its argument y with key $\mathrm{kgen}(r)$. The right-hand side provides two corresponding oracles: the first one still encrypts under $\mathrm{kgen}(r)$, but additionally stores the ciphertext in the variable z. This variable is implicitly an array indexed by the number of the call to the encryption oracle. The second oracle, instead of decrypting its argument y, looks for y in the array z that contains all computed ciphertexts. When y is found in this array, that is, there exists j such that $z[j] = y$, the oracle returns the corresponding plaintext $x[j]$, injected by $\mathrm{i}_\perp$ into the set of bitstrings union the special symbol $\perp$. When no such y is found, the oracle returns $\perp$, meaning that decryption failed. Ciphertext integrity implies that the left-hand side and the right-hand side are indistinguishable for an attacker: with overwhelming probability, the attacker is unable to produce a valid ciphertext without calling the encryption oracle, so the valid ciphertexts are those stored in z and decryption succeeds if and only if the ciphertext is found in the array z.

  Using this equivalence, CryptoVerif can transform a game by replacing the left-hand side of the equivalence with its right-hand side as follows: provided rKs is a random number used only in terms of the form $\mathrm{enc}(M, \mathrm{kgen}(rKs), r')$ for a fresh random number r' and $\mathrm{dec}(M', \mathrm{kgen}(rKs))$, it replaces occurrences of $\mathrm{enc}(M, \mathrm{kgen}(rKs), r')$ with $\mathsf{let}\ x = M\ \mathsf{in}\ \mathsf{let}\ z = \mathrm{enc}(x, \mathrm{kgen}'(rKs), r')\ \mathsf{in}\ z$ for some new variables x and z, and $\mathrm{dec}(M', \mathrm{kgen}(rKs))$ with a lookup that looks for M' in all variables z and returns the corresponding value of $\mathrm{injbot}(x)$ in case of success and $\perp$ in case of failure. (The previous game transformations were useful in order to make terms of the form $\mathrm{enc}(M, \mathrm{kgen}(rKs), r')$ and $\mathrm{dec}(M', \mathrm{kgen}(rKs))$ appear.)

  As a final technical detail, the right-hand side of the equivalence uses the function symbol kgen' instead of kgen: this prevents repeated application of the game transformation since after transformation, terms of the form $\mathrm{enc}(x, \mathrm{kgen}(r), r')$ are no longer found.
- After each cryptographic transformation, the game is simplified. CryptoVerif uses essentially equational reasoning to replace terms with simpler terms and tries to determine the result of tests, thus removes branches that cannot be executed. In particular, if the initial game contained a statement of the form $\mathsf{let}\ \mathrm{injbot}(\mathrm{concat2}(SK, tt, hostC)) = \mathrm{dec}(M, k_S)\ \mathsf{in}\ \ldots$, the decryption has been replaced by a lookup that returns plaintexts, so simplification can then select only the branche(s) of the lookup that return a value that can be equal to $\mathrm{i}_\perp(\mathrm{concat2}(SK, tt, hostC))$.

  The simplification also removes collisions between random numbers: for instance, when a test requires that two independent random nonces are equal, this test fails with overwhelming probability.
- CryptoVerif applies the IND-CPA property of the symmetric encryption on the key $k_S = \mathrm{kgen}(rKs)$. The IND-CPA property is represented in CryptoVerif by the equivalence (IND-CPA) of Figure 4. This equivalence expresses that the oracle that encrypts x is indistinguishable from an oracle that encrypts $Z(x)$, where $Z(x)$ represents a bitstring of zeroes, of the same length as x. This property is implied by IND-CPA.

  CryptoVerif will then replace terms $\mathrm{enc}(M, \mathrm{kgen}'(rKs), r')$ with $\mathrm{enc}'(Z(M), \mathrm{kgen}'(rKs), r')$, provided rKs is a random number occuring only in such terms and r' is a fresh random number.

  The right-hand side uses enc' instead of enc to prevent repeated application of the game transformation.
- After applying this transformation, the game is simplified. In particular, terms of the form $Z(M)$ are simplified to constants when the length of M is constant, which removes the dependency on M.

CryptoVerif then applies similar steps for keys k_T and k_C. After applying the INT-CTXT property for k_C, it succeeds proving the desired correspondence.

The probability $P(t)$ that an attacker running in time t breaks the correspondence $\mathsf{inj\text{-}event}(fullCK(T, n, x, y)) \Rightarrow \mathsf{inj\text{-}event}(fullKC(C, T, n, z, y))$ is bounded by CryptoVerif by $P(t) \leq \frac{N^2}{2\,|nonce|} + \frac{N}{|nonce|} + P_{\text{INT-CTXT}}(t + t_{C1}, N, N) + P_{\text{IND-CPA}}(t + t_{C2}, N) + P_{\text{INT-CTXT}}(t + t_{C3}, N, N) + P_{\text{IND-CPA}}(t + t_{C4}, N) + P_{\text{INT-CTXT}}(t + t_{C5}, N, N)$ where N is the maximum number of sessions of the protocol participants, $|nonce|$ is the cardinal of the set of nonces, $P_{\text{INT-CTXT}}(t, n, n')$ is the probability that an attacker running in time t breaks the INT-CTXT equivalence with at most n calls to the encryption oracle and n' calls to the decryption oracle (for one encryption key), $P_{\text{INT-CPA}}(t, n)$ is the probability that an attacker running in time t breaks the IND-CPA equivalence with at most n calls to the encryption oracle, and t_{C1}, t_{C2}, t_{C3}, t_{C4}, and t_{C5} are bounds on the running time of the part of the transformed games not included in the INT-CTXT or IND-CPA equivalence, which are therefore considered as part of the attacker against the INT-CTXT or IND-CPA equivalence. The first two terms of $P(t)$ come from elimination of collisions between nonces, while the other terms come from cryptographic transformations using the INT-CTXT or IND-CPA properties of encryption for keys k_S, k_T, and k_C. (Only the INT-CTXT property is used for k_C.)

Note that, if CryptoVerif applied the INT-CTXT property of encryption on key k_C first, it would prove the query without needing the security of encryption for k_S and k_T, and with a tighter bound on the probability $P(t) \leq \frac{N^2}{2\,|nonce|} + \frac{N}{|nonce|} + P_{\text{INT-CTXT}}(t + t'_{C1}, N, N)$. This proof can be obtained by manually giving to CryptoVerif the instruction `crypto dec rKc`, which instructs it to apply the INT-CTXT equivalence (the only equivalence that has the dec symbol in its left-hand side) to the key generated from rKc. CryptoVerif automatically guesses the few syntactic transformations that it has to do before applying this equivalence.

Public-key Kerberos case: when the client process completes its participation in PKINIT, it executes an event $fullCK(hostZ, hostT, n_1, m21, TGT, m24)$ that contains the name $hostZ$ of the KAS, the name $hostT$ of the TGS and the unsigned nonce n_1 from the client's first message. Fur-

thermore, it contains the reply from *hostZ* in *m21*, TGT, and *m24*, where *m21* is the part of the reply encrypted under C's public key and *m24* is the reply component that contains the authentication key (AK). When the KAS process completes its participation in PKINIT, it executes an event $fullKC(hostY, hostW, n_1', e21, TGT', e24)$ that contains the name *hostY* of the client, the name *hostW* of the TGS and the unsigned nonce n_1' listed in the request. Furthermore, it contains the public-key encryption component *e21* of K's reply (under *hostY*'s public key), the ticket granting ticket TGT' generated by the KAS containing the authentication key (AK'), and the reply component *e24* that contains the authentication key. CryptoVerif can then prove the query: $\mathsf{inj\text{-}event}(fullCK(K, T, n, w, x, y)) \Rightarrow \mathsf{inj\text{-}event}(fullKC(C, T, n, w, z, y))$.

We note that the proof for the public-key Kerberos case is an interactive proof which uses the following commands: `crypto sign rkCA`, `crypto sign rkCs`, `crypto penc rkC`, `crypto sign rkKs`, `crypto keyderivation`, `simplify`, `crypto keyderivation`, `simplify`, and `auto`. (The commands are given in `typeface` and separated by commas; the i^{th} command is given on the i^{th} occasion that CryptoVerif requests user input.) The command `crypto sign rkCA` instructs CryptoVerif to transform the game using the security of the signature for the keys generated from the random number `rkCA`. In this case, `rkCA` was used to generate the signature key of the certificate authority who signed the certificates of the client and the KAS. Similarly, `rkCs` and `rkKs` generated the signature keys of the client and the KAS, respectively. The command `crypto penc rkC` instructs CryptoVerif to apply the security for the client's public-key encryption key generated by the random number `rkC` to transform the game. The command `crypto keyderivation` instructs CryptoVerif to make a game transformation by applying the security of the key derivation function (*i.e.*, pseudo randomness), and the command `simplify` instructs CryptoVerif to apply the build-in simplification algorithm to the current game. The command `auto` instructs CryptoVerif to continue the proof automatically, using its built-in proof strategy. □

Theorems 2–5 below can be proved in a similar way. We detail the proof of Theorem 4 as a second example, and omit the other proofs because of length constraints.

We formalize Property 1(b) as the following theorem.

Theorem 2 (Authentication of request for ST) *In basic and public-key Kerberos, if there is an instance of:*

- *an honest TGS T receiving a valid request m_{req} for a service ticket from an honest client C*

then, with overwhelming probability, there is an instance of:

- *the KAS completing an AS exchange with C,*
- *in which the KAS generated the ticket granting ticket for the use between C and T, which equals the one contained in m_{req},*

and an instance of:

- *the client C requesting a service ticket from T,*
- *in which C sent the authenticator, which equals the one contained in m_{req}.*

We formalize Property 1(c) as the following theorem.

Theorem 3 (Authentication of TGS to client) *In basic and public-key Kerberos, for each instance of:*

- *an honest client C completing a TG exchange with an honest TGS T*
- *in which the client sent the request m_{req} to receive a service ticket ST for the use with honest server S,*
- *received what appears to be a valid reply m'_{rep}*

there exist, with overwhelming probability, a distinct corresponding instance of:

- *the TGS T completing a TG exchange with client C*
- *in which the TGS received the request m'_{req} for a ST for the use between C and S,*
- *sent reply m_{rep}, where the message component of m_{rep} encrypted under the authentication key, which contains the service key SK, is equal to the corresponding component in m'_{rep}.*

We formalize Property 1(d) as the following theorem.

Theorem 4 (Authentication of request to server) *In basic and public-key Kerberos, if there is an instance of:*

- *an honest server S receiving a valid request m_{req} from an honest client C*

then, with overwhelming probability, there is an instance of:

- *the TGS completing a TG exchange with C,*
- *in which the TGS generated a service ticket contained for the use between C and S, which is equal to the one in m_{req}*

and an instance of:

- *the client C sending an authentication requesting a service from S,*
- *in which C sent an authenticator, which is equal to the one contained in m_{req}.*

Proof. *Basic Kerberos case*: when the server process validates a received request in a CS exchange, it executes an event $partSC(hostC, m14, m15)$ that contains the name *hostC* of the client contained in the ST, the ST itself in *m14*, and the matching authenticator in *m15*. When the TGS process completes its participation in a TG exchange, it executes an event $fullTC(hostY, hostW, n', m8, m9, ST', e11)$ that contains the name *hostY* of the client, the name *hostW* of the server, the nonce n', the TGT *m8*, and the authenticator *m9*, which were all listed in the request m'_{req}. Furthermore, the event contains the service ticket ST' generated by the TGS containing the service key SK', and the message component *e11* of the reply that is encrypted under the authentication key. When the client process sends a request to a server, it executes an event $partCS(hostX, hostY, ST, e12)$ that contains the name *hostX* of the TGS from which the client requested a service ticket, the name *hostY* of the server, the alleged ST in ST, and the authenticator sent by C in *e12* containing C's name and a timestamp encrypted under the service key SK, which C received in the same TS reply as TGT. CryptoVerif can then automatically prove the query: $\mathsf{event}(partSC(C, z, y)) \Rightarrow \mathsf{event}(partCS(S, T, x, y)) \wedge \mathsf{event}(fullTC(C, S, n, v, v', z, w))$.

Public-key Kerberos case: As the CS exchange in public-key Kerberos does not differ from the CS exchange in

basic Kerberos V5, the events $partSC(hostC, m14, m15)$, $partCS(hostX, hostY, ST, e12)$, and $fullTC(hostY, hostW, n', m8, m9, ST', e11)$ are just as the ones described above in the basic Kerberos case. CryptoVerif can then prove the same query as above in basic Kerberos case using the same commands as in the public-key Kerberos case from Theorem 1. □

We formalize Property 1(e) as the following theorem.

Theorem 5 (Authentication of server to client) *In basic and public-key Kerberos, if there is an instance of:*

- *an honest client C completing a CS exchange with an honest server S*
- *in which the client sent the request m_{req},*
- *received a valid reply m_{rep}*

then, with overwhelming probability, there is an instance of

- *the server S completing a CS exchange with client C*
- *in which the TGS received the request m'_{req}*
- *sent the reply m_{rep}.*

We note that the injectivity of the correspondences in Theorems 1 and 3 stems from their challenge-response character; *i.e.*, a fresh nonce is sent and subsequently received. The correspondences in Theorems 2 and 4 are non-injective because the 3rd and 5th messages of Kerberos, respectively, can be replayed. In practice, however, the server should use an anti-replay cache in order to prevent the replay of the 5th message [34]; we do not yet include this cache in our model, but doing so in the future may allow us to show that each instance of the server corresponds to a distinct instance of the client in Theorem 4. The reason for the non-injectivity of the correspondence in Theorem 5, however, is a little different and has to do with how we model timestamps in CryptoVerif. If the client C sends two requests to the server S with the same timestamp t'_C, then the adversary can prevent the second request from reaching S and replay S's reply to the first request as reply for the second request. In this case, two sessions of the client correspond to a single session of the server, so the correspondence is non-injective. Our model in CryptoVerif allows the timestamps of several requests to be equal with non-negligible probability, as in the above scenario. However, this is rather unlikely to happen in the real world, since the timestamps have a 1 μs resolution. If we treat timestamps as nonces, which can be equal only with negligible probability, then the correspondence of Theorem 5 can be shown to be injective (but timestamps are then considered as unguessable).

Remark 1 CryptoVerif can prove all correspondences for basic Kerberos mentioned in Theorems 1–5 simultaneously, using a single sequence of games, and likewise it can prove the correspondences for public-key Kerberos simultaneously, using the same interactive commands.

4.3 Key Secrecy Results

In the following we present the key secrecy results we proved in the computational model using CryptoVerif 1.06pl3 under the assumptions from Section 4.1. First we will discuss key indistinguishability results and then we will discuss key secrecy results with respect to the notion of *key usability*, introduced in [21] and generalized here.

4.3.1 Key Indistinguishability

The key secrecy results in this section are proved with respect to the *real-or-random* definition of security, which is a stronger notion than the standard notion from the literature [2]. We note that, as discussed in [4], the authentication keys and the service keys in Kerberos become distinguishable from random as soon as they are used for encryption during the protocol and the resulting ciphertext is broadcasted on the network, *i.e.*, right after the second and third round respectively, since they are used for encryption of a partially known message (namely, the client's name and a timestamp).

We formalize Property 2(a) as the following theorem. We omit its proof and detail only the proof of the more important result on secrecy of SK (Theorem 7 below), which is similar.

Theorem 6 (Secrecy of AK) *Let $Q_{K5_{1R}}$ be the game in the process calculus formalizing solely the AS exchange of basic Kerberos and let Q_{PKINIT} be the game formalizing the public-key mode of PKINIT. Furthermore, let keyAK denote in $Q_{K5_{1R}}$ and in Q_{PKINIT}, respectively, the authentication key received by an honest client from the KAS and generated by the KAS for the use between the client and an honest TGS. Then $Q_{K5_{1R}}$ and Q_{PKINIT} preserve the secrecy of keyAK.*

Remark 2 For the flawed draft version of PKINIT, CryptoVerif was not able to produce a positive proof of either the secrecy of the key AK or the authentication of K to C. In fact, neither property holds for the flawed protocol, due to a known attack [19].

We formalize Property 2(b) as the following theorem.

Theorem 7 (Secrecy of SK) *Let $Q_{K5_{2R}}$ be the game in the process calculus formalizing the AS and the TG exchange (*i.e.*, the first two rounds) of basic Kerberos and let $Q_{PK_{2R}}$ be the game formalizing the first two rounds of public-key Kerberos. Furthermore, let keySK denote in $Q_{K5_{2R}}$ and in $Q_{PK_{2R}}$, respectively, the service key received by an honest client from an honest TGS and generated by the TGS for the use between the client and an honest server. Then $Q_{K5_{2R}}$ and $Q_{PK_{2R}}$ preserve the secrecy of keySK.*

PROOF. *For both basic Kerberos and public-key Kerberos*: when the client process completes its participation in a TG exchange with an honest TGS it stores the session key SK in $keySK$. CryptoVerif can then prove the query: secret $keySK$, where in the public-key Kerberos case, the same commands as in the public-key Kerberos case from Theorem 1 are used. □

We note that cryptographic secrecy, *i.e.*, indistinguishability from random, which follows for the keys AK and SK from Theorems 6 and 7, respectively, does not hold any longer once AK is used in a TS request or SK is used in a CS request, as shown in [4]. This is due to the fact that AK and SK are used to encrypt the authenticators in the TG and CS exchange, respectively, which contain a partially known plaintext; namely the client name and a timestamp which was generated during a bounded time period that is typically known to the adversary. If an adversary tries to distinguish either AK or SK from random keys, he

just needs to attempt to decrypt the appropriate authenticator and makes his guess dependent on whether the adversary sees the client's name and a timestamp generated in the bounded time period or not. This gives the adversary an overwhelming advantage of guessing correctly. However, Kerberos allows for the generation of an optional sub-session key [34], which is intended for the encryption of subsequent communication (instead of the session key). This optional sub-session key may be generated by either the client or the server in the CS exchange and included in the message which the client or the server send to each other encrypted under the session key. In [4] it was noted that the optional sub-session key satisfies the notion of cryptographic key secrecy, independent of whether it is generated by the client or the server.

Theorem 8 (Secrecy of Optional Sub-Session Key) *Let $Q_{K5}^{Opt,C}$ and $Q_{PK}^{Opt,C}$ be the games in the process calculus formalizing basic Kerberos and public-key Kerberos, where in both cases an optional subsession key is generated by the client. And let $Q_{K5}^{Opt,S}$ and $Q_{PK}^{Opt,S}$ be the games in the process calculus formalizing basic Kerberos and public-key Kerberos, when an optional subsession key is generated by the server. If OPkeyC and OPkeyS denote in all cases the sub-session keys an honest client and an honest server, respectively, possess after having communicated via a Kerberos session involving an honest TGS, then*

- *$Q_{K5}^{Opt,C}$ and $Q_{PK}^{Opt,C}$ preserve the secrecy of OPkeyC and the one-session secrecy of OPkeyS.*
- *$Q_{K5}^{Opt,S}$ and $Q_{PK}^{Opt,S}$ preserve the secrecy of OPkeyS and the one-session secrecy of OPkeyC.*

PROOF. *For both basic Kerberos and public-key Kerberos*: when the client process completes its participation in a CS exchange with an honest server and involving an honest TGS it stores the optional sub-session key in *OPkeyC*, and, likewise, the server process stores the optional sub-session key in *OPkeyS*. If the optional sub-session key is generated by the server, then CryptoVerif can prove the queries: secret1 *OPkeyC* and secret *OPkeyS*, if the commands in the public-key Kerberos case are the same as in the proof of Theorem 1. The appropriate queries are proved by CryptoVerif if the optional sub-session key is generated by the client. □

In order to understand why CryptoVerif can in some instances only prove one-session secrecy but not secrecy, we distinguish the cases in which a server receives an optional sub-session key generated by the client from the cases in which the client receives a sub-session key generated by a server. In the first case, an adversary can force the server to accept the same sub-session key in multiple sessions that use the same session key SK, by replaying the 5th message of Kerberos. This replay allows an adversary to distinguish these sub-session keys from independent random keys. However, in practice, this replay should be prevented by an anti-replay cache of the server [34], which is not included in our model. The second case stems, again, from the fact that our CryptoVerif model allows two timestamps to be equal with a non-neglible probability (see discussion at the end of Section 4.2). This makes it possible for an adversary to launch a similar attack as above by replaying the response from a server in multiple sessions that use the same key SK and the same timestamp t'_C. If we treat timestamps as nonces so that two timestamps can be equal only with a negligible probability, then CryptoVerif can prove secrecy of the sub-session in the second case.

4.3.2 Key Usability

Weaker than key indistinguishability, the notion of *key usability* [21] aims to capture whether an exchanged key, although possibly not indistinguishable from random, is still "good" to be used subsequently for certain cryptographic operations, *e.g.*, IND-CCA secure encryption. An exchanged key, which is indistinguishable from random, can be used just as a freshly generated key for any cryptographic operations. This notion, however, could sometimes be considered as a too strong since, *e.g.*, keys that are used for encryption of a partially known payload during a key exchange protocol, as is the case in Kerberos, involuntarily become distinguishable. Nonetheless, a dinstinguishable key may still be *usable* and leave an adversary with an at most negligible advantage at winning, *e.g.*, an IND-CCA attack game.

Paralleling the definition of key indistinguishability, the definition of key usability by Datta et al. [21] involves a two-phase attacker $\mathcal{A} = (\mathcal{A}_e, \mathcal{A}_c)$. Informally, given a key exchange protocol Σ and a class of applications S, in the *key exchange phase*, honest parties first run (multiple) sessions of the protocol over a network that is controlled by $\mathcal{A}_e$. Afterwards the attacker $\mathcal{A}_e$ chooses a session and hands the session id together with the information she collected over to $\mathcal{A}_c$. Now the *challenge phase* begins where $\mathcal{A}_c$ is trying to win an attack game against a scheme $\Pi \in S$ which uses keys from the session previously picked by $\mathcal{A}_e$. The syntax of the process calculus used by CryptoVerif does not allow us to formalize a sequence consisting of an exchange phase followed by an challenge phase, nor does it allow us to directly formalize a two-phase attacker who picks a session ID and its key to play the attack game against. Therefore we use an 'auxiliary construction' to prove key usability results for Kerberos using CryptoVerif, which in fact enables us to prove a stronger version of key usability in the case of Kerberos, as we describe in the following, and which, therefore, may contribute to future discussions on the notion of key usability. Our construction involves two aspects that address the syntactical obstacles mentioned above: Firstly, the syntax of the process calculus used by CryptoVerif forces us to let the processes formalizing the exchange phase and the challenge phase run in parallel, *i.e.*, an attacker playing, *e.g.*, an IND-CCA2 game against a symmetric encryption scheme which uses the session key SK, is still able to interact with Kerberos protocol sessions and could utilize these protocol sessions in order to win the IND-CCA2 attack game. However, we make some restriction in definition 1 below which implies, for instance, that if the adversary is trying to win an IND-CCA2 attack game against the symmetric encryption scheme under the cryptographic assumptions in Section 4.1 using the session key SK, then we do not allow the adversary to send any output of the encryption oracle to sessions of the honest protocol principals that are carrying out the CS exchange (*i.e.*, the third round). Secondly, instead of letting the adversary choose the session ID and the key for the attack game, the key is drawn at random from the polynomially many sessions and keys.

Definition 1 (Strong Key Usability) Let $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D}) \in S$ be a symmetric encryption scheme, $b \in \{0, 1\}$,

Σ a key exchange protocol, and $\mathcal{A}$ an adversary. We consider the following experiment $\mathbf{Exp}^{*b}_{\mathcal{A},\Sigma,\Pi}(\eta)$:

- First, $\mathcal{A}$ is given the security parameter η and $\mathcal{A}$ can interact, as an active adversary, with polynomially many protocol sessions of Σ.
- At some point, at the request of $\mathcal{A}$, a session identifier *sid* is drawn at random and $\mathcal{A}$ is given access to a left-right encryption oracle $\mathcal{E}_k(LR(.,.,b))$ and an decryption oracle $\mathcal{D}_k(.)$ both keyed with a key k locally output in session *sid*.
- Adversary $\mathcal{A}$ plays a variant of an IND-CCA2 game
  - where $\mathcal{A}$ submits same-length message pairs (m_0, m_1) to $\mathcal{E}_k(LR(.,.,b))$, which returns $\mathcal{E}_k(m_b)$,
  - $\mathcal{A}$ never queries $\mathcal{D}_k(.)$ on a ciphertext output by $\mathcal{E}_k(LR(.,.,b))$,
  - and $\mathcal{A}$ may interact with uncompleted protocol sessions,
  - all sessions of the protocol do not accept ciphertexts output by the encryption oracle when they reach a point of the protocol at which at least one session expects to receive a message encrypted under the key k.
- At some point, $\mathcal{A}$ outputs a guess bit d, which is also the output of the experiment.

We define the advantage of an adversary $\mathcal{A}$ by $\mathbf{Adv}^{*ke}_{\mathcal{A},\Sigma,\Pi} = |\Pr(\mathbf{Exp}^{*1}_{\mathcal{A},\Sigma,\Pi}(\eta) = 1) - \Pr(\mathbf{Exp}^{*0}_{\mathcal{A},\Sigma,\Pi}(\eta) = 1)|$ and say that keys exchanged through protocol Σ are strongly usable for schemes in S if for all $\Pi \in S$ and any probabilistic, polynomial-time adversary $\mathcal{A}$, the advantage $\mathbf{Adv}^{*ke}_{\mathcal{A},\Sigma,\Pi}$ is negligible.

It is clear that allowing the adversary to interact with protocol sessions during the attack game gives the adversary more power compared to a two-phase attacker as in [21]. Not letting the adversary pick the session ID (which corresponds to a replication index in CryptoVerif's process calculus), on the other hand, restricts the adversary's capabilities. However, since the number of sessions is polynomial (in the security parameter), a non-negligible advantage of winning an attack game for a two-phase adversary as in [21] implies a non-negligible advantage for the adversary we described above.

Furthermore, an attacker in [21] may be more restricted than necessary in order to model many realistic settings. For instance, if a key k is exchanged through protocol Σ_1 to be used in an application protocol Σ_2, then usability of k with respect to the definition in [21] guarantees k to be good for, say, encryption in Σ_2 under the condition that all users on the network stop running protocol Σ_1, which is generally not very realistic. On the other hand, if one requires that messages encrypted under k during a run of Σ_1 differ syntactically from messages encrypted in Σ_2 then messages encrypted under k in Σ_2 will be rejected from participants of Σ_1. Therefore a restriction on the adversary like the one in definition 1 could be realized and k can be securely used in Σ_2 if it satisfies strong key usability. This example suggests yet another definition of key usability; one which comes with a composition theorem for protocols Σ_1 and Σ_2. We intend to explore in subsequent work such a variant definition of key usability and how one could utilize CryptoVerif to prove that an exchanged key satisfies that notion.

We formalize Property 2(c) as the following theorem. We omit its proof and detail only the proof of the more important result on usability of SK (Theorem 11 below), which is similar.

Theorem 9 (Usability of AK) *Let $Q^{AK,use}_{\Sigma,X}$ be the game in the process calculus formalizing the experiment $\mathbf{Exp}^{*b1}_{\mathcal{A},\Sigma,\Pi}(\eta)$, where Σ is basic or public-key Kerberos involving client C, TGS T, and KAS K, Π is the symmetric encryption scheme of Kerberos, and the left-right oracle uses an authentication key AK that was locally output after a completed process of $X \in \{C, T, K\}$. If C, T, and K are honest, then $Q^{AK,use}_{\Sigma,X}$ preserves the secrecy of b1.*

Corollary 10 *Basic and public-key Kerberos satisfy IND-CCA2 (strong) key usability for the authentication key AK, for the symmetric encryption scheme of Kerberos.*

We formalize Property 2(d) as the following theorem.

Theorem 11 (Usability of SK) *Let $Q^{SK,use}_{\Sigma,X}$ be the game in the process calculus formalizing the experiment $\mathbf{Exp}^{*b1}_{\mathcal{A},\Sigma,\Pi}(\eta)$, where Σ is basic or public-key Kerberos involving client C, KAS K, TGS T, and server S, Π is the symmetric encryption scheme of Kerberos, and the left-right oracle uses an authentication key SK that was locally output after a completed process of $X \in \{C, S, T\}$. If C, K, T, and S are honest, then $Q^{SK,use}_{\Sigma,X}$ preserves the secrecy of b1.*

Corollary 12 *Basic and public-key Kerberos satisfy IND-CCA2 (strong) key usability for the service key SK, for the symmetric encryption scheme of Kerberos.*

Proof of Theorem 11. *Basic Kerberos case*: In the case $X = C$, the client process completes its participation in a CS exchange involving an honest TGS, it stores the session key *SK* in *keyCSK*. From these keys one is drawn at random and passed to the encryption oracle and decryption oracle. For the boolean *b1* used by the encryption oracle, we can, using CryptoVerif, prove the query: secret *b1*. This proof requires the user to inspect the last game, which CryptoVerif reaches upon the command `auto`, in order to verify that terms that are dependent on *b1* and which may help an adversary in guessing *b1* occur only in find branches that are never executed. The case $X = T$ is similar, where the session key *SK* is stored in *keyTSK* after the TGS sent the TS_reply. And an analogous result holds for $X = S$, where the proof requires the following commands before the manual inspection of the last game: `auto`, `SArename SK_33`, `simplify`, and `auto` (formatted and entered as described above). The command `SArename SK_33` is used when the variable `SK_33` is defined several times in the game. It instructs CryptoVerif to rename each definition of this variable to a different name, which subsequently allows to distinguish cases depending on the program point at which the variable has been defined.

Public-key Kerberos case: analogously to the basic Kerberos case, the secrecy of the bit *b1* can be concluded by inspecting the last game that CryptoVerif reaches after a sequence of commands. If $X = C$ or $X = T$, the interactive commands are just the ones from the public-key Kerberos case of Theorem 1. If $X = S$, the secrecy of *b1* can be concluded after the sequence of commands: `crypto sign rkCA`, `crypto sign rkCs`, `crypto penc rkC`, `crypto sign rkKs`, `crypto keyderivation`, `simplify`, `crypto keyderivation`, `simplify`, `auto`, `SArename SK_55`, `simplify`, and `auto` (formatted and entered as described above). □

4.4 Varying the Strength of Cryptography

We observe that the symbolic proofs of security for Kerberos in, *e.g.*, [17] do not rely on the secrecy of the encrypted data within the authenticators ($\{C, t_C\}_{AK}$ and $\{C, t'_C\}_{SK}$) sent by the client to the TGS and end server. CryptoVerif is also able to prove security properties for Kerberos without relying on the secrecy of the authenticator data. In particular, we can modify CryptoVerif scripts so that the client sends a second, *unencrypted* copy of the authenticator contents alongside the authenticator and CryptoVerif can still prove security properties for Kerberos. For the case that the client sends a subsession key in the CS exchange authenticator, we make this modification only in the TG exchange; if the server sends the subsession key (but not the client), then we may make this modification in both the TG and CS exchanges. Using CryptoVerif, we can then prove the following theorem about authentication and secrecy when the authenticator contents are leaked as just described.

Theorem 13 *If*

- *the client sends the contents of the authenticator, unencrypted, along with the encrypted authenticator in both the TG and CS exchanges when she does not include a subsession key in the authenticator for the CS exchange; or*
- *the client sends the contents of the authenticator, unencrypted, along with the encrypted authenticator in the TG exchange only when she includes a subsession key in the authenticator for the CS exchange*

then Theorems 1–5 and 8 hold for both basic and public-key Kerberos.

PROOF. If we modify the CryptoVerif scripts to expose the authenticator contents as described, CryptoVerif proves the queries needed for proving Theorems 1–5 and 8; in the case of public-key Kerberos, the interactive commands are the same as before. □

Similar results might be achieved by suitably relaxing the assumptions about the encryption function used for the authenticators. That, and studies of other ways in which the cryptographic assumptions can be weakened without compromising the protocol, remains a topic of ongoing work.

4.5 Improvements of CryptoVerif

This case study enabled us to find and fix two bugs in CryptoVerif, which did not affect the proof of simpler protocols of the literature on which it was previously tested. This case study also led to an improvement in CryptoVerif simplification algorithm, which was useful in order to handle the pseudo-random key derivation function. It also suggested future improvements of CryptoVerif that would make it easier to use. In particular,

- Improvements in the proof strategy should allow us to fully automate the proof in many more cases, in particular for public-key protocols. The prover should automatically distinguish cases in which the public key belongs to a honest principal or to the adversary.
- CryptoVerif is sometimes sensitive to the ordering of instructions, although the semantics of the game does not depend on this ordering. This problem could be solved by automatically moving let $x = \ldots$ and new x instructions under tests (duplicating them if necessary); this transformation would allow CryptoVerif to distinguish cases depending on which branch assigns x. (This transformation is currently performed only for new.)
- An additional game transformation would be helpful in order to prove some secrecy properties, in particular for key usability: tests if b then P else P' should be transformed into P when P and P' make indistinguishable actions, which would allow us to prove the secrecy of b. A first step would be to perform this transformation when P and P' are equal up to renaming of variables.

5. CONCLUSIONS

We have formalized and mechanically analyzed all three rounds of the Kerberos 5 protocol, both with and without its public-key extension PKINIT, using version 1.06pl3 of the CryptoVerif tool. This is the first mechanical security proof of an industrial protocol at the computational level. The success of CryptoVerif in proving security properties for Kerberos—and especially for PKINIT, the use of which makes Kerberos particularly complex—provides evidence of its utility for analyzing industrial protocols. This also extends other work on analyzing Kerberos to include mechanical analysis tools. In carrying out this work, we extended the idea of key usability to a new notion of strong key usability; this definition was helpful here, and we are interested in exploring its utility elsewhere.

We are currently broadening our study of how the cryptographic assumptions made here may be varied and how CryptoVerif copes with such changes. From our work with CryptoVerif thus far, we see that the use of this tool sharpens the user's understanding of the cryptographic subtleties involved in a protocol.

In the present work we have verified that the authentication keys and session keys are strongly usable for IND-CCA2 encryption. As the encryption scheme is assumed to also guarantee INT-CTXT security, it would be interesting to use CryptoVerif in order to find out whether the authentication keys and session keys are also (strongly) usable for INT-CTXT encryption.

Since the specifications of Kerberos and PKINIT [34, 23] are actually more complicated than our formalization, we would like utilize CryptoVerif on formalizations of basic and public-key Kerberos that are closer to the specifications, *e.g.*, by using a key derivation function for all symmetric keys.

Another area for future work is the mechanized analysis of PKINIT's Diffie-Hellman mode, which we did not study here. As noted in [13], the language of equivalences used by CryptoVerif will need to be extended in order to handle Diffie-Hellman key exchange, so this problem holds both theoretical and practical interest.

Acknowledgements. We are grateful to Michael Backes, Ricardo Corin, John Mitchell, Kenny Paterson, and Arnab Roy for helpful discussions.

6. REFERENCES

[1] M. Abadi and P. Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). In *First IFIP*, volume 1872 of *LNCS*. Springer, Aug. 2000.

[2] M. Abdalla, P.-A. Fouque, and D. Pointcheval. Password-Based Authenticated Key Exchange in the

Three-Party Setting. *IEE Proc. Information Security*, 153(1), 2006.

[3] A. Armando et al. The Avispa tool for the automated validation of internet security protocols and applications. In *CAV 2005*, volume 3576 of *LNCS*. Springer.

[4] M. Backes, I. Cervesato, A. D. Jaggard, A. Scedrov, and J.-K. Tsay. Cryptographically Sound Security Proofs for Basic and Public-key Kerberos. In *ESORICS 2006*, volume 4189 of *LNCS*. Springer, September 2006.

[5] M. Backes, B. Pfitzmann, and M. Waidner. A Composable Cryptographic Library with Nested Operations. In *CCS'03*. ACM, 2003.

[6] G. Bella and L. C. Paulson. Using Isabelle to Prove Properties of the Kerberos Authentication System. In *DIMACS'97, Workshop on Design and Formal Verification of Security Protocols (CD-ROM)*, 1997.

[7] G. Bella and L. C. Paulson. Kerberos Version IV: Inductive Analysis of the Secrecy Goals. In *ESORICS'98*, volume 1485 of *LNCS*. Springer, 1998.

[8] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *CRYPTO'96*, volume 1109 of *LNCS*. Springer, 1996.

[9] M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *ASIACRYPT 2000*, volume 1976 of *LNCS*. Springer, December 2000.

[10] S. M. Bellovin and M. Merritt. Limitations of the Kerberos Authentication System. In *USENIX Conference Proceedings*, Winter 1991.

[11] B. Blanchet. A computationally sound mechanized prover for security protocols. *IEEE Transactions on Dependable and Secure Computing*. To appear. Technical report version available at `http://eprint.iacr.org/2005/401`.

[12] B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *CSFW-14*, June 2001.

[13] B. Blanchet. A Computationally Sound Mechanized Prover for Security Protocols. In *IEEE Symposium on Security and Privacy*, May 2006.

[14] B. Blanchet. Computationally sound mechanized proofs of correspondence assertions. In *CSF 2007*, July 2007.

[15] B. Blanchet and D. Pointcheval. Automated Security Proofs with Sequences of Games. In *CRYPTO 2006*, volume 4117 of *LNCS*. Springer, Aug. 2006.

[16] A. Boldyreva and V. Kumar. Provable-security analysis of authenticated encryption in Kerberos. In *IEEE Symp. Security and Privacy*, 2007.

[17] F. Butler, I. Cervesato, A. D. Jaggard, A. Scedrov, and C. Walstad. Formal Analysis of Kerberos 5. *Theoretical Computer Science*, 367(1–2), 2006.

[18] R. Canetti and J. Herzog. Universally composable symbolic analysis of mutual authentication and key exchange protocols. In *TCC'06*, volume 3876 of *LNCS*. Springer, March 2006.

[19] I. Cervesato, A. D. Jaggard, A. Scedrov, J.-K. Tsay, and C. Walstad. Breaking and fixing public-key Kerberos. *Information and Computation*, FCS-ARSPA'06 Special Issue. To appear.

[20] V. Cortier and B. Warinschi. Computationally sound, automated proofs for security protocols. In *ESOP'05*, volume 3444 of *LNCS*. Springer, Apr. 2005.

[21] A. Datta, J. Mitchell, and B. Warinschi. Computationally Sound Compositional Logic for Key Exchange Protocols. In *CSFW'06*, July 2006.

[22] C. He, M. Sundararajan, A. Datta, A. Derek, and J. C. Mitchell. A modular correctness proof of TLS and IEEE 802.11i. In *CCS'05*. ACM, November 2005.

[23] IETF. Public Key Cryptography for Initial Authentication in Kerberos, 1996–2006. RFC 4556. Preliminary versions available as a sequence of Internet Drafts at `http://tools.ietf.org/wg/krb-wg/draft-ietf-cat-kerberos-pk-init/`.

[24] A. D. Jaggard, A. Scedrov, and J.-K. Tsay. Computationally Sound Mechanized Proof of PKINIT for Kerberos. Abstract presented at FCC'07.

[25] P. Laud. Secrecy Types for a Simulatable Cryptographic Library. In *CCS 2005*, May 2005.

[26] P. D. Lincoln, J. C. Mitchell, M. Mitchell, and A. Scedrov. A probabilistic poly-time framework for protocol analysis. In *CCS-5*, November 1998.

[27] P. D. Lincoln, J. C. Mitchell, M. Mitchell, and A. Scedrov. Probabilistic polynomial-time equivalence and security protocols. In *FM'99*, volume 1708 of *LNCS*. Springer, Sept. 1999.

[28] G. Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol using FDR. In *TACAS'96*, volume 1055 of *LNCS*. Springer, 1996.

[29] C. Meadows. Analysis of the Internet Key Exchange Protocol using the NRL Protocol Analyzer. In *IEEE Symp. Security and Privacy*, 1999.

[30] C. A. Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2), 1996.

[31] Microsoft. Security Bulletin MS05-042. `http://www.microsoft.com/technet/security/bulletin/MS05-042.mspx`, August 2005.

[32] J. Mitchell, A. Ramanathan, A. Scedrov, and V. Teague. A Probabilistic Polynomial-Time Process Calculus for the Analysis of Cryptographic Protocols. *Theoretical Computer Science*, 353(1–3), 2006.

[33] J. C. Mitchell, V. Shmatikov, and U. Stern. Finite-State Analysis of SSL 3.0. In 7^{th} *USENIX Security Symp.*, pages 201–216, 1998.

[34] C. Neuman, T. Yu, S. Hartman, and K. Raeburn. The Kerberos Network Authentication Service (V5), July 2005. `http://www.ietf.org/rfc/rfc4120`.

[35] K. Raeburn. Encryption and Checksum Specifications for Kerberos 5. `http://www.ietf.org/rfc/rfc3961.txt`, Feb. 2005.

[36] A. Roy, A. Datta, A. Derek, and J. C. Mitchell. Inductive proofs of computational secrecy. In *ESORICS 2007*, volume 4734 of *LNCS*. Springer, Sept. 2007.

[37] A. Roy, A. Datta, and J. C. Mitchell. Formal proofs of cryptographic security of Diffie-Hellman-based protocols. In *TGC'07*, Nov. 2007. To appear.

[38] C. Sprenger, M. Backes, D. Basin, B. Pfitzmann, and M. Waidner. Cryptographically Sound Theorem Proving. In *CSFW 2006*, July 2006.