Cryptographically Sound Security Proofs for Basic and Public-Key Kerberos

Michael Backes · Iliano Cervesato · Aaron D. Jaggard · Andre Scedrov · Joe-Kai Tsay

Abstract We present a computational analysis of basic Kerberos with and without its public-key extension PKINIT in which we consider authentication and key secrecy properties. Our proofs rely on the Dolev–Yao-style model of Backes, Pfitzmann, and Waidner, which allows for mapping results obtained symbolically within this model to cryptographically sound proofs if certain assumptions are met. This work was the first verification at the computational level of such a complex fragment of an industrial protocol. By considering a recently fixed version of PKINIT, we extend symbolic correctness results we previously attained in the Dolev–Yao model to cryptographically sound results in the computational model.

Backes was partially supported by the German Research Foundation (DFG) under grant 3194/1-1. Cervesato was partially supported by ONR under Grant N00014-01-1-0795 and by the Qatar Foundation under grant number 930107. Jaggard was partially supported by NSF Grants DMS-0239996, CNS-0429689, and CNS-0753492, and by ONR Grant N00014-05-1-0818. Scedrov was partially supported by OSD/ONR CIP/SW URI "Software Quality and Infrastructure Protection for Diffuse Computing" through ONR Grant N00014-01-1-0795 and OSD/ONR CIP/SW URI "Trustworthy Infrastructure, Mechanisms, and Experimentation for Diffuse Computing" through ONR Grant N00014-04-1-0725. Additional support from NSF Grants CNS-0429689 and CNS-0524059. Tsay was partially supported by ONR Grant N00014-01-1-0795 and NSF grant CNS-0429689. A preliminary version of this work appeared as [5].

Saarland University E-mail: backes@cs.uni-sb.de · Carnegie Mellon University, Qatar E-mail: iliano@cmu.edu · Rutgers University E-mail: adj@dimacs.rutgers.edu · University of Pennsylvania E-mail: scedrov@math.upenn.edu · LSV, ENS Cachan & CNRS & INRIA, France E-mail: tsay@lsv.ens-cachan.fr

1 Introduction

Cryptographic protocols have traditionally been verified in one of two ways: the first, known as the Dolev–Yao or symbolic approach, abstracts cryptographic concepts into an algebra of symbolic messages [33]; the second, known as the computational or cryptographic approach, retains the concrete view of messages as bitstrings and cryptographic operations as algorithms on bitstrings, while drawing security definitions from complexity theory [16, 35, 36]. While proofs in the computational approach (with its much more comprehensive adversary model) entail stronger security guarantees, conducting such proofs by hand is tedious and error-prone even for simple protocols and impractical for larger protocols. A first approach in mechanizing proofs in this model has so far only been tested on one commercial protocol [18]. On the other hand, verification methods based on the Dolev–Yao abstraction have become efficient and robust enough to tackle a wide range of large commercial protocols, often even automatically [1, 14, 15, 20, 21, 44].

Kerberos, a widely deployed protocol that allows a user to authenticate herself to multiple end servers based on a single login, constitutes one of the most important examples that have been formally analyzed within the Dolev–Yao approach so far. Kerberos 4, the then prevalent version, was verified using the Isabelle theorem prover [14,15]. The currently predominant version, Kerberos 5 [50], has been extensively analyzed using the Dolev–Yao approach. This analysis of Kerberos 5 showed: (a) the core protocol enjoys the expected authentication and secrecy properties except for some relatively innocuous anomalies [20]; (b) "cross-realm" authentication in Kerberos is correct when compared against its specification but has weaknesses in practice [27]; and (c) the then-current specification of the public-key extension (PKINIT) of Kerberos was susceptible to a serious attack [24–26]. The discovery of the attack on PKINIT led to an immediate correction of the specification and a security bulletin and patch for Microsoft Windows [46].

The earlier security proofs for both Kerberos 5 and the fixes to PKINIT were carried out in the Dolev–Yao approach. Thus, despite the extensive research dedicated to the Kerberos protocol, and despite its tremendous importance in practice, at the time of our preliminary report on this work [5] it remained an open question whether an actual implementation of Kerberos based on provably secure cryptographic primitives is secure under cryptographic security definitions with its much more comprehensive adversary. We closed this gap (at least partially) in the preliminary version of this paper [5] by providing the first security proofs of the core aspects of the Kerberos protocol in the computational approach. More precisely, we showed in the preliminary version of this paper [5] that core parts of Kerberos 5 are secure against arbitrary active attacks if the Dolev–Yao-based abstraction of the employed cryptography is implemented with actual cryptographic primitives that satisfy the commonly accepted security notions under active attacks, e.g., IND-CCA2 for public-key encryption.

Obviously, establishing proofs in the computational approach presupposes dealing with cryptographic details such as computational restrictions and error probabilities, hence one naturally assumes that our proofs heavily rely on complexity theory. However, our proofs are not performed from scratch in the cryptographic setting, but based on the Dolev–Yao-style framework of Backes, Pfitzmann, and Waidner [8, 11, 12] (called the *BPW model* henceforth), which provides computationally faithful symbolic abstractions of cryptographic primitives. The symbolically proved security properties also hold computationally when the symbolic abstractions are implemented with actual (secure) cryptographic operations. Thus our proofs themselves are symbolic in nature, but refer to primitives from the BPW model. Kerberos is the largest and most complex protocol whose cryptographic security has so far been inferred from proofs in this approach. Earlier proofs in this approach were conducted mainly for small examples of primarily academic interest [4, 7, 10]; some similar work had been done on industrial protocols, e.g., by He and Mitchell [38], although none that were as complex as Kerberos. (In Sec. 1.1, we note other analyses of industrial protocols that appeared after the preliminary report on this work [5].) We furthermore analyze the recently fixed version of PKINIT and derive computational guarantees for it from a symbolic proof based on the BPW model. Finally, we also draw some lessons learned in the process, which highlight areas where to focus research in order to simplify the verification of large commercial protocols with computational security guarantees. In particular it would be desirable to devise suitable proof techniques, based on the BPW model, for splitting large protocols into smaller pieces which can then be analyzed modularly while still retaining the strong link between the Dolev–Yao and the computational approaches. We view this as a research opportunity for the short-term future.

This paper extends work that has previously appeared in abbreviated form [5]. Differently from that presentation, here we present the full set of algorithms formalizing Kerberos and PKINIT as well as more complete proofs of our results. We have also changed our formalization of certificates binding keys to principals. Essentially, we now represent certificates using data structures which have previously been studied in the BPW model instead of defining new structures, which would require separate analysis of those structures.

1.1 Related Work

Early work on linking Dolev–Yao models and cryptography [2, 3, 37, 42] only considered passive attacks, and therefore cannot make general statements about protocols. A cryptographic justification for a Dolev–Yao model in the sense of simulatability [51], i.e., under active attacks and within arbitrary surrounding interactive protocols, was first given by Backes, Pfitzmann, and Waidner in [11] with extensions in [8, 12]. Based on that Dolev–Yao model, the well-known Needham-Schroeder-Lowe, Otway-Rees, and Yahalom protocols were proved secure in [4, 7, 10]. All these protocols are considerably simpler than Kerberos, which we analyze in this paper, and arguably of much more limited practical interest. Some work has been done on industrial protocols, such as 802.11i [38], although Kerberos is still a much more complex protocol.

Laud [43] has presented a cryptographic underpinning for a Dolev–Yao model of symmetric encryption under active attacks. His work is directly connected with a formal proof tool, but it is specific to certain confidentiality properties and protocol classes. Herzog et al. [39] and Micciancio and Warinschi [45] have also given a cryptographic underpinning under active attacks. They consider slightly simpler real implementations than in [11], but their results are specific for public-key encryption and certain protocol classes and are thus narrower than those in [11]. Cortier and Warinschi [29] have shown that symbolically secret nonces are also computationally secret, i.e., indistinguishable from a fresh random value given the view of a cryptographic adversary. Backes and Pfitzmann [9] and Canetti and Herzog [23] have established new symbolic criteria for proving a key cryptographically secret. Unfortunately, none of this work is comprehensive enough to provide computational security guarantees for Kerberos based on an existing symbolic proof; the work is missing suitable cryptographic primitives or it relies on slightly changed symbolic abstractions, e.g., as in [11].

Subsequent to the first version of this work [5], various additional related work has appeared in the literature. Boldyreva and Kumar showed in [19] that the encryption algorithm of the simplified profile of basic Kerberos satisfies the cryptographic assumptions made in [5] for symmetric encryption. They also showed that the general profile encryption of basic Kerberos is weak, and propose a corrected version of the general profile encryption that satisfies these properties. Roy et al. [52] also proved computational security of Kerberos. In [53], Roy et al. proved computational security of the PKINIT operation mode ("DH mode") that we do not consider here, as well as security of IKEv2. As another example of cryptographic proofs of security of an industrial-scale protocol, Gajek et al. [34] proved security properties of TLS. In work with Blanchet [18], the last three authors of this paper used the CryptoVerif tool [17] to mechanically prove security properties of Kerberos in the computational model. CryptoVerif relies on a probabilistic polynomial-time process calculus [48].

There is also other work on formulating syntactic calculi for dealing with probability and polynomial-time considerations and encoding them into proof tools, in particular [30, 41, 47]. This is orthogonal to the work of justifying Dolev–Yao models, which offer a higher level of abstractions and thus much simpler proofs where applicable, so that proofs of larger systems can be automated.

1.2 Structure of the Paper

We start in Section 2 with a review of Kerberos and its public-key extension PKINIT. In Section 3, we recall the BPW model (e.g., [6, 8, 12, 13]), and apply it to the specification of Kerberos 5 and public-key Kerberos (i.e., Kerberos with PKINIT). Section 4 proves security results for these protocols and lifts them to the computational level. Finally, Section 5 summarizes this effort and outlines areas of future work.

2 Kerberos 5 and its Public-Key Extension

The Kerberos protocol [49, 50] allows a legitimate user to log on to her terminal once a day (typically) and then transparently access all the networked resources she needs for the rest of that day in her organization. Each time she wants to, e.g., retrieve a file from a remote server, a Kerberos client running on her behalf securely handles the required authentication. The client acts behind the scenes, without any user intervention.

The main Kerberos protocol comprises three exchanges: the initial round of authentication, in which the client obtains log-in credentials that might be good for a full day; the second round of authentication, in which she presents her first credentials in order to obtain a short-term credentials (five-minute lifetime) to use a particular network service; and the client's interaction with the network service, in which she presents her short-term credentials in order to negotiate access to the service.

In the core specification of Kerberos 5 [50], all three exchanges solely use symmetric (shared-key) cryptography. Since the initial specification of Kerberos 5, the protocol has been extended by the definition of an alternate first round which uses asymmetric (public-key) cryptography. This alternative exchange, that is called PKINIT, may be used in two modes: "public-key encryption mode" and "Diffie–Hellman (DH) mode." In recent work [24–26], we showed that there was an attack against the then-current draft specification of PKINIT when the public-key encryption mode was used and then symbolically proved the security of the specification as it was revised in response to our attack. Here we study both basic Kerberos (without PKINIT) and the public-key mode of PKINIT as it was revised to prevent our attack. The fix first appeared in revision 27 of the PKINIT specification [40]; subsequent drafts have not changed this aspect of PKINIT. The fix is also present in the current version of PKINIT [55], which is now a RFC within the IETF [54] standards process. In the rest of this section, we describe the operation of both basic Kerberos and Kerberos with PKINIT in public-key mode.

Kerberos Basics The client process—usually acting for a human user—interacts with three additional types of principals when using Kerberos 5 (with or without PKINIT). The client's goal is to be able to authenticate herself to various application servers (e.g., email, file, and print servers). This is done by obtaining a "ticket-granting ticket" (TGT) from a "Kerberos Authentication Server" (KAS) and then presenting this to a "Ticket-Granting Server" (TGS) in order to obtain

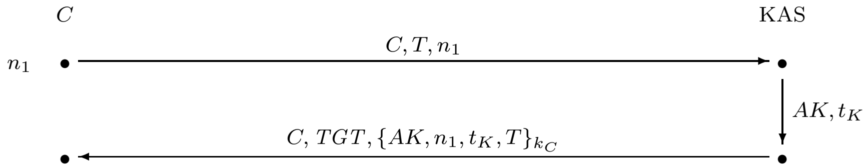

Fig. 1 Message Flow in the Traditional AS Exchange, where $TGT = \{AK, C, t_K\}_{k_T}$.

a "service ticket" (ST), the credentials that the client uses to authenticate herself to the application server. A TGT might be valid for a day, and may be used to obtain several STs for many different application servers from the TGS, while a single ST is valid for a few minutes (although it may be used repeatedly) and is used for a single application server. The KAS and the TGS are together known as the "Key Distribution Center" (KDC).

The client's interactions with the KAS, TGS, and different application servers are called the Authentication Service (AS), Ticket-Granting (TG), and Client-Server (CS) exchanges, respectively. We will describe the AS exchange separately for basic and public-key Kerberos; as PKINIT does not modify the other exchanges, we only need to describe them once.

The Traditional AS Exchange The abstract structure of the AS exchange is given in Figure 1. A client C generates a fresh nonce n_1 and sends it, together with her own name and the name T of the TGS for whom she desires a TGT, to the KAS K. This message is called the AS_REQ message [50]. The KAS responds by generating a fresh authentication key AK for use between the client and the TGS and sending an AS_REP message to the client. Within this message, AK is sent back to the client in the encrypted message component $\{AK, n_1, t_K, T\}_{k_C}$; this also contains the nonce n_1 from the AS_REQ, the KAS's local time t_K, and the name of the TGS for whom the TGT was generated. (The AK and t_K to the right of the figure illustrate that these values are new between the two messages.) This component is encrypted under a long-term key k_C shared between C and the KAS; this key is usually derived from the user's password. This is the only time that k_C is used in a standard Kerberos run because later exchanges use freshly generated keys. AK is also included in the ticket-granting ticket TGT sent alongside the message encrypted for the client. The TGT consists of AK, C, t_K, where t_K is K's local time, encrypted under a long-term key k_T shared between the KAS and the TGS named in the request. The computational model we use here does not support timestamps, so we will treat these as nonces; as shown in [28], this does not alter the authentication and confidentiality properties of a protocol such as Kerberos. These encrypted messages are accompanied by the client's name—and other data that we abstract away—sent in the clear. Once the client has received this reply, she may undertake the Ticket-Granting exchange.

It should be noted that the actual AS exchange, as well as the other exchanges in Kerberos, is more complex than the abstract view given here. We refer the reader to [50] for the complete specification of Kerberos 5, [55] for the specification of PKINIT, and [20] for a formalization of Kerberos at an intermediate level of detail.

The AS Exchange with PKINIT PKINIT [40] is an extension to Kerberos 5 that uses public key cryptography to avoid shared secrets between a client and KAS; it modifies the AS exchange but not other parts of the basic Kerberos 5 protocol. The long-term shared key (k_C) in the traditional AS exchange is typically derived from a password, which limits the strength of the authentication to the user's ability to choose and remember good passwords; PKINIT does not use k_C and thus avoids this problem. Furthermore, if a public key infrastructure (PKI) is already in place, PKINIT allows network administrators to use it rather than expending additional effort to manage users' long-term keys as in traditional Kerberos. This protocol extension adds complexity to Kerberos as it retains symmetric encryption in the later rounds but relies on asymmetric encryption, digital signatures, and corresponding certificates in the first round.

In PKINIT, the client C and the KAS each possess public/private key pairs, (pk_C, sk_C) and (pk_K, sk_K), respectively. Certificate sets $Cert_C$ and $Cert_K$ issued by a PKI independent from Kerberos are used to testify of the binding between each principal and her purported public key. This simplifies administration as authentication decisions can now be made based on the trust the KDC holds in just a few known certification authorities within the PKI, rather than keys individually shared with each client (local policies can, however, still be installed for user-by-user authentication). Dictionary attacks are defeated as user-chosen passwords

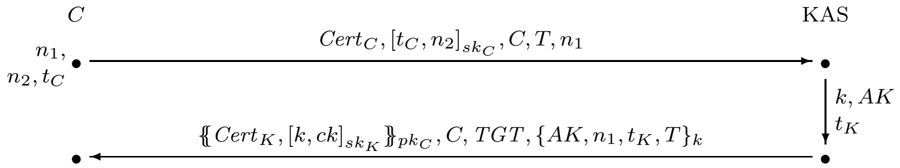

Fig. 2 Message flow in the fixed version of PKINIT, where $TGT = \{AK, C, t_K\}_{k_T}$.

are replaced with automatically generated asymmetric keys.[1]

As noted above, PKINIT can operate in two modes. These resemble the basic AS exchange in that the KAS generates a fresh key AK for the client and TGS to use, and then the KAS transmits AK and the TGT to the client. The modes of PKINIT provide two different ways for the KAS to transmit this key using the asymmetric key pairs rather than a key that is shared between the client and KAS. In DH mode, the key pairs (pk_C, sk_C) and (pk_K, sk_K) are used to provide digital signature support for an authenticated Diffie-Hellman key agreement which produces a key which is then used to protect the fresh key AK. A variant of this mode allows the reuse of previously generated shared secrets. In public-key encryption mode, analyzed here, the key pairs are used for both signature and encryption. The latter is designed to (indirectly) protect the confidentiality of AK, while the former ensures its integrity.

We will not discuss the DH mode any further; the only support we are aware of for this mode is in the PacketCable system [22], developed by CableLabs, a cable television research consortium. As noted above, DH mode has been analyzed by Roy et al. [53].

Figure 2 illustrates the AS exchange when the fixed version (which defends against the attack of [24–26]) of PKINIT is used. Here we use $[m]_{sk}$ for the digital signature of message m with secret key sk, $\{\!\{m\}\!\}_{pk}$ for the encryption of m with the public key pk, and $\{m\}_k$ for the encryption of m with the symmetric key k.

The first line of Fig. 2 shows our formalization of the AS_REQ message that a client C sends to a KAS K when using PKINIT. The last part of the message — C, T, n_1 — is the message in the traditional AS_REQ message. The new data that is added by PKINIT consists of the client's certificates $Cert_C$ and her signature (with her secret key sk_C) over a timestamp t_C and another nonce n_2. (The nonces and timestamp at the left of this line indicate that these are generated by C specifically for this request.)

The second line in Figure 2 shows our formalization of K's response, which is more complex than in basic Kerberos. The last part of the message—C, TGT, $\{AK, n_1, t_K, T\}_k$—is very similar to K's reply in basic Kerberos; the difference is that the symmetric key k (which takes the place of k_C in basic Kerberos) protecting AK is now freshly generated by K and is not a long-term shared key. Because k is freshly generated for the reply, it must be communicated to C before she can learn AK. PKINIT does this by adding the message $\{\!\{Cert_K, [k, ck]_{sk_K}\}\!\}_{pk_C}$. This contains K's certificates and his signature, using his secret key sk_K, over k and a keyed hash ck ('checksum' in the language of [50]) taken over the entire request AS_REQ from C using the key k; all of this is encrypted under C's public key pk_C. The keyed hash ck binds this response to the client's request and was added in response to the attack we discovered and reported in [24–26].

The Later Exchanges After the client C has obtained the key AK and the TGT, either through the basic AS exchange or the PKINIT AS exchange, she then initiates the TGS exchange. This exchange is shown in Fig. 3. The first line of this figure shows our formalization of the client's request, called a TGS_REQ message; it contains the TGT (which is opaque to the client), an *authenticator* $\{C, t_C\}_{AK}$, the name of the server S for which C desires a service ticket, and a nonce n_3. Once the TGS receives this message, he decrypts the TGT to learn AK and uses this to decrypt the authenticator. Assuming his local policies for granting a service ticket are satisfied (while we do not model these here, they might include checks such as whether the request is sufficiently fresh), the TGS produces a fresh key SK for C and S to share and sends this back to the client in a TGS_REP message. The form of this message is essentially the same as the basic AS_REP message from the KAS to C: it contains a ticket (now the service ticket, or ST, $\{SK, C, t_T\}_{k_S}$ instead of the TGT) encrypted for the next server (now S instead of T) and encrypted data for C (now encrypted under AK instead of k_C or k).

[1] The login process changes as very few users would be able to remember a random public/secret key pair. In Microsoft Windows, keys and certificate chains are stored in a smartcard that the user swipes in a reader at login time. A passphrase is generally required as an additional security measure [32]. Other possibilities include keeping these credentials on the user's hard drive, again protected by a passphrase.

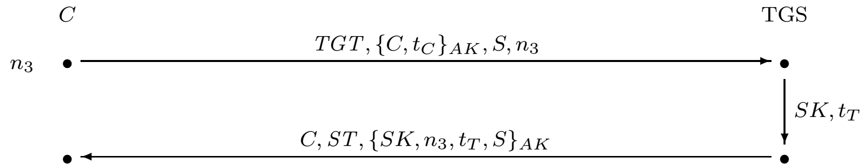

Fig. 3 Message flow in the TGS exchange, where $TGT = \{AK, C, t_K\}_{k_T}$ and $ST = \{SK, C, t_T\}_{k_S}$.

Finally, after using the AS exchange to obtain the key SK and the ST, the client may use the CS exchange to authenticate herself to the end server. Figure 4 shows this exchange, including the optional reply from the server that authenticates this server to the client. As shown in the first line of the figure, C starts by sending a message (AP_REQ) that is similar to the TGS_REQ message of the previous round: in contains the (service) ticket and an authenticator ($\{C, t'_C\}_{SK}$) that is encrypted under the key contained in the ST. As shown in the second line of the figure, the server S simply responds with an AP_REP message $\{t'_C\}_{SK}$ containing the timestamp from the authenticator encrypted under the key from the ST.

Attack on PKINIT The attack we found against the then-current specification of PKINIT was reported in [24–26]. This attack was possible because, at the time, the reply from the KAS to the client contained $[k, n_2]_{sk_K}$ in place of $[k, ck]_{sk_K}$. In particular, the KAS did not sign any data that depended upon the client's name. This allowed an attacker who was herself a legitimate client to intercept a message from another client C to the KAS, use this data in her own request to the KAS, read the reply from the KAS, and then send this reply to C as though it was generated by the KAS for C (instead of for the attacker). The effect of this attack was that the attacker could gain knowledge of all new keys shared between the client and various servers. It could do so either by translating messages as in the AS exchange (collecting keys along the way), or by impersonating these servers (and creating the keys in the first place). In the former variation, the client would be authenticated as the attacker and not as C.

Security Properties We now summarize the security properties that we prove here at the symbolic level for both basic Kerberos and Kerberos with PKINIT; the implications on the computational level are discussed in the subsequent sections. We have proved similar properties in symbolic terms using a formalization in MSR for basic Kerberos [20, 21] and for the AS exchange when PKINIT is used [24–26]. Our subsequent work with CryptoVerif has given mechanized proofs of similar properties in the computational model [18]. The first property we prove here concerns the secrecy of exchanged keys, a notion that is captured formally as Definition 1 in Section 4. This property may be summarized as follows.

Property 1 (Key secrecy) For any honest client C and honest server S, if the TGS T generates a symmetric key SK for C and S to use (in the CS exchange), then the intruder does not learn the key SK.

The second property we study here concerns entity authentication, formalized as Definition 2 in Section 4. This property may be summarized as follows.

Property 2 (Authentication properties)

i. If a server S completes a run of Kerberos, apparently with C, then earlier: (a) C started the protocol with some KAS to get a ticket-granting ticket; and (b) then requested a service ticket from some TGS.
ii. If a client C completes a run of Kerberos, apparently with server S, then S sent a valid AP_REP message to C.

Theorem 1 below shows that these properties hold for our symbolic formalizations of basic and public-key Kerberos in the BPW model; Theorem 2 shows that the authentication property holds as well for cryptographic implementations of these protocols if provably secure primitives are used; the standard cryptographic definition of key secrecy however turns out not to hold for cryptographic implementations of Kerberos. We will return to this point below. Because authentication can be shown to hold for Kerberos with PKINIT, it follows that at the level of cryptographic implementation, the fixed specification of PKINIT does indeed defend against the attack reported in [24–26].

3 The BPW Model

We will now abstractly review the BPW model and then formalize Kerberos using it.

3.1 Review of the BPW Model

The BPW model introduced in [13] offers a deterministic Dolev–Yao style formalism of cryptographic proto-

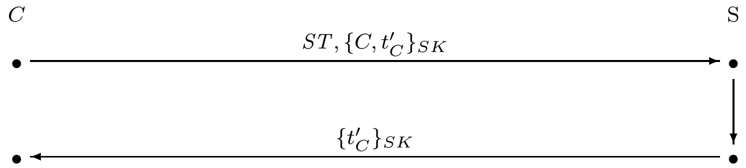

Fig. 4 Message flow in the CS exchange, where $ST = \{SK, C, t_T\}_{k_S}$.

cols with commands for a vast range of cryptographic operations such as public-key and symmetric-key encryption/decryption, generation and verification of digital signatures as well as message authentication codes, and nonce generation as well as the inclusion of payloads (application data). Every protocol participant is assigned a machine (an I/O automaton), which is connected to the machines of other protocol participants and which executes the protocol for its user by interacting with the other machines (see Fig. 5). In this reactive scenario, semantics is based on state, i.e., on who already knows which terms. The state is here represented by an abstract "database" and handles to its entries: Each entry (denoted $D[j]$) of the database has a type (e.g., "signature") and pointers to its arguments (e.g., "private key" and "message"). This corresponds to the way Dolev–Yao terms are represented. Furthermore, each entry in the abstract database also comes with handles to participants who have access to that entry. These handles determine the state. The BPW model does not allow cheating: only if a participant has a handle to the entry $D[j]$ itself or to the right entries that could produce a handle to $D[j]$ can the participant learn the term stored in $D[j]$. For instance, if the BPW model receives a command, e.g., from a user machine, to encrypt a message m with key k, then it makes a new abstract database entry for the cyphertext with a handle to the participant that sent the command and pointers to the message and the key as arguments; only if a participant has handles to the cyphertext and also to the key can the participant ask for decryption. Furthermore, if the BPW model receives the same encryption command a second time then it will generate a new (different) entry for the cyphertext. This meets the fact that secure encryption schemes are necessarily probabilistic. Entries are made known to other participants by a send command, which adds handles to the entry.

The BPW model is based on a detailed model of asynchronous reactive systems introduced in [51] and is represented as a deterministic machine $\mathrm{TH}_{\mathcal{H}}$ (also an I/O automaton), called *trusted host*, where $\mathcal{H} \subset \{1, \ldots, n\}$ denotes the set of honest participants out of all m participants. This machine executes the commands from the user machines, in particular including the commands for cryptographic operations. A *system* consists of several possible *structures*. A structure consists of a set $\hat{M}$ of connected correct user machines and a subset S of the free ports, i.e., S is the user interface of honest users. In order to analyze the security of a structure $(\hat{M}, S)$, an arbitrary probabilistic polynomial-time *user* machine H is connected to the user interface S and a polynomial-time *adversary* machine A is connected to all the other ports and H. This completes a structure into a *configuration* of the system (see Fig. 5). The machine H represents all users. A configuration is a runnable system, i.e., for each security parameter k, which determines the input lengths (including the key length), one gets a well-defined probability space of *runs*. The BPW model maintains length functions on the entries of the abstract database; to guarantee that the system is polynomially bounded in the security parameter, there are then bounds on the lengths of messages, as well as bounds on the number of signatures per key and the number of inputs per port [13]. The *view* of H in a run is the restriction to all inputs and outputs that H sees at the ports it connects to, together with its internal states. Formally one defines the view $view_{conf}(\mathsf{H})$ of H for a configuration *conf* to be a family of random variables X_k where k denotes the security parameter. For a given security parameter k, X_k maps runs of the configuration to a view of H.

Corresponding to the BPW model, there exists a cryptographic implementation of the BPW model and a computational system, in which honest participants also operate via handles on cryptographic objects. However, the objects are now bitstrings representing real cryptographic keys, cyphertexts, etc., acted upon by interactive polynomial-time Turing machines (instead of the symbolic machines and the trusted host). The implementation of the commands now uses provably secure cryptographic primitives according to standard cryptographic definitions (with small additions like type tagging and additional randomization). In [8, 11–13] it was established that the cryptographic implementation of the BPW model is *at least as secure as* the BPW model (denoted by $\geq$, see Fig. 6), meaning that whatever an active adversary can do in the implementation can also be achieved by another adversary in the

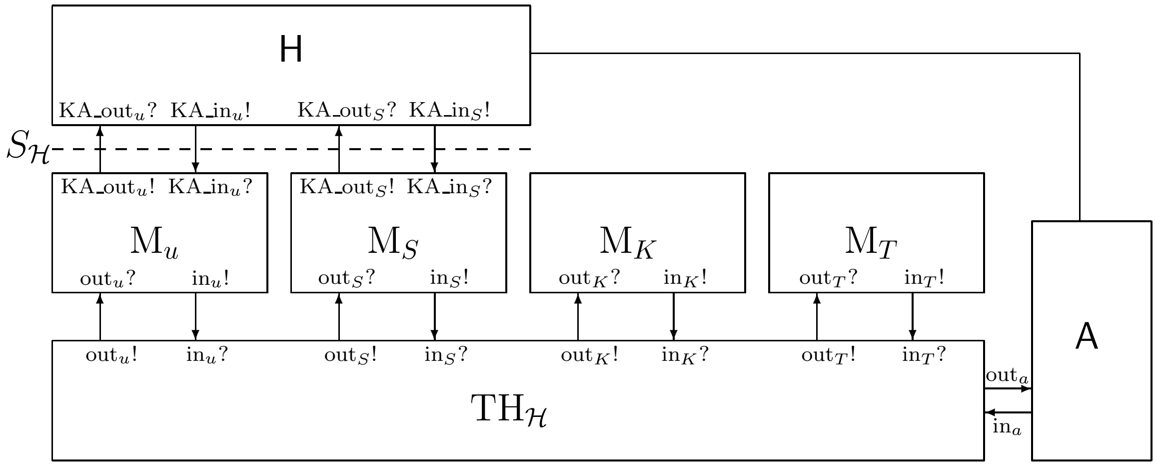

Fig. 5 Overview of the Kerberos symbolic system

BPW model, or the underlying cryptography can be broken. More formally, a system Sys_1 being at least as secure as another system Sys_2 means that for all probabilistic polynomial-time user H, for all probabilistic polynomial-time adversary A_1 and for every computational structure $(\hat{M}_1, \mathsf{S}) \in Sys_1$, there exists a polynomial-time adversary A_2 on a corresponding symbolic structure $(\hat{M}_2, \mathsf{S}) \in Sys_2$ such that the view of H is computationally indistinguishable in both configurations (Fig. 6). This captures the cryptographic notion of *reactive simulatability*.

3.2 Public-key Kerberos in the BPW Model

We now model the Kerberos protocol in the framework of [13] using the BPW model. We write ":=" for deterministic assignment, "=" for testing for equality and "$\leftarrow$" for probabilistic assignment.

The descriptions of the symbolic systems of Kerberos 5 and PKINIT are very similar, with the difference that the user machines follow different algorithms for the two protocols. We denote Kerberos with PKINIT by "PK," and basic Kerberos by "K5." If we let Kerb$\in$\{PK, K5\} then, as described in Section 3.1, for each user $u \in \{1, \ldots, n\}$ there is a *protocol machine* $\mathrm{M}_u^{\mathrm{Kerb}}$ which executes the protocol for u. There are also protocol machines for the KAS K and the TGT T, denoted by $\mathrm{M}_K^{\mathrm{Kerb}}$ and $\mathrm{M}_T^{\mathrm{Kerb}}$. Furthermore, if $S_1, \ldots, S_l$ are the servers in T's 'realm',[2] then there are server machines $\mathrm{M}_S^{\mathrm{Kerb}}$ for $S \in \{S_1, \ldots, S_l\}$. Each user machine is connected to the user via ports: A port for outputs to the user and a port for inputs from the user, labeled KA_out$_u$! and KA_in$_u$?, respectively ("KA" for"Key sharing and Authentication"). The ports for the server machines are labeled similarly (see Fig. 5).

The behaviors of the protocol machines is described in Algorithm 1 to 5 (Figs. 7–15). In the following, we comment on two algorithms of PKINIT (Fig. 7 and Fig. 8), the others are displayed in Appendix A. If, for instance, a protocol machine $\mathrm{M}_u^{\mathrm{PK}}$ receives a message ($\mathsf{new\_prot}$, PK, K, T) at KA_in$_u$? then it will execute Algorithm 1A (Fig. 7) to start a protocol run. We give a description below. The state of the protocol machine $\mathrm{M}_u^{\mathrm{Kerb}}$ consists of the bitstring u and the sets $Nonce_u$, $Nonce2_u$, $TGTicket$, and $Session\_KeysS_u$, in which $\mathrm{M}_u^{\mathrm{Kerb}}$ stores nonces, ticket-granting tickets, and the session keys for server S, respectively. This is the information a client needs to remember during a protocol run.

Only the machines of honest users $u \in \{1, \ldots, n\}$ and honest servers $S \in \{S_1, \ldots, S_l\}$ will be present in the protocol run, in addition to the machines for K and T. The others are subsumed in the adversary. We denote by $\mathcal{H} \subset \{1, \ldots, n,\ K,\ T,\ S_1, \ldots, S_l\}$ the honest participants, i.e., for $v \in \mathcal{H}$ the machine $\mathrm{M}_v^{\mathrm{Kerb}}$ is guaranteed to run correctly. And we assume that KAS K and TGS T are always honest, i.e., $K, T \in \mathcal{H}$.

Furthermore, given a set $\mathcal{H}$ of honest participants, with $\{K, T\} \subset \mathcal{H} \subset \{1, \ldots, n,\ K,\ T,\ S_1, \ldots, S_l\}$ the user interface of public-key Kerberos will be the set $S_{\mathcal{H}} := \{\text{KA\_out}_u!, \text{KA\_in}_u? \,|\, u \in \mathcal{H} \setminus \{K, T\}\}$. The symbolic system is the set $Sys^{\mathrm{Kerb,\,symb}} := \{(\hat{M}_{\mathcal{H}}, S_{\mathcal{H}})\}$. Note that, because we are working in an asynchronous system, we are replacing protocol timestamps by arbitrary messages that we assume are known to the participants generating the timestamps (e.g. nonces). All

[2] I.e., administrative domain; we do not consider cross-realm authentication here, although it has been analyzed symbolically in [27]

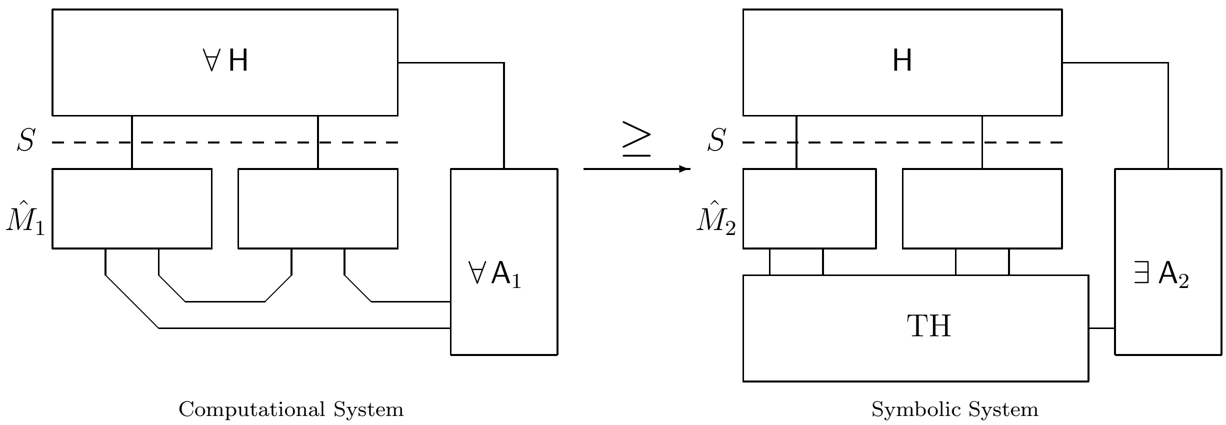

Fig. 6 Simulatability: The views of H must be indistinguishable

algorithms should immediately abort if a command to the BPW model yields an error, e.g., if a decryption request fails.

Notation The entries of the database D are all of the form *(ind, type, arg,* $hnd_{u_1}, \ldots, hnd_{u_m}, hnd_a$*, len)*, where $\mathcal{H} = \{u_1, \ldots, u_m\}$. We denote by $\downarrow$ an error element available to all ranges and domains of all functions and algorithms. So, e.g., $hnd_a = \downarrow$ means the adversary does not have a handle to the entry. For entries $x \in D$, the *index* $x.ind \in \mathcal{INDS}$ consecutively numbers all entries in D. The set $\mathcal{INDS}$ is isomorphic to $\mathbb{N}$ and is used to distinguish index arguments. We write $D[i]$ for the selection $D[ind = i]$, i.e., it is used as a primary key attribute of the database. The entry $x.type \in typeset = \{\text{auth, cert, enc, nonce, list, pke, pkse, sig, ske, skse}\}$ identifies the type of x. Here ske/pke is a private/public key pair and skse is a symmetric key which comes with a 'public' key pkse. This "public key identifier" pkse cannot be used for any cryptographic operation but works as a pointer to skse instead (see [7] for a more detailed explanation). The entry $x.arg = (a_1, \ldots, a_j)$ is a possibly empty list of arguments. Many values a_i are in $\mathcal{INDS}$. $x.hnd_u \in \mathcal{HNDS} \cup \{\downarrow\}$ for $u \in \mathcal{H} \cup \{a\}$ are handles by which u knows this entry. We always use a superscript "*hnd*" for handles. $x.len \in \mathbb{N}_0$ denotes the "length" of the entry; it is computed by applying length functions (mentioned in Sec. 3.1).

Initially, D is empty. $\text{TH}_{\mathcal{H}}$ has a counter $size \in \mathcal{INDS}$ for the current size of D. For the handle attributes, it has counters $currhnd_u$ initially 0. First we need to add the symmetric keys shared exclusively by K and T, S and T. Public-key Kerberos uses certificates; therefore, in this case all users need to know the public key for certificate authorities and have their own public-key certificates signed by a certificate authority. For simplicity we use only one certificate authority CA. Therefore, we add to D an entry for the public key of CA with handles to all users (i.e., to all user machines). And for every user we add an entry for the certificate of that user signed by the certificate authority with a handle to the user (machine). In the case of Kerberos 5, we are adding entries for the key k_u shared exclusively by K and u, for all user u.

Note that, in contrast to the analysis of Kerberos in MSR, the BPW model does not come with a type for certificates. As certificates rely on signatures of a certificate authority with the intend of binding a users name and the user's public key, here we formalize certificates as signature by an certificate authority over a list consisting of a user's name and the user's public key. Alternatively, one could extend the symbolic BPW model by defining a type certificate and basic commands for the ideal BPW model that, e.g., verify the certificates and extract the public key of a user from the certificate (the approach that we took in our initial report [5]). One should then also construct corresponding commands for the computational BPW model and show that the soundness (which relies on the already proved soundness of signatures) of the results showed in the symbolic BPW model still holds when one reasons about certificates; we expect that this would be similar to the proof of soundness for signatures, and that it would not pose a problem.

Example of Algorithms We are only going to examine PKINIT (Fig. 2) and explain the steps of its Algorithms 1A and 2 (Fig. 7 and Fig. 8), which are more complex than the algorithms in Kerberos 5. However, with these explanations the remaining algorithms (Appendix A) should be easily understandable. For details on the definition of the used commands see [8, 12, 13]. For readability of the figures, we note on the right (in curly brackets) to which terms in the more commonly

A) Input:$(\mathsf{new\_prot}, \mathsf{PK}, K, T)$ at $KA\_\text{in}_u?$.

1. $n_{u,1}t_u^{hnd} \leftarrow \mathsf{gen\_nonce}()$
2. $n_{u,2}^{hnd} \leftarrow \mathsf{gen\_nonce}()$
3. $l^{hnd} \leftarrow \mathsf{list}(t_u^{hnd}, n_{u,2}^{hnd})$ $\{l \approx (t_C, n_2)\}$
4. $s^{hnd} \leftarrow \mathsf{sign}(ske_u^{hnd}, l^{hnd})$ $\{s \approx [t_C, n_2]_{sk_C}\}$
5. $u^{hnd} \leftarrow \mathsf{store}(u)$
6. $T^{hnd} \leftarrow \mathsf{store}(T)$
7. $m_1^{hnd} \leftarrow \mathsf{list}(cert_u^{hnd}, s^{hnd}, u^{hnd}, T^{hnd}, n_{u,1}^{hnd})$ $\{m_1 \approx Cert_C, [t_C, n_2]_{sk_C}, C, T, n_1\}$
8. $Nonce_u := Nonce_u \cup \{(n_{u,1}^{hnd}, m_1^{hnd}, K)\}$
9. $\mathsf{send\_i}(K, m_1^{hnd})$

B) Input:$(\mathsf{continue\_prot}, \mathrm{PK}, T, S, AK^{hnd})$ at $\text{KS\_in}_u?$ for $S \in \{S_1, ..., S_l\}$

1. **if** $(\nexists (TGT^{hnd}, AK^{hnd}, T) \in TGTicket_u)$ **then**
2. Abort
3. **end if**
4. $z^{hnd} \leftarrow \mathsf{list}(u^{hnd}, t_u^{hnd})$ $\{z \approx C, t_C\}$
5. $auth^{hnd} \leftarrow \mathsf{sym\_encrypt}(AK^{hnd}, z^{hnd})$ $\{auth \approx \{C, t_C\}_{AK}\}$
6. $n_{u,3}^{hnd} \leftarrow \mathsf{gen\_nonce}()$
7. $Nonce2_u := Nonce2_u \cup \{n_{u,3}^{hnd}, T, S)\}$
8. $m_3^{hnd} \leftarrow \mathsf{list}(TGT^{hnd}, auth^{hnd}, u^{hnd}, S^{hnd}, n_{u,3}^{hnd})$ $\{m_3 \approx TGT, \{C, t_C\}_{AK}, C, S, n_3\}$
9. $\mathsf{send\_i}(T, m_3^{hnd})$

Fig. 7 Algorithm 1 of Public-key Kerberos: Evaluation of inputs from the user (starting the AS and TG exchange).

used Dolev–Yao notation the terms in the algorithms correspond ($\approx$).

Protocol start of PKINIT. In order to start a new run of PKINIT, user u inputs (new_prot, PK, K, T) at port KA_in$_u$?. Upon such an input, $\mathrm{M}_u^{\mathrm{PK}}$ runs Algorithm 1A (Fig. 7) which prepares and sends the AS_REQ, i.e., the first message in the AS exchange, to K using the BPW model. $\mathrm{M}_u^{\mathrm{PK}}$ generates symbolic nonces in steps 1A.1 and 1A.2 by sending the command $\mathsf{gen\_nonce}()$. In step 1A.3 the command $\mathsf{list}(\_,\_)$ concatenates t_u and $n_{u,2}$ into a new list that is signed in step 1A.4 with u's private key. Because we are working in an asynchronous system, the timestamp t_u is approximated by some arbitrary message (e.g., by a nonce). The command $\mathsf{store}(\_)$ in step 1A.5–6 makes entries in the database for the names of u and T. Handles for the names u and T are returned, which are added to a list in the next step. $\mathrm{M}_u^{\mathrm{PK}}$ stores information in the set $Nonce_u$, which it will need later in the protocol to verify the message authentication code sent by K. In step 1A.8 $Nonce_u$ is updated. Finally, in step 1A.9 the AS_REQ is sent over an insecure ("i" for "insecure") channel.

Behavior of the KAS K in PKINIT. Upon input (v, K, i, m^{hnd}) at port out$_K$? with $v \in \{1, .., n\}$, the machine $\mathrm{M}_K^{\mathrm{PK}}$ runs Algorithm 2 (Fig. 8), which first checks if the message m is a valid AS_REQ and then prepares and sends the corresponding AS_REP. In order to verify that the input is a possible AS_REQ, the types of

Input:(v, K, i, m^{hnd}) at $\text{out}_K?$ with $v \in \{1, ..., n\}$.

1. $x_i^{hnd} \leftarrow \mathsf{list\_proj}(m^{hnd}, i)$ for $i = 1, ..., 5$
2. $type_i \leftarrow \mathsf{get\_type}(x_i^{hnd})$ for $i = 1, 2, 5$ $\{x_1 \approx Cert_C, x_2 \approx [t_C, n_2]_{sk_C}, x_5 \approx n_1\}$
3. $x_i \leftarrow \mathsf{retrieve}(x_i^{hnd})$ for $i = 3, 4$ $\{x_3 \approx C, x_4 \approx T\}$
4. **if** $(type_1 \neq \text{sig}) \vee (type_2 \neq \mathsf{sig}) \vee (type_5 \neq \text{Nonce}) \vee (x_3 \neq v) \vee (x_4 \neq T)$ **then**
5. Abort
6. **end if**
7. $y_2^{hnd} \leftarrow \mathsf{msg\_of\_sig}(x_1^{hnd})$
8. $b \leftarrow \mathsf{verify}(x_1^{hnd}, pke_{CA}^{hnd}, y_2^{hnd})$
9. **if** $b = false$ **then**
10. Abort
11. **end if**
12. $w_j^{hnd} \leftarrow \mathsf{list\_proj}(y_2^{hnd}, i)$ for $i = 1, 2$
13. $w_1 \leftarrow \mathsf{retrieve}(w_1^{hnd})$
14. $type_6 \leftarrow \mathsf{get\_type}(w_2^{hnd})$
15. **if** $(type_6 \neq \text{pke} \vee w_1 \neq v$ **then**
16. Abort
17. **end if**
18. $y_1^{hnd} \leftarrow \mathsf{msg\_of\_sig}(x_2^{hnd})$ $\{y_1 \approx t_C, n_2\}$
19. $b \leftarrow \mathsf{verify}(x_2^{hnd}, w_2^{hnd}, y_1^{hnd})$ $\{x_2 \approx [t_C, n_2]_{sk_C}\}$
20. **if** $b = false$ **then**
21. Abort
22. **end if**
23. $y_{1i}^{hnd} \leftarrow \mathsf{list\_proj}(y_1^{hnd}, i)$ for $i = 1, 2$ $\{y_{11} \approx t_C, y_{12} \approx n_2\}$
24. $type_{12} \leftarrow \mathsf{get\_type}(y_{12}^{hnd})$
25. **if** $(type_{12} \neq \text{nonce}) \vee ((y_{12}^{hnd}, .) \in Nonce3_K)$ **then**
26. Abort
27. **end if**
28. $Nonce3_K := Nonce3_K \cup \{(y_{12}^{hnd}, v)\}$
29. $k_e^{hnd} \leftarrow \mathsf{gen\_symenc\_key}()$
30. $k_a^{hnd} \leftarrow \mathsf{gen\_auth\_key}()$
31. $AK^{hnd} \leftarrow \mathsf{gen\_symenc\_key}()$
32. $auth^{hnd} \leftarrow \mathsf{auth}(k_a^{hnd}, m^{hnd})$ $\{auth \approx ck\}$
33. $z_1^{hnd} \leftarrow \mathsf{list}(k_e^{hnd}, k_a^{hnd}, auth^{hnd})$ $\{z_1 \approx k_e, k_a, ck\}$
34. $s_2^{hnd} \leftarrow \mathsf{sign}(ske_K^{hnd}, z_1^{hnd})$ $\{s_2 \approx [k_e, k_a, ck]_{sk_K}\}$
35. $z_2^{hnd} \leftarrow \mathsf{list}(cert_K^{hnd}, s_2^{hnd})$ $\{z_2 \approx Cert_K, [k_e, k_a, ck]_{sk_K}\}$
36. $m_{21} \leftarrow \mathsf{encrypt}(pke_v^{hnd}, z_2^{hnd})$ $\{m_{21} \approx \{\!\{Cert_K, [k_e, k_a, ck]_{sk_K}\}\!\}_{pk_C}\}$
37. $z_3^{hnd} \leftarrow \mathsf{list}(AK^{hnd}, x_3^{hnd}, t_K^{hnd})$ $\{z_3 \approx AK, C, t_K, T\}$
38. $TGT^{hnd} \leftarrow \mathsf{sym\_encrypt}(skse_{K,x_4}^{hnd}, z_3^{hnd})$ $\{TGT \approx \{AK, C, t_K\}_{k_T}\}$
39. $z_4^{hnd} \leftarrow \mathsf{list}(AK^{hnd}, x_5^{hnd}, t_K^{hnd}, x_4^{hnd})$ $\{z_4 \approx AK, n_1, t_K, T\}$
40. $m_{24} \leftarrow \mathsf{sym\_encrypt}(k_e^{hnd}, z_4^{hnd})$ $m_{24} \approx \{Ak, n_1, t_K, T\}_{k_e}\}$
41. $m_2^{hnd} \leftarrow \mathsf{list}(m_{21}^{hnd}, x_3^{hnd}, TGT^{hnd}, m_{24}^{hnd})$ $\{m_2 \approx \{\!\{Cert_K, [k_e, k_a, ck]_{sk_K}\}\!\}_{pk_C}, C, TGT, \{Ak, n_1, t_K, T\}_{k_e}\}$
42. $\mathsf{send\_i}(v, m_2^{hnd})$

Fig. 8 Algorithm 2 of Public-key Kerberos : Behavior of the KAS

the input message m's components are checked in steps 2.1–2.5. The command $\mathsf{retrieve}(x_i^{hnd})$ in step 2.3 returns the bitstring of the entry $D[hnd_u = x_i^{hnd}]$. Next the machine verifies the received certificate x_1 of v by checking the signature of the certificate authority CA (steps 2.6–2.10). Then the machine extracts the public key w_2 and v's name out of the certificate (steps 2.12–16) and uses this public key to verify the signature x_2 received in the AS_REQ (steps 2.18–2.21). In steps 2.23–2.26 the types of the message components of the signed message y_1

are checked, as well as the freshness of the nonce y_{12} by comparison to nonces stored in $Nonce3_K$. If the nonce is fresh then it will be stored in the set $Nonce3_K$ in step 2.28 for freshness checks in future protocol runs. Finally, in steps 2.29–2.42 $\mathrm{M}_K^{\mathrm{PK}}$ generates symmetric keys k_e, k_a, and AK, composes the AS_REP, and sends it to v over an insecure channel.

Note: Unlike in the symbolic model, one cannot use the same key for the use in two different cryptographic primitives in the computational model, e.g., for symmetric encryption and within a message authentication code. Otherwise the security guarantees of the cryptographic primitives may no longer hold. This needs to be considered when working with computationally sound symbolic frameworks like the BPW model. Therefore Algorithm 2 is generating a key pair, consisting of a symmetric encryption key k_e and an message authentication key k_a, instead of a single symmetric key (which is denoted by k in Fig. 2).

4 Formal Results

We will now prove that the fragments of Kerberos 5 discussed earlier possess the properties informally outlined in Section 2. We begin by formalizing the respective security properties and verify them properties in the BPW model in Section 4.1. Then, in Section 4.2, we make use of previous work to transfer the authentication aspect of these results to the computational setting, and we discuss the notion of computational secrecy.

4.1 Security in the Symbolic Setting

In order to use the BPW model to prove the computational security of Kerberos, we first formalize the respective security properties and verify them in the BPW model. We prove that Kerberos keeps the symmetric key, which the TGS T generated for use between user u and server S, symbolically secret from the adversary. In order to prove this, we show that Kerberos also keeps the keys generated by KAS K for the use between u and the TGS T secret. Furthermore, we prove entity authentication of the user u to a server S (and subsequently entity authentication of S to u). This form of authentication is weaker than the authentication Kerberos offers, because we do not consider the purpose of timestamps in Kerberos. Timestamps are currently not modeled in the BPW model.

Secrecy and Authentication Requirements Next we define the notion of key secrecy, which was formally captured already in Property 1 of Section 2, as the following formal requirement in the language of the BPW model.

Definition 1 (Key secrecy requirement)
For Kerb $\in${PK, K5} the secrecy requirement $Req^{\mathsf{Sec}}_{Kerb}$ is:

For all $u \in \mathcal{H} \cap \{1, \dots, n\}$, and $S \in \mathcal{H} \cap \{S_1, \dots, S_l\}$, and $t_1, t_2, t_3 \in \mathbb{N}$:

$$\begin{aligned} &(t_1 : KA\_out_S!\,(\mathsf{ok}, \mathsf{Kerb}, u, SK^{hnd}) \\ \vee\ & t_2 : KA\_out_u!\,(\mathsf{ok}, \mathsf{Kerb}, S, SK^{hnd}) \\ \Rightarrow\ & t_3 : D[hnd_u = SK^{hnd}].hnd_a = \downarrow \end{aligned}$$

where $t : D$ denotes the contents of database D at time t. Similarly $t : p?m$ and $t : p!m$ denotes that message m occurs at input (respectively output) port p at time t. As above, PK refers to Public-key Kerberos and K5 to Kerberos 5. In the next section, Theorem 1 will show that the symbolic Kerberos systems specified in Section 3.2 satisfy this notion of secrecy, and therefore Kerberos enjoys Property 1.

Next we define the notion of authentication in Property 2 in the language of the BPW model.

Definition 2 (Authentication requirements) For $Kerb \in \{PK, K5\}$:

i. The authentication requirement $Req^{\mathsf{Auth1}}_{Kerb}$ is: For all $v \in \mathcal{H} \cap \{1, \dots, n\}$, for all $S \in \mathcal{H} \cap \{S_1, \dots, S_l\}$, and K, T:

$$\begin{aligned} &\exists\, t_3 \in \mathbb{N}.t_3\ :\ KA\_out_S!\,(\mathsf{ok}, \mathsf{Kerb}, v, SK^{hnd}) \\ &\Rightarrow \\ &\exists\, t_1, t_2 \in \mathbb{N}\ with\ t_1 < t_2 < t_3. \\ &t_2\ :\ KA\_in_v!\,(\mathsf{continue\_prot}, \mathsf{Kerb},\, T, S, \cdot) \\ &\qquad \wedge\, t_1\ :\ KA\_in_v!\,(\mathsf{new\_prot}, \mathsf{Kerb}, K, T) \end{aligned}$$

ii. The authentication requirement $Req^{\mathsf{Auth2}}_{Kerb}$ is: For all $u \in \mathcal{H} \cap \{1, \dots, n\}$, for all $S \in \mathcal{H} \cap \{S_1, \dots, S_l\}$, and K, T:

$$\begin{aligned} &\exists\, t_2 \in \mathbb{N}.t_2\ :\ KA\_out_u!\,(\mathsf{ok}, \mathsf{Kerb}, S, SK^{hnd}) \\ &\Rightarrow \\ &\exists\, t_1 \in \mathbb{N}\ with\ t_1 < t_2. \\ &\qquad t_1\ :\ KA\_out_S!\,(\mathsf{ok}, \mathsf{Kerb}, u, SK^{hnd}) \end{aligned}$$

iii. The overall authentication $Req^{\mathsf{Auth}}_{Kerb}$ for the protocol $Kerb$ is:

$$Req^{\mathsf{Auth}}_{Kerb} := Req^{\mathsf{Auth1}}_{Kerb} \wedge Req^{\mathsf{Auth2}}_{Kerb}$$

Theorem 1 will show that this notion of authentication is satisfied by the symbolic Kerberos system. Therefore Kerberos has Property 2.

When proving that Kerberos has these properties, we will use the notion of a system *Sys perfectly fulfilling a requirement Req*, denoted $Sys \models^{\mathrm{perf}} Req$. This means the property *Req* holds with probability 1 over the probability space of runs for a fixed security parameter (as defined in Sect. 3.1). Later we will also need the notion of a system *Sys computationally fulfilling a requirement Req*, denoted $Sys \models^{\mathrm{poly}} Req$; this means the property holds with negligible error probability for all polynomially bounded users and adversaries (again, over the probability space of all runs for a fixed security parameter). In particular, perfect fulfillment implies computational fulfillment.

In order to prove Theorem 1, we first need to prove a number of auxiliary properties (previously called *invariants* in, e.g., [4,10]). Although these properties are nearly identical for Kerberos 5 and Public-key Kerberos, their proofs had to be carried out separately. We consider it interesting future work to augment the BPW model with proof techniques that allow for conveniently analyzing security protocols in a more modular manner. In fact, a higher degree of modularity would simplify the proofs for each individual protocol as it could exploit the highly modular structure of Kerberos; moreover, it would also simplify the treatment of the numerous optional behaviors of this protocol.

Some of the key properties needed in the proof of Theorem 1, which formalizes Properties 1 and 2, make authentication and confidentiality statements for the first two rounds of Kerberos. These properties are described in English below and formalized and proved in Appendix B.

i) Authentication of KAS to client and Secrecy of AK: If user u receives a valid AS_REP message then this message was indeed generated by K for u and an adversary cannot learn the contained symmetric keys.
ii) TGS Authentication of the TGT: If TGS T receives a TGT and an authenticator $\{v, t_v\}_{AK}$ where the key AK and the username v are contained in the TGT, then the TGT was generated by K and the authenticator was created by v.
iii) Authentication of TGS to client and Secrecy of SK: If user u receives a valid TGS_REP then it was generated by T for u and S. And an adversary cannot learn the contained session key SK.
iv) Server Authentication of the ST: If server S receives a ST and an authenticator $\{v, t_v\}_{SK}$ where the key SK and the name v are contained in the ST, then the ST was generated by T and the authenticator was created by v.

We can now capture the security of Kerberos in the BPW model in the following theorem, which says that Properties 1 and 2 hold symbolically for Kerberos. We show a proof excerpt in the case of Public-key Kerberos (the outline is analogous for Kerberos 5). The full proofs can be found in Appendix B.4 and B.6.

Theorem 1 *(Security of the Kerberos Protocol based on the BPW Model)*

- *Let* $Sys^{\mathrm{K5,\ symb}}$ *be the symbolic Kerberos 5 system defined in Section 3.2, and let* Req^{Sec}_{K5} *and* Req^{Auth}_{K5} *be the secrecy and authentication requirements defined above. Then*

$$Sys^{\mathrm{K5,\ symb}} \models^{\mathrm{perf}} Req^{\mathsf{Sec}}_{K5} \wedge Req^{\mathsf{Auth}}_{K5}.$$

- *Let* $Sys^{\mathrm{PK,\ symb}}$ *be the symbolic Public-key Kerberos system, and let* Req^{Sec}_{PK} *and* Req^{Auth}_{PK} *be the secrecy and authentication requirements defined above. Then*

$$Sys^{\mathrm{PK,\ symb}} \models^{\mathrm{perf}} Req^{\mathsf{Sec}}_{PK} \wedge Req^{\mathsf{Auth}}_{PK}.$$

Proof (sketch) We assume that all parties are honest. If user u successfully terminates a session run with a server S, i.e., there was an output (ok, PK, S, k^{hnd}) at $\mathrm{KA\_out}_u!$, then we know that the key k was stored in the set $Session\_KeysS_u$. This implies that the key was generated by T and sent to u in a valid TGS_REP. By auxiliary property iv) mentioned above, an adversary cannot learn k. A similar argument holds for the case that S successfully terminates a session run. This shows the key secrecy property Req^{Sec}_{PK}. As for the authentication property $Req^{\mathsf{Auth1}}_{PK}$, if server S successfully terminates a session with u, i.e., there was an output (ok, PK, u, k^{hnd}) at $\mathrm{KA\_out}_S!$, then S must have received a ticket generated by T (for S and u) and also a matching authenticator generated by user u (by auxiliary property iv)). But the ticket will only be generated if u sends the appropriate request to T, i.e., there was an input ($\mathsf{continue\_prot}$, PK, T, S, AK^{hnd}) at $\mathrm{KA\_}in_u?$. The request, on the other hand, contains a TGT that was generated by K for u (by auxiliary property ii)), therefore u must have sent an request to K. In particular, there had been an input ($\mathsf{new\_prot}$, PK, K, T) at $\mathrm{KA\_}in_u?$. As for the authentication property $Req^{\mathsf{Auth2}}_{PK}$, if the user u successfully terminates a session with server S, i.e., there was an output (ok, PK, S, k^{hnd}) at $\mathrm{KA\_out}_u!$, then it must have received a message encrypted under k that does not contain u's name. The key k was contained in a valid TGS_REP and was therefore generated by T, by auxiliary property iii). Only T, u, or S could know the key k, but only S uses this key to encrypt and send a message that u received. On the other hand, S follows sending such a message immediately by an output (ok, PK, u, k^{hnd}) at $\mathrm{KA\_out}_S!$. □

This proof shares similarities with the Dolev–Yao style proofs of analogous results attained for Kerberos 5 and

PKINIT using the MSR framework [20, 21, 24–26]. The two approaches are similar in the sense that both reconstruct a necessary trace backward from an end state, and in that they rely on some form of induction (based on rank/co-rank functions in MSR). An intriguing problem for future work is a formal comparison between these two Dolev–Yao encodings of a protocol and between the proof techniques they support.

4.2 Security in the Cryptographic Setting

The results of [13] allow us to take the authentication results in Theorem 1 and derive a corresponding authentication results for a cryptographic implementation of Kerberos. Just as Property 2 holds symbolically for Kerberos, this shows that it holds in a cryptographic implementation as well. In particular, entity authentication between a user and a server in Kerberos holds with overwhelming probability (over the probability space of runs). However, symbolic results on key secrecy can only be carried over to cryptographic implementations if the protocol satisfies certain additional conditions. Kerberos unfortunately does not fulfill these definitions, and it can easily be shown that cryptographic implementations of Kerberos do not fulfill the standard notion of cryptographic key secrecy, see below. With regard to authentication, the following does hold.

Theorem 2 *(Computational security of the Kerberos protocol)*

– *Let* $Sys^{\mathrm{K5,\ comp}}$ *denote the computational basic Kerberos system implemented with provably secure cryptographic primitives. Then*

$$Sys^{\mathrm{K5,\ comp}} \models^{\mathrm{poly}} Req^{\mathsf{Auth}}_{K5}.$$

– *Let* $Sys^{\mathrm{PK,\ comp}}$ *denote the computational Public-key Kerberos system implemented with provably secure cryptographic primitives. Then*

$$Sys^{\mathrm{PK,\ comp}} \models^{\mathrm{poly}} Req^{\mathsf{Auth}}_{PK}.$$

Proof (Sketch for Public-key Kerberos) By Theorem 1, we know that $Sys^{PK,\ id} \models^{perf} Req^{\mathsf{Auth}}_{PK}$. And, as we mentioned earlier, the cryptographic implementation of the BPW model (using provably secure cryptographic primitives) is at least as secure as the BPW model, $Sys^{\mathrm{cry,\ comp}} \geq^{\mathrm{poly}}_{\mathrm{sec}} Sys^{\mathrm{cry,\ id}}$.

The requirements Req^{Auth}_{K5} and Req^{Auth}_{PK}, which are defined above, are *integrity properties* as defined in Definition 2 of [6]. Because of the polynomial bounds on message length and number of inputs, it is decidable in polynomial time whether a run satisfies these requirements. Thus, we may prove that $Sys^{\mathrm{K5,\ comp}} \models^{poly} Req^{\mathsf{Auth}}_{K5}$ and that $Sys^{\mathrm{PK,\ comp}} \models^{poly} Req^{\mathsf{Auth}}_{PK}$ by applying Theorem 1 (Conservation of Integrity Properties) of [6] to our Theorem 1 above, so long as these protocols do not have the "Commitment Problem."

The Commitment Problem occurs when keys that have been used for cryptographic work while being at that time only known to honest users are revealed to the adversary later in the protocol. If the simulator in [13] (with which one can simulate a computational adversary attack on the symbolic system) learns in some abstract way that, e.g., a cyphertext was sent, the simulator generates a distinguishable cyphertext without knowing the symmetric key nor the plaintext. If the symmetric key is revealed later in the protocol then the trouble for the simulator will be to generate a suitable symmetric key that decrypts the cyphertext into the correct plaintext. This is typically an impossible task. In order for the simulation with the BPW model to work, one thus needs to check that the Commitment Problem does not occur in the protocol (and for Kerberos, it fortunately doesn't; see Lemma 1 of App. B.1).

We may thus invoke the Conservation of Integrity Properties (Theorem 1 of [6]) to obtain Theorem 2. □

As far as key secrecy is concerned, it can be proven that the adversary attacking the cryptographic implementation does not learn the secret key string as a whole. However, it does not necessarily rule out that an adversary will be able to distinguish the key from other fresh random keys, as required by the definition of *cryptographic key secrecy*. This definition of secrecy says that an adversary cannot learn any partial information about such a key and is hence considerably stronger than requiring that an adversary cannot obtain the whole key. For Kerberos we can show that the key SK does not satisfy cryptographic key secrecy after the last round of Kerberos, i.e., SK is distinguishable from other fresh random keys. It should also be noted that this key SK is still indistinguishable from random after the second round but before the start of the third round of Kerberos. We have the following proposition

Proposition 1 *a) Kerberos does not provide cryptographic key secrecy for the key* SK *generated by the TGS* T *for the use between client* C *and server* S *after the start of the last round of Kerberos.*

b) After the TGS exchange and before the start of the CS exchange is the key SK *generated by the TGS* T *still cryptographically secret.*

Proof a) To see that Kerberos does not offer cryptographic key secrecy for SK after the start of the third round, note that the key SK is used in the protocol for symmetric encryption. As symmetric encryption always provides partial information to an adversary if the adversary also knows the message that was encrypted. An adversary can exploit this to distinguish the key SK as follows: the adversary first completes a regular Kerberos execution between C and S learning the message $\{C, t'\}_{SK}$ encrypted under the unknown key SK. The adversary will also learn a bounded time period TP (of a few seconds) in which the timestamp t' was generated. Next a bit b is flipped and the adversary receives a key k, where $k = SK$ for $b = 0$ and k is a fresh random key for $b = 1$. The adversary now attempts to decrypt $\{C, t'\}_{SK}$ with k yielding a message m. If $m \neq C, t$ for a timestamp t then the adversary guesses $b = 1$. If $m = C, t$ for a timestamp t, then the adversary checks whether $t \in TP$ or not. If $t \notin TP$ then the adversary guesses $b = 1$, otherwise the adversary guesses $b = 0$. The probability of the adversary guessing correctly is then $1 - \epsilon$, where ϵ is the probability that for random keys k, SK the cyphertext $\{C, t'\}_{SK}$ decrypted with k is C, t with $t \in TP$. Clearly, ϵ is negligible (since the length of the time period TP does not depend on the security parameter). Hence, SK is distinguishable and cryptographic key secrecy does not hold.

b) However, before the third round has been started the key SK is not only unknown to the adversary but, in particular, SK has not been used for symmetric encryption yet. We can therefore invoke the key secrecy preservation theorem of [9], which states that a key that is symbolically secret and symbolically unused is also cryptographically secret. This allows us to conclude that SK is cryptographically secret from the adversary.

For similar reasons, we also have the next proposition

Proposition 2 *a) Kerberos does not provide cryptographic key secrecy for the key AK generated by the KAS K for the use between client C and TGS T after the start of the second round of Kerberos.*

b) After the AS exchange and before the start of the TGS exchange is the key AK generated by the KAS K still cryptographic secret.

Optional Sub-Session Key Kerberos may allow the client or the server to generate a sub-session key. This optional key can then be used for the encryption of further communication between the two parties. To send the optional sub-session key to the other party, the generator of this optional key (C or S) includes the key as part of the message which is encrypted using the session key SK. For instance, server S may generate the optional key k and send $\{t', k\}_{SK}$ as the AP_REP. It is easy to see that, due to the key secrecy of SK, an adversary cannot learn the optional key (i.e., in the language of the BPW model, an adversary does not get a handle to this key). Since the optional key is not used in the protocol, we may invoke Theorem IV.1 of [9]. This theorem says that unused keys, which the adversary cannot learn, are kept cryptographically secret by the protocol. This approach is illustrated for the Yahalom protocol in [10].

Corollary 1 *(Computational security of the optional sub-session) Let Kerb $\in \{PK, K5\}$ be fixed. Then the symbolic Kerberos system $Sys^{\text{Kerb, id}}$ from Section 3.2 keeps the optional sub-session key symbolically secret, and all polynomial-time configurations of the computational Public-key Kerberos system $Sys^{\text{Kerb, comp}}$ keep the optional sub-session key cryptographically secret.*

5 Conclusions and Future Work

In this paper, we have exploited the Dolev–Yao style model of Backes, Pfitzmann, and Waidner [8, 11, 12] to obtain the first computational proof of authentication for the core exchanges of the Kerberos protocol and its extension to public keys (PKINIT). Although the proofs sketched here are conducted symbolically, grounding the analysis on the BPW model automatically lifts the results to the computational level, assuming that all cryptography is implemented using provably secure primitives. We could establish cryptographic key secrecy (in the sense of indistinguishability of the exchanged key from a random key) only for the optional sub-key exchanged in Kerberos; for the actually exchanged key, we could not prove cryptographic key secrecy.

Concerning future work, we plan to investigate if the algorithms that are supported by PKINIT [55] satisfy the cryptographic assumptions of our proofs. For the symmetric encryption scheme, which is also used in basic Keberos, this has been done in [19]. It remains an open question whether the supported public-key encryption schemes, the digital signature schemes and the checksum algorithms meet the cryptographic assumptions we make in this work.

Furthermore, it seems promising to augment the BPW model with specialized proof techniques that allow for conveniently performing proofs in a modular manner. Such techniques would provide a simple and elegant way to integrate the numerous optional behaviors supported by Kerberos and nearly all commercial protocols; for example, this would facilitate the analysis of DH mode in PKINIT which is part of our ongoing work. We intend to tackle the invention of such proof techniques that are specifically tailored towards the BPW model, e.g., by exploiting recent ideas from [31]. Another potential improvement is to augment the BPW model with timestamps; this would in particular allow us to establish authentication properties that go beyond entity authentication [20,21,24–27]. An additional item on our research agenda is to fully understand the relation between the symbolic correctness proof for Kerberos 5 presented here and the corresponding results achieved in the MSR framework [20, 21, 24–26].

Acknowledgements. We would like to thank John Mitchell, Anupam Datta, Ante Derek, and Arnab Roy for helpful discussions.

References

1. The AVISPA tool for the automated validation of internet security protocols and applications. In *Proc. Computer-aided Verification (CAV)*. Springer, 2005. URL: `www.avispa-project.org`.
2. M. Abadi and J. Jürjens. Formal eavesdropping and its computational interpretation. In *Proc. TACS*, pages 82–94, 2001.
3. M. Abadi and P. Rogaway. Reconciling two views of cryptography: The computational soundness of formal encryption. In *Proc. 1st IFIP International Conference on Theoretical Computer Science*, pages 3–22. Springer LNCS 1872, 2000.
4. M. Backes. A cryptographically sound Dolev-Yao style security proof of the Otway-Rees protocol. In *Proc. ESORICS*, pages 89–108. Springer LNCS 3193, 2004.
5. 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*, pages 362–383, 2006.
6. M. Backes and C. Jacobi. Cryptographically sound and machine-assisted verification of security protocols. In *Proc. 20th STACS*, pages 675–686. Springer LNCS 2607, 2003.
7. M. Backes and B. Pfitzmann. A cryptographically sound security proof of the Needham-Schroeder-Lowe public-key protocol. *Journal on Selected Areas in Communications*, 22(10):2075–2086, 2004.
8. M. Backes and B. Pfitzmann. Symmetric encryption in a simulatable Dolev-Yao style cryptographic library. In *Proc. CSFW'04*, pages 204–218, June 2004.
9. M. Backes and B. Pfitzmann. Relating symbolic and cryptographic secrecy. *IEEE Trans. Dependable Secure Comp.*, 2(2):109–123, April–June 2005.
10. M. Backes and B. Pfitzmann. On the cryptographic key secrecy of the strengthened Yahalom protocol. In *Proceedings of 21st IFIP International Information Security Conference (SEC)*, pages 233–245, May 2006.
11. M. Backes, B. Pfitzmann, and M. Waidner. A composable cryptographic library with nested operations (extended abstract). In *Proc. CCS'03*, pages 220–230, 2003.
12. M. Backes, B. Pfitzmann, and M. Waidner. Symmetric authentication within a simulatable cryptographic library. In *Proc. ESORICS'03*, pages 271–290. Springer LNCS 2808, 2003.
13. M. Backes, B. Pfitzmann, and M. Waidner. A universally composable cryptographic library. IACR Cryptology ePrint Archive, Report 2003/015, `http://eprint.iacr.org/`, January 2003.
14. G. Bella and L. C. Paulson. Kerberos Version IV: Inductive Analysis of the Secrecy Goals. In *Proc. ESORICS'98*, pages 361–375. Springer LNCS 1485, 1998.
15. G. Bella and E. Riccobene. Formal Analysis of the Kerberos Authentication System. *J. Universal Comp. Sci.*, 3(12):1337–1381, December 1997.
16. M. Bellare and P. Rogaway. Entity authentication and key distribution. In *Proc. CRYPTO '93*, pages 232–249. Springer LNCS 773, 1994.
17. B. Blanchet. A computationally sound mechanized prover for security protocols. In *Proc. 27th IEEE Symposium on Security & Privacy*, 2006.
18. B. Blanchet, A. D. Jaggard, A. Scedrov, and J.-K. Tsay. Computationally sound mechanized proofs for basic and public-key kerberos. In *ASIACCS'08*, pages 87–99, 2008.
19. A. Boldyreva and V. Kumar. Provable-security analysis of authenticated encryption in Kerberos. In *IEEE Symposium on Security and Privacy*, 2007.
20. F. Butler, I. Cervesato, A. D. Jaggard, and A. Scedrov. An Analysis of Some Properties of Kerberos 5 Using MSR. In *Proc. CSFW'02*, 2002.
21. F. Butler, I. Cervesato, A. D. Jaggard, A. Scedrov, and C. Walstad. Formal Analysis of Kerberos 5. *Theoretical Computer Science*, 367(1-2):57–87, November 2006.
22. Cable Television Laboratories, Inc. PacketCable Security Specification, 2004. Technical document PKT-SP-SEC-I11-040730.
23. R. Canetti and J. Herzog. Universally composable symbolic analysis of cryptographic protocols (the case of encryption-based mutual authentication and key exchange). In *Proc. 3rd Theory of Cryptography Conference (TCC)*, 2006.
24. I. Cervesato, A. D. Jaggard, A. Scedrov, J.-K. Tsay, and C. Walstad. Breaking and fixing public-key Kerberos, 2006. Presented at WITS'06.
25. I. Cervesato, A. D. Jaggard, A. Scedrov, J.-K. Tsay, and C. Walstad. Breaking and fixing public-key Kerberos. In *Proc. ASIAN'06, LNCS 4435*, 2006.
26. I. Cervesato, A. D. Jaggard, A. Scedrov, J.-K. Tsay, and C. Walstad. Breaking and fixing public-key Kerberos. *Inf. Comput.*, 206(2–4):402–424, 2008.
27. I. Cervesato, A. D. Jaggard, A. Scedrov, and C. Walstad. Specifying Kerberos 5 Cross-Realm Authentication. In *Proc. WITS'05*, pages 12–26, 2005.
28. I. Cervesato, C. Meadows, and D. Pavlovic. An Encapsulated Authentication Logic for Reasoning About Key Distribution Protocol. In *Eighteenth Computer Security Foundations Workshop — CSFW-18*, pages 48–61, Aix-en-Provence, France, 20–22 June 2005. IEEE Computer Society Press.
29. V. Cortier and B. Warinschi. Computationally sound, automated proofs for security protocols. In *Proc. ESOP-14*, pages 157–171, 2005.
30. A. Datta, A. Derek, J. Mitchell, V. Shmatikov, and M. Turuani. Probabilistic polynomial-time semantics for a protocol security logic. In *Proc. ICALP*, pages 16–29. Springer LNCS 3580, 2005.
31. A. Datta, A. Derek, J. Mitchell, and B. Warinschi. Key exchange protocols: Security definition, proof method, and applications. In *19th IEEE Computer Security Foundations Workshop (CSFW 19)*, Venice, Italy, 2006. IEEE Press.
32. J. De Clercq and M. Balladelli. Windows 2000 authentication. `http://www.windowsitlibrary.com/Content/617/06/6.html`, 2001. Digital Press.
33. D. Dolev and A. Yao. On the security of public-key protocols. *IEEE Trans. Info. Theory*, 2(29):198–208, 1983.
34. S. Gajek, M. Manulis, O. Pereira, A.-R. Sadeghi, and J. Schwenk. Universally Composable Security Analysis of TLS. In *Proceedings of the 2nd International Conference on Provable Security (ProvSec 2008)*, volume 5324 of *Lecture Notes in Computer Science*, pages 313–327. Springer, 2008.

35. O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game – or – a completeness theorem for protocols with honest majority. In *Proc. STOC*, pages 218–229, 1987.
36. S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28:270–299, 1984.
37. J. D. Guttman, F. J. Thayer Fabrega, and L. Zuck. The faithfulness of abstract protocol analysis: Message authentication. In *Proc. CCS-8*, pages 186–195, 2001.
38. C. He and J. C. Mitchell. Security Analysis and Improvements for IEEE 802.11i. In *Proc. NDSS'05*, 2005.
39. J. Herzog, M. Liskov, and S. Micali. Plaintext awareness via key registration. In *Proc. CRYPTO*, pages 548–564. Springer LNCS 2729, 2003.
40. IETF. Public Key Cryptography for Initial Authentication in Kerberos, 1996–2006. Sequence of Internet drafts available from `http://tools.ietf.org/wg/krb-wg/draft-ietf-cat-kerberos-pk-init/`.
41. R. Impagliazzo and B. M. Kapron. Logics for reasoning about cryptographic constructions. In *Proc. FOCS*, pages 372–381, 2003.
42. P. Laud. Semantics and program analysis of computationally secure information flow. In *Proc. ESOP*, pages 77–91, 2001.
43. P. Laud. Symmetric encryption in automatic analyses for confidentiality against active adversaries. In *Proc. Symp. Security and Privacy*, pages 71–85, 2004.
44. C. Meadows. Analysis of the internet key exchange protocol using the NRL Protocol Analyzer. In *Proc. IEEE Symp. Security and Privacy*, pages 216–231, 1999.
45. D. Micciancio and B. Warinschi. Soundness of formal encryption in the presence of active adversaries. In *Proc. TCC*, pages 133–151. Springer LNCS 2951, 2004.
46. Microsoft. Security Bulletin MS05-042. `http://www.microsoft.com/technet/security/bulletin/MS05-042.mspx`, Aug. 2005.
47. J. Mitchell, M. Mitchell, and A. Scedrov. A linguistic characterization of bounded oracle computation and probabilistic polynomial time. In *Proc. FOCS*, pages 725–733, 1998.
48. 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.
49. C. Neuman and T. Ts'o. Kerberos: An Authentication Service for Computer Networks. *IEEE Communications*, 32(9):33–38, Sept. 1994.
50. C. Neuman, T. Yu, S. Hartman, and K. Raeburn. The Kerberos Network Authentication Service (V5), July 2005. `http://www.ietf.org/rfc/rfc4120`.
51. B. Pfitzmann and M. Waidner. A model for asynchronous reactive systems and its application to secure message transmission. In *Proc. S&P*, pages 184–200, 2001.
52. A. Roy, A. Datta, A. Derek, and J. C. Mitchell. Inductive proofs of computational secrecy. In J. Biskup and J. Lopez, editors, *ESORICS*, volume 4734 of *Lecture Notes in Computer Science*, pages 219–234. Springer, 2007.
53. A. Roy, A. Datta, and J. C. Mitchell. Formal proofs of cryptographic security of Diffie-Hellman-based protocols. In G. Barthe and C. Fournet, editors, *TGC*, volume 4912 of *Lecture Notes in Computer Science*, pages 312–329. Springer, 2007.
54. The Internet Engineering Task Force. `http://www.ietf.org`.
55. L. Zhu and B. Tung. Public Key Cryptography for Initial Authentication in Kerberos (PKINIT), June 2006. `http://www.ietf.org/rfc/rfc4556`.

A Additional Algorithms

For completeness, this appendix collects the algorithms omitted from the main body of this paper. The algorithms for public-key Kerberos are in Figures 7–12. The algorithms for Kerberos 5 are in Figures 11–15. Note that the algorithms for the TGS T and a server S (i.e., Algorithms 4 and 5 in Fig. 11 and 12) are identical for Public-key Kerberos and Kerberos 5.

B Additional Proofs

B.1 Absence of the Commitment Problem

For the proof of Theorem 2 we need to show that the Commitment Problem does not occur for $Sys^{PK,\ id}$ or $Sys^{K5,\ id}$. As in [8], let NoComm be the following property: "*If there exists an input from an honest user that causes a symmetric encryption to be generated such that the corresponding key is not known to the adversary, then future inputs may only cause this key to be sent within an encryption that cannot be decrypted by the adversary*".

Lemma 1 *Absence of the Commitment Problem The ideal Kerberos* $Sys^{\mathrm{Kerb,\ id}}$*, for* $Kerb \in \{PK, K5\}$*, perfectly fulfills the property* NoComm*, i.e.,* $Sys^{\mathrm{Kerb,\ id}} \models^{\mathrm{perf}}$ NoComm.

Proof Note first that the long-term symmetric keys shared between KAS and TGS, TGS and server, and, in the case of basic Kerberos, between user and KAS will never be sent if the user, the TGS and the server are honest. Therefore, we are left to verify that the Commitment Problem does not occur for the keys generated during a protocol run. In the case of Public-key Kerberos: Say $i \le size$, $D[j].type =$ skse such that $D[i]$ was created in steps 2.29, 2.30, 2.31 or 4.18. In both Algorithms 2 and 4, the keys are generated for $v \in \{1, \ldots, n\}$ by $\mathrm{M}_K^{\mathrm{PK}}$, respectively $\mathrm{M}_T^{\mathrm{PK}}$. If v and K, respectively v and T, are dishonest, then the adversary would get a handle to $D[i]$ right after the execution of $\mathrm{M}_K^{\mathrm{PK}}$, respectively $\mathrm{M}_T^{\mathrm{PK}}$, since the adversary knows the keys shared between dishonest parties. Note that the message sent at the end of the execution of $\mathrm{M}_K^{\mathrm{PK}}$, respectively $\mathrm{M}_T^{\mathrm{PK}}$ also contains a part that is encrypted using a handle to $D[i]$. However, this will not cause the simulator to encrypt with a arbitrary random key since it parses all messages completely before constructing the computational version bottom-up (as described in [8, 13]), i.e., the simulator will get a handle to $D[i]$ before constructing the cyphertext under $D[i]$. If u, K, T and S are honest, then the Key Secrecy property (Lemma 3, respectively Lemma 9) implies that for the keys created in steps 2.29, 2.30, 2.31 and 4.18, respectively in steps 2.9 and 4.18, one has $D[j].hnd_a = \downarrow$ for all time t. The argument for basic Kerberos is analogous.

B.2 Conventions

In the subsequent proofs we will use following convention for the algorithms:

Convention 1 *Let* $\mathrm{Kerb} \in \{\mathrm{PK}, \mathrm{K5}\}$. *For all* $w \in \{1, \ldots, n\} \cup \{S_1, \ldots, S_l\} \cup \{K, T\}$ *the following holds. If* M_w^{Kerb} *enters a command at port* $\mathrm{in}_w!$ *and receives* $\downarrow$ *at port* $\mathrm{out}_w?$ *as the immediate answer from* $TH_{\mathcal{H}}$*, then* M_w^{Kerb} *aborts the execution of the current algorithm, except if the command was of the form* list_proj *or* send_i.

Input: (v, u, i, m^{hnd}) at $\text{out}_u?$

1. **if** $v = K$ **then** $\{$AS_REP is input$\}$
2. $c_i^{hnd} \leftarrow \mathsf{list\_proj}(m^{hnd}, i)$ for $i = 1, 2, 3, 4$ $\{c_1 \approx \{\{Cert_K, [k_e, k_a, ck]_{sk_K}\}\}_{pk_C}, c_2 \approx C, c_3 \approx TGT, c_4 \approx \{AK, n_1, t_K, T\}_{k_e}\}$
3. $c_2 \leftarrow \mathsf{retrieve}(c_2^{hnd})$
4. **if** $c_2 \neq u$ **then**
5. Abort
6. **end if**
7. $l_1^{hnd} \leftarrow \mathsf{decrypt}(ske_u^{hnd}, c_1^{hnd})$ $\{l_1 \approx Cert_K, [k_e, k_a, ck]_{sk_K}\}$
8. $l_{1.i}^{hnd} \leftarrow \mathsf{list\_proj}(l_1^{hnd}, i)$ for $i = 1, 2$ $\{l_{1.1} \approx Cert_K, l_{1.2} \approx [k_e, k_a, ck]_{sk_K}\}$
9. $z_1^{hnd} \leftarrow \mathsf{msg\_of\_sig}(l_{1.1}^{hnd})$
10. $b \leftarrow \mathsf{verify}(l_{1.1}^{hnd}, pke_{CA}^{hnd}, z_1^{hnd})$
11. **if** $b = false$ **then**
12. Abort
13. **end if**
14. $w_j^{hnd} \leftarrow \mathsf{list\_proj}(z_1^{hnd}, i)$ for $i = 1, 2$ $\{w_2 \approx pk_K\}$
15. $w_1 \leftarrow \mathsf{retrieve}(w_1^{hnd})$
16. $type_8 \leftarrow \mathsf{get\_type}(w_2^{hnd})$
17. **if** $(type_8 \neq \text{pke}) \vee (w_1 \neq K)$ **then**
18. Abort
19. **end if**
20. $z_{1.2}^{hnd} \leftarrow \mathsf{msg\_of\_sig}(l_{1.2})$ $\{z_{1.2} \approx k_e, k_a, ck\}$
21. $b \leftarrow \mathsf{verify}(l_{1.2}^{hnd}, w_2^{hnd}, z_{1.2}^{hnd})$
22. **if** $b = false$ **then**
23. Abort
24. **end if**
25. $x_i^{hnd} \leftarrow \mathsf{list\_proj}(z_{1.2}, i)$ for $i = 1, 2, 3$ $\{x_1 \approx k_e, x_2 \approx k_a, x_3 \approx ck\}$
26. $type_i \leftarrow \mathsf{get\_type}(x_i^{hnd})$ for $i = 1, 2, 3$
27. **if** $(type_1 \neq \text{skse}) \vee (type_2 \neq \text{ska}) \vee (type_3 \neq \text{auth})$ **then**
28. Abort
29. **end if**
30. $l_4^{hnd} \leftarrow \mathsf{sym\_decrypt}(x_1^{hnd}, c_4^{hnd})$ $\{x_1 \approx k_e, c_4 \approx \{AK, n_1, t_K, T\}_{k_e}\}$
31. $y_i^{hnd} \leftarrow \mathsf{list\_proj}(l_4^{hnd}), i$ for $i = 1, 2, 4$ $\{y_1 \approx AK, y_2 \approx n_1, y_4 \approx T\}$
32. $type_4 \leftarrow \mathsf{get\_type}(y_1^{hnd})$
33. $type_5 \leftarrow \mathsf{get\_type}(y_2^{hnd})$
34. $y_4 \leftarrow \mathsf{retrieve}(y_4^{hnd})$
35. **if** $(type_4 \neq \text{skse}) \vee (type_5 \neq \text{nonce}) \vee (y_4 \neq T) \vee (\nexists! \, \tilde{m}^{hnd} : (y_2^{hnd}, \tilde{m}^{hnd}) \in Nonce_u)$ **then**
36. Abort
37. **end if**
38. $b \leftarrow \mathsf{auth\_test}(x_3^{hnd}, x_2^{hnd}, \tilde{m}^{hnd})$ $\{x_3 \approx ck = H_k(\tilde{m})\}$
39. **if** $b = false$ **then**
40. Abort
41. **end if**
42. $TGTicket_u := TGTicket_u \cup \{(c_3^{hnd}, y_1^{hnd}, T)\}$ $\{c_3 \approx TGT, y_1 \approx AK\}$
43. output $(\mathsf{ok}, \mathsf{KAS\_exchange\ PK}, K, T, y_1^{hnd}, c_3^{hnd})$ at $\text{KA\_out}_u!$
44. **else if** $v = T$ **then** $\{$TGS_REP is input$\}$
45. $d_i^{hnd} \leftarrow \mathsf{list\_proj}(m^{hnd}, i)$ for $i = 1, 2, 3$ $\{d_1 \approx C, d_2 \approx ST, d_3 \approx \{SK, n_3, t_T, S\}_{AK}\}$
46. $d_1 \leftarrow \mathsf{retrieve}(d_1^{hnd})$
47. **if** $(d_1 \neq u)$ $\vee$ $(\nexists! \, (., AK^{hnd}, T) \in TGTicket_u : \mathsf{sym\_decrypt}(AK^{hnd}, d_3^{hnd}) \neq \downarrow)$ **then**
48. Abort
49. **end if**
50. $l_2^{hnd} \leftarrow \mathsf{sym\_decrypt}(AK^{hnd}, d_3^{hnd})$ $\{l_2 \approx SK, n_3, t_T, S\}$
51. $x_{2.i}^{hnd} \leftarrow \mathsf{list\_proj}(l_2^{hnd}, i)$ for $i = 1, 2, 4$ $\{x_{2.1} \approx SK, x_{2.2} \approx n_3, x_{2.4} \approx S\}$
52. $type_6 \leftarrow \mathsf{get\_type}(x_{2.1}^{hnd})$
53. $type_7 \leftarrow \mathsf{get\_type}(x_{2.2}^{hnd})$
54. $S \leftarrow retrieve(x_{2.4}^{hnd})$
55. **if** $(type_6 \neq \text{skse}) \vee (type_7 \neq \text{nonce}) \vee ((x_{2.2}^{hnd}, T, S) \notin Nonce2_u)$ **then**
56. Abort
57. **end if**
58. $x_5^{hnd} \leftarrow \mathsf{list}(u^{hnd}, t_u'^{hnd})$ $\{x_5 \approx C, t_C'\}$
59. $m_{5.2}^{hnd} \leftarrow \mathsf{sym\_encrypt}(x_{2.1}^{hnd}, x_5^{hnd})$ $\{m_{5.2} \approx \{C, t_C'\}_{SK}\}$
60. $m_5^{hnd} \leftarrow \mathsf{list}(d_2^{hnd}, m_{5.2}^{hnd})$ $\{m_5 \approx ST, \{C, t_C'\}_{SK}\}$
61. $Session\_KeysS_u := Session\_KeysS_u \cup \{(S, x_{2.1}^{hnd})\}$ $\{x_{2.1} \approx SK\}$
62. $\mathsf{send\_i}(S, m_5^{hnd})$
63. **else if** $v = S \in \{S_1, ..., S_l\}$ **then** $\{$AP_REP is input$\}$
64. **if** $(\nexists!(S, SK^{hnd}) \in Session\_KeysS_u : \mathsf{sym\_decrypt}(SK^{hnd}, m^{hnd}) \neq \downarrow)$ **then**
65. Abort
66. **end if**
67. $l_3^{hnd} \leftarrow \mathsf{sym\_decrypt}(SK^{hnd}, m^{hnd})$ $\{m \approx \{t_C'\}_{SK}\}$
68. $x_{3.1}^{hnd} \leftarrow \mathsf{list\_proj}(l_3^{hnd}, 1)$ $\{x_{3.1} \approx t_C'\}$
69. $x_{3.1} \leftarrow \mathsf{retrieve}(x_{3.1}^{hnd})$
70. **if** $x_{3.1} = u$ **then**
71. Abort
72. **end if**
73. output $(\mathsf{ok}, \mathsf{PK}, S, SK^{hnd})$ at $\text{KA\_out}_u!$

Fig. 9 Algorithm 3 of Public-key Kerberos, part 1: Behavior of user in after initialization

Fig. 10 Algorithm 3 of Public-key Kerberos, part 2: Behavior of user after initialization

Input: (v, T, i, m^{hnd}) at $\text{out}_T?$ with $v \in \{1, ..., n\}$.

1. $x_i^{hnd} \leftarrow \mathsf{list\_proj}(m^{hnd}, i)$ for $i = 1, 2, 3, 4, 5$ $\{x_1 \approx TGT, x_2 \approx \{C, t_C\}_{AK}, x_3 \approx C, x_4 \approx S, x_5 \approx n_3\}$
2. $y_1^{hnd} \leftarrow \mathsf{sym\_decrypt}(skse_{KT}^{hnd}, x_1^{hnd})$ $\{y_1 \approx AK, C, t_K\}$
3. $y_{1.i}^{hnd} \leftarrow \mathsf{list\_proj}(y_1^{hnd}, i)$ for $i = 1, 2$ $\{y_{1.1} \approx AK, y_{1.2} \approx C\}$
4. $type_1 \leftarrow \mathsf{get\_type}(y_{1.1}^{hnd})$
5. $type_2 \leftarrow \mathsf{get\_type}(x_5^{hnd})$
6. $x_i \leftarrow \mathsf{retrieve}(x_i^{hnd})$ for $i = 3, 4$
7. $y_{1.2} \leftarrow \mathsf{retrieve}(y_{1.2}^{hnd})$
8. **if** $(type_1 \neq \text{skse}) \vee (type_2 \neq \text{nonce}) \vee ((x_5^{hnd}, v) \in Nonce_T) \vee (x_3 \neq v) \vee (x_4 = S \notin \{S_1, ..., S_l\}) \vee (y_{1.2} \neq v)$**then**
9. Abort
10. **end if**
11. $Nonce4_T := Nonce4_T \cup \{(x_5^{hnd}, v)\}$
12. $z^{hnd} \leftarrow \mathsf{sym\_decrypt}(y_{1.1}^{hnd}, x_2^{hnd})$ $\{z \approx C, t_C\}$
13. $z_1^{hnd} \leftarrow \mathsf{list\_proj}(z^{hnd}, 1)$ $\{z_1 \approx C\}$
14. $z_1 \leftarrow \mathsf{retrieve}(z_1^{hnd})$
15. **if** $(z_1 \neq v)$ **then**
16. Abort
17. **end if**
18. $SK^{hnd} \leftarrow \mathsf{gen\_symenc\_key}()$
19. $l^{hnd} \leftarrow \mathsf{list}(SK^{hnd}, z_1^{hnd}, t_T^{hnd})$ $\{l \approx SK, C, t_T\}$
20. $ST^{hnd} \leftarrow \mathsf{sym\_encrypt}(skse_{TS}^{hnd}, l^{hnd})$ $\{ST \approx \{SK, C, t_T\}_{k_S}\}$
21. $\tilde{l}^{hnd} \leftarrow \mathsf{list}(SK^{hnd}, x_5^{hnd}, t_T^{hnd}, x_4^{hnd})$ $\{\tilde{l} \approx SK, n_3, t_T, S\}$
22. $m_{4.3}^{hnd} \leftarrow \mathsf{sym\_encrypt}(y_{1.1}^{hnd}, \tilde{l}^{hnd})$ $\{m_{4.3} \approx \{SK, n_3, t_T, S\}_{AK}\}$
23. $m_4^{hnd} \leftarrow \mathsf{list}(z_1^{hnd}, ST^{hnd}, m_{4.3}^{hnd})$ $\{m_4 \approx C, ST, \{SK, n_3, t_T, S\}_{AK}\}$
24. $\mathsf{send\_i}(v, m_4^{hnd})$

Fig. 11 Algorithm 4: Behavior of TGS

Input: (v, S, i, m^{hnd}) at $\text{out}_S?$ with $v \in \{1, ..., n\}$.

$m_{5.i}^{hnd} \leftarrow \mathsf{list\_proj}(m^{hnd}, i)$ for $i = 1, 2$ $\quad \{m_{5.1} \approx ST,\ m_{5.2} \approx \{C, t'_C\}_{SK}\}$
$x^{hnd} \leftarrow \mathsf{sym\_decrypt}(skse_{TS}^{hnd}, m_{5.1}^{hnd})$
$x_i^{hnd} \leftarrow \mathsf{list\_proj}(x^{hnd}, i)$ for $i = 1, 2$ $\quad \{x_1 \approx SK,\ x_2 \approx C\}$
$x_2 \leftarrow \mathsf{retrieve}(x_2^{hnd})$
$type_1 \leftarrow \mathsf{get\_type}(x_1^{hnd})$
if $(type_1 \neq \text{skse}) \vee (x_2 \neq v)$ **then**
    Abort
end if
$y^{hnd} \leftarrow \mathsf{sym\_decrypt}(x_1^{hnd}, m_{5.2}^{hnd})$ $\quad \{y \approx C, t'_C\}$
$y_i^{hnd} \leftarrow \mathsf{list\_proj}(y^{hnd}, i)$ for $i = 1, 2$ $\quad \{y_1 \approx C,\ y_2 \approx t'_C\}$
$y_1 \leftarrow \mathsf{retrieve}(y_1^{hnd})$
if $(y_1 \neq v)$ **then**
    Abort
end if
$m_6^{hnd} \leftarrow \mathsf{sym\_encrypt}(x_1^{hnd}, y_2^{hnd})$ $\quad \{m_6 \approx \{t'_C\}_{SK}\}$
$\mathsf{send\_i}(S, m_6^{hnd})$
output $(\mathsf{ok}, \mathsf{PK}, v, x_1^{hnd})$ at $\text{KA\_out}_S!$

Fig. 12 Algorithm 5: Behavior of server

A) Input:$(\mathsf{new\_prot}, \mathsf{K5}, K, T)$ at $KA\_\text{in}_u?$.

$n_{u,1}^{hnd} \leftarrow \mathsf{gen\_nonce}()$
$u^{hnd} \leftarrow \mathsf{store}(u)$
$T^{hnd} \leftarrow \mathsf{store}(T)$
$m_1^{hnd} \leftarrow \mathsf{list}(u^{hnd}, T^{hnd}, n_{u,1}^{hnd})$ $\quad \{m_1 \approx C, T, n_1\}$
$Nonce_u := Nonce_u \cup \{(n_{u,1}^{hnd}, K)\}$
$\mathsf{send\_i}(K, m_1^{hnd})$

B) Input:$(\mathsf{continue\_prot}, \text{K5}, T, S, AK^{hnd})$ at $\text{KS\_in}_u?$ for $S \in \{S_1, ..., S_l\}$

if $(\nexists(TGT^{hnd}, AK^{hnd}, T) \in TGTicket_u)$ **then**
    Abort
end if
$z^{hnd} \leftarrow \mathsf{list}(u^{hnd}, t_u^{hnd})$ $\quad \{z \approx C, t_C\}$
$auth^{hnd} \leftarrow \mathsf{sym\_encrypt}(AK^{hnd}, z^{hnd})$ $\quad \{auth \approx \{C, t_C\}_{AK}\}$

$n_{u,3}^{hnd} \leftarrow \mathsf{gen\_nonce}()$
$Nonce2_u := Nonce2_u \cup \{n_{u,3}^{hnd}, T, S)\}$
$m_3^{hnd} \leftarrow \mathsf{list}(TGT^{hnd}, auth^{hnd}, u^{hnd}, S^{hnd}, n_{u,3}^{hnd})$
$\quad \{m_3 \approx TGT, \{C, t_C\}_{AK}, C, S, n_3\}$
$\mathsf{send\_i}(T, m_3^{hnd})$

Fig. 13 Algorithm 1 of Kerberos 5: Evaluation of inputs from the user (starting the AS and TG exchange).

Input:(v, K, i, m^{hnd}) at $\text{out}_K?$ with $v \in \{1, ..., n\}$.

$x_i^{hnd} \leftarrow \mathsf{list\_proj}(m^{hnd}, i)$ for $i = 1, 2, 3$
$\quad \{x_1 \approx C, x_2 \approx T, x_3 \approx n_1\}$
$type_1 \leftarrow \mathsf{get\_type}(x_3^{hnd})$
$x_i \leftarrow \mathsf{retrieve}(x_i^{hnd})$ for $i = 1, 2$
if $(type_1 \neq \text{nonce}) \vee ((x_3, .) \in Nonce3_K) \vee (x_1 \neq v) \vee (x_2 \neq T)$ **then**
    Abort
end if
$v^{hnd} \leftarrow \mathsf{store}(v)$
$Nonce3_K := Nonce3_K \cup \{(x_3^{hnd}, v)\}$
$AK^{hnd} \leftarrow \mathsf{gen\_symenc\_key}()$
$z_1^{hnd} \leftarrow \mathsf{list}(AK^{hnd}, v^{hnd}, t_K^{hnd})$ $\quad \{z_1 \approx AK, C, t_K\}$
$TGT^{hnd} \leftarrow \mathsf{sym\_encrypt}(skse_{K,x_2}^{hnd}, z_1^{hnd})$
$\quad \{TGT \approx \{AK, C, t_K\}_{k_T}\}$
$z_2^{hnd} \leftarrow \mathsf{list}(AK^{hnd}, x_3^{hnd}, t_K^{hnd}, x_2^{hnd})$ $\quad \{z_2 \approx AK, n_1, t_K, T\}$
$m_{23} \leftarrow \mathsf{sym\_encrypt}(k_v^{hnd}, z_2^{hnd})$
$\quad \{m_{23} \approx \{AK, n_1, t_K, T\}_{k_C}\}$
$m_2^{hnd} \leftarrow \mathsf{list}(v^{hnd}, TGT^{hnd}, m_{23}^{hnd})$
$\quad \{m_2 \approx C, TGT, \{AK, n_1, t_K, T\}_{k_C}\}$
$\mathsf{send\_i}(v, m_2^{hnd})$

Fig. 14 Algorithm 2 of Kerberos 5: Behavior of the KAS

B.3 Auxiliary Properties for Public-key Kerberos

Next we will consider the auxiliary properties from Section 4.1 for Public-key Kerberos. We will again informally state the property, formalize it as a lemma in the language of the BPW model, and prove it:

Handles contained in the sets $Nonce_u$ and $Nonce2_u$ are indeed handles of u to nonces.

Lemma 2 (Correct Nonce Owner) *For all $u \in \mathcal{H}$, if $(x^{hnd}, \ldots) \in Nonce_u$ or $(x^{hnd}, \ldots) \in Nonce2_u$, then $D[hnd_u = x^{hnd}] \neq \downarrow$ and $D[hnd_u = x^{hnd}].type = nonce$.*

Proof Let $(x^{hnd}, \ldots) \in Nonce_u$. By construction, this entry has been added to $Nonce_u$ by M_u^{PK} in step 1A.8. x^{hnd} has been generated through the input of the command $\mathsf{gen\_nonce}()$ at some time t at port $\text{in}_u?$ of $\text{TH}_{\mathcal{H}}$. Convention 1 implies $x^{hnd} \neq \downarrow$, as M_u^{PK} would abort otherwise and not add the entry to $Nonce_u$. By definition of $\mathsf{gen\_nonce}()$ and using Lemma 5.2 of [4] one gets that $D[hnd_u = x^{hnd}] \neq \downarrow$ and $D[hnd_u = x^{hnd}].type$ = nonce holds (the proof of the statement for $Nonce2_u$ is analogous). □

If K generated a symmetric key k or AK for honest v (i.e., on receiving a AS_REQ from v) and w has a handle to k or AK then w must either be v or K. And if T generated a symmetric key SK for honest v and server S and w has a handle to SK, then w must be either v, T or S.

Lemma 3 (Key Secrecy) *For all $v \in \mathcal{H}$, honest K, T, and $S \in \{S_1, \ldots, S_l\}$, and for all $j \leq size$ with $D[j].type = skse$:*

a) If $D[j]$ was created by M_K^{PK} in step 2.29 or step 2.30 then (with the notation of Algorithm 2 (Fig. 8))

$$D[j].hnd_w \neq \downarrow \quad \textit{implies } w \in \{v,\ K\}.$$

b) If $D[j]$ was created by M_K^{PK} in step 2.31 then (with the notation of Algorithm 2 (Fig. 8))

$$D[j].hnd_w \neq \downarrow \quad \textit{implies } w \in \{v,\ K,\ T\}.$$

c) If $D[j]$ was created by M_T^{PK} in step 4.18 then (with the notation of Algorithm 4 in Fig. 11)

$$D[j].hnd_w \neq \downarrow \quad \textit{implies } w \in \{v,\ T,\ S\}$$

where with the notation of Algorithm 4, $S = x_4$.

Proof a) Say $j \leq size$, $D[j].type$ = skse such that $D[j]$ was created by $\mathrm{M}_K^{\mathrm{PK}}$ in step 2.29 at time t (the case where $D[j]$ was generated in step 2.30 is analogous). The message m_2 (in the notation of Algorithm 2), to which a handle is sent out in step 2.42, contains $D[j]$ encrypted under v's public key. More precisely, a handle to $D[j]$ is part of the input to the command list creating z_1^{hnd} in step 2.33. The list z_1 is then signed in step 2.34 creating s_2, and the list z_2 is created in step 2.35 using a handle to s_2; finally z_2 is encrypted in step 2.36 under v's public key creating m_{21}^{hnd}, where $m_{21} = m_2.arg[1]$. Note that one can obtain a handle to $D[j]$ from m_{21}^{hnd} if one has v's private key (by applying the basic commands $\mathsf{decrypt}$, $\mathsf{list\_proj}$, $\mathsf{msg\_of\_sig}$); but the other components of m_2 do not contain any handle to $D[j]$, at most they only contain the index $D[j-1].ind$ (i.e., the index to the public identifier of the secret key $D[j]$, e.g. see m_{24}^{hnd} created in step 2.40). Since, by assumption, handles to private keys are not allowed to be sent around, only v can decrypt m_{21} and obtain

Input: (v, u, i, m^{hnd}) at out_u?

if $v = K$ **then** $\{$AS_REP is input$\}$
$c_i^{hnd} \leftarrow \mathsf{list\_proj}(m^{hnd}, i)$ for $i = 1, 2, 3$ $\{c_1 \approx C, c_2 \approx TGT, c_3 \approx \{AK, n_1, t_K, T\}_{k_C}\}$
$c_1 \leftarrow \mathsf{retrieve}(c_1^{hnd})$
if $(c_1 \neq u)$ **then**
  Abort
end if
$type_i \leftarrow \mathsf{get\_type}(c_i^{hnd})$ for $i = 2, 3$
if $(type_2 \neq \text{skse}) \vee (type_3 \neq \text{auth})$ **then**
  Abort
end if
$l_3^{hnd} \leftarrow \mathsf{sym\_decrypt}(skse_{uK}^{hnd}, c_3^{hnd})$ $\{l_3 \approx AK, n_1, t_K, T\}$
$y_i^{hnd} \leftarrow \mathsf{list\_proj}(l_3^{hnd}), i$ for $i = 1, 2, 4$ $\{y_1 \approx AK, y_2 \approx n_1, y_4 \approx T\}$
$type_4 \leftarrow \mathsf{get\_type}(y_1^{hnd})$
$type_5 \leftarrow \mathsf{get\_type}(y_2^{hnd})$
$y_4 \leftarrow \mathsf{retrieve}(y_4^{hnd})$
if $(type_3 \neq \text{skse}) \vee (type_4 \neq \text{nonce}) \vee (y_4 \neq T) \vee (\nexists!\,(\tilde{n}^{hnd}, K) \notin Nonce_u)$ **then**
  Abort
end if
$TGTicket_u := TGTicket_u \cup \{(c_2^{hnd}, y_1^{hnd}, T)\}$
output $(\mathsf{ok}, \mathsf{KAS\_exchange\ K5}, K, T, y_1^{hnd}, c_2^{hnd})$ at $\mathrm{KA\_out}_u$!
else if $v = T$ **then** $\{$TGS_REP is input$\}$
$d_i^{hnd} \leftarrow \mathsf{list\_proj}(m^{hnd}, i)$ for $i = 1, 2, 3$ $\{d_1 \approx C,\ d_2 \approx ST,\ d_3 \approx \{SK, n_3, t_T, S\}_{AK}\}$
$d_1 \leftarrow \mathsf{retrieve}(d_1^{hnd})$
if $(d_1 \neq u)$ $\vee$ $(\nexists!\,(., AK^{hnd}, T) \in TGTicket_u) :$ $\mathsf{sym\_decrypt}(AK^{hnd}, d_3^{hnd}) \neq \downarrow)$ **then**
  Abort
end if
$l_2^{hnd} \leftarrow \mathsf{sym\_decrypt}(AK^{hnd}, d_3^{hnd})$ $\{l_2 \approx SK, n_3, t_T, S\}$
$x_{2.i}^{hnd} \leftarrow \mathsf{list\_proj}(l_2^{hnd}, i)$ for $i = 1, 2, 4$ $\{x_{2.1} \approx SKey,\ x_{2.2} \approx n_3,\ x_{2.4} \approx S\}$
$type_5 \leftarrow \mathsf{get\_type}(x_{2.1}^{hnd})$
$type_6 \leftarrow \mathsf{get\_type}(x_{2.2}^{hnd})$
$S \leftarrow retrieve(x_{2.4}^{hnd})$
if $(type_5 \neq \text{skse}) \vee (type_6 \neq \text{nonce}) \vee ((x_{2.2}^{hnd}, T, S) \notin Nonce2_u)$ **then**
  Abort
end if
$x_5^{hnd} \leftarrow \mathsf{list}(u^{hnd}, t_u'^{hnd})$ $\{x_5 \approx C, t_C'\}$
$m_{5.2}^{hnd} \leftarrow \mathsf{sym\_encrypt}(x_{2.1}^{hnd}, x_5^{hnd})$ $\{m_{5.2} \approx \{C, t_u'\}_{SK}\}$
$m_5^{hnd} \leftarrow \mathsf{list}(d_2^{hnd}, m_{5.2}^{hnd})$ $\{m_5 \approx ST, \{C, t_u'\}_{SK}\}$
$Session\_KeysS_u := Session\_KeysS_u \cup \{(S, x_{2.1}^{hnd})\}$
$\mathsf{send\_i}(S, m_5^{hnd})$
else if $v = S \in \{S_1, ..., S_l\}$ **then** $\{$AP_REP is input$\}$
if $(\nexists!(S, SK^{hnd}) \in Session\_KeysS_u:$ $\mathsf{sym\_decrypt}(SK^{hnd}, m^{hnd}) \neq \downarrow)$ **then**
  Abort
end if
$l_3^{hnd} \leftarrow \mathsf{sym\_decrypt}(SK^{hnd}, m^{hnd})$ $\{m \approx \{t_C'\}_{SK}\}$
$x_{3.1}^{hnd} \leftarrow \mathsf{list\_proj}(l_3^{hnd}, 1)$ $\{x_{3.1} \approx t_C'\}$
$x_{3.1} \leftarrow retrieve(x_{3.1}^{hnd})$
if $x_{3.1} = u$ **then**
  Abort
end if
output $(\mathsf{ok}, \mathsf{K5}, S, SK^{hnd})$ at $\mathrm{KA\_out}_u$!

Fig. 15 Algorithm 3 of Kerberos 5: Behavior of user after initialization

a handle to $D[j]$ after m_2^{hnd} is sent in step 2.42. And since v is honest, $\mathrm{M}_v^{\mathrm{PK}}$ never sends any message from which a handle to $D[j]$ can be obtained for time $t' > t$ (by Algorithms 1 and 3). One immediately gets $D[j].hnd_a = \downarrow$ for all $t' > t$.

b) Let $j \leq size$, $D[j].type = \text{skse}$ such that $D[j]$ was created by $\mathrm{M}_K^{\mathrm{PK}}$ in step 2.31 at time t. The message m_2 (in the notation of Algorithm 2), to which a handle is sent out in step 2.42, contains $D[j]$ encrypted under the symmetric key k_e created by $\mathrm{M}_K^{\mathrm{PK}}$ in step 2.29 (more precisely, $D[m_{24}.ind = m_2.arg[4]]$ is created by applying the command $\mathsf{sym\_encrypt}$ taking as arguments a handle to k_e and a handle to the list z_4 where $z_4.arg[1] = D[j].ind$). By Key Secrecy a) and since v is honest, only v or K can decrypt m_{24}. The message m_2 further contains $D[j]$ encrypted under a symmetric key $\text{skse}_{K,T}$ shared exclusively between K and T (more precisely, $D[TGT.ind = m_2.arg[3]]$ is created in step 2.38 by applying the command $\mathsf{sym\_encrypt}$ taking as arguments a handle to $\text{skse}_{K,T}$ and a handle to the list z_3, where $z_3.arg[1] = D[j].ind$). By construction of Algorithm 4 (Fig. 11) and since T is honest, one sees that $\mathrm{M}_T^{\mathrm{PK}}$ never sends anything from which a handle to $D[j]$ can be obtained as part of a new list for time $t' > t$. Also, by construction of Algorithms 1 and 3 (Figs. 7, 9, and 10) and since v and K are honest, one sees that v and K do not send out any list from which a handle to $D[j]$ can be obtained for time $t' > t$.

c) Let $j \leq size$, $D[j].type = \text{skse}$ such that $D[j]$ was created by $\mathrm{M}_T^{\mathrm{PK}}$ in step 4.18 at time t. The list m_4 (in the notation of Algorithm 4), to which a handle is sent out in step 4.24, contains $D[j]$ in ST which is a symmetric encryption under a symmetric key skse_{TS} shared exclusively between T and S (i.e., $m_4.arg[2] = ST.ind$, $ST.arg[1] = D[j].ind$), and m_4 also contains $D[j]$ in a list $m_{4.3}$ (where $m_{4.3}.ind = m_4.arg[3]$), which is a symmetric encryption under a key $y_{1.1}$. T gets a handle to the key $y_{1.1}$ in step 4.3, i.e., after decryption with the symmetric key shared exclusively between T and K (i.e., skse_{KT}^{hnd}; see step 4.1), otherwise there would be an abort, by Convention 1. Since, by construction, $\mathrm{M}_T^{\mathrm{PK}}$ does not use the key skse_{KT} for encryption, $\mathrm{M}_K^{\mathrm{PK}}$ must have created the cyphertext containing a handle to the key $y_{1.1}$. From Algorithm 2 one can now infer that $\mathrm{M}_K^{\mathrm{PK}}$ must have created the key $y_{1.1}$. Key Secrecy b) and the honesty of v, K and T imply that only v, T, K have handles to this key. T and K do not use this second key for decryption, therefore only v can get a handle to $D[j]$ through decryption with the key $y_{1.1}$. Also, only $\mathrm{M}_S^{\mathrm{PK}}$ uses skse_{TS}^{hnd} for decryption (in step 5.2). But, by construction, neither $\mathrm{M}_S^{\mathrm{PK}}$ nor $\mathrm{M}_v^{\mathrm{PK}}$ send out any message, from which a handle to $D[j]$ can be obtained for time $t' > t$. $\square$

If honest user u receives what appears to u to be a valid AS_REP message then this message (disregarding the TGT) was indeed generated by K for u and an adversary cannot learn the contained symmetric keys.

Lemma 4 (Authentication of KAS to client and Secrecy of AK) *For all $u \in \mathcal{H}$, honest KAS K and TGS T, and for all $j \leq size$ with $D[j].type = list$ and $j^{hnd} := D[j].hnd_u \neq \downarrow$:*
If $l_i := D[j].arg[i]$ for $i = 1, 4$
with $D[l_1].type = enc$ and $D[l_4].type = symenc$,
$x_1 := D[l_1].arg[2]$ with $D[x_1].type = list$, $\{\approx cert_K, [k_e, k_a, ck]_{sk_K}\}$
$x_{1.1} := D[x_1].arg[1]$ with $D[x_{1.1}].type = sig$, $\{\approx cert_K\}$
$x_{1.2} := D[x_1].arg[2]$ with $D[x_{1.2}].type = sig$, $\{\approx [k_e, k_a, ck]_{sk_K}\}$
$z_1 := D[x_{1.1}].arg[2]$ with $D[z_1].type = list$, $\{\approx K, pk_K\}$
$y_{1.1} := D[z_1].arg[2]$ with $D[y_{1.1}].type = pke$, $\{\approx pk_K\}$
$y_{1.2} := D[x_{1.2}].arg[2]$ with $D[y_{1.2}].type = list$, $\{\approx k_e, k_a, ck\}$
$s_1 := D[y_{1.2}].arg[1]$ with $D[s_1].type = skse$, $\{\approx k_e\}$
$s_2 := D[y_{1.2}].arg[2]$ with $D[s_2].type = ska$, $\{\approx k_a\}$
$r_1 := D[y_{1.2}].arg[3]$ with $D[r_1].type = auth$, $\{\approx ck\}$
$q_1 := D[r_1].arg[1]$ with $D[q_1].type = list$, $\{\approx m_1\}$
$p_1 := D[r_1].arg[2]$ with $D[p_1].type = pka$, $\{\approx k_a\}$

$x_4 := D[l_4].arg[1]$ *with* $D[x_4].type = list$, $\{\approx AK, n_1, t_k, T\}$
$y_4 := D[x_4].arg[2]$ *with* $D[y_4].type = nonce$, $\{\approx n_1\}$
and if furthermore

a) *for* $pke := D[l_1].arg[1]$ *one has* $D[pke-1].hnd_u \neq \downarrow$
b) $y_{1.1} = D[x_{1.2}].arg[1]$ *and* $D[z_1.arg[1]] = K$
c) $p_1 = s_2 + 1$ *and* $D[l_4].arg[2] = s_1 + 1$
d) $(D[y_4].hnd_u,\ D[q_1].hnd_u,\ K) \in Nonce_u$

then $D[l_1]$ *was created by* M_K^{PK} *in step 2.32 and* $D[l_4]$ *was created by* M_K^{PK} *in step 2.36 and both their indices are arguments of a list created by* M_K^{PK} *in step 2.41 and sent to* u *in step 2.42. Furthermore,* $D[s_1].hnd_a = \downarrow$ *and therefore also* $D[x_4\ .arg[1]].hnd_a = \downarrow$.

Proof By hypothesis on the structure of $D[j]$, the entry $D[x_{1.2}]$ is a list signed using a private key corresponding to the public key $D[y_{1.1}]$, i.e., the index of the private key is $y_{1.1} - 1$. By hypothesis b) and since handles to private keys are never sent around by honest K, $\mathrm{TH}_{\mathcal{H}}$ must have generated $D[x_{1.2}]$ when receiving the command sign from $\mathrm{M}_K^{\mathrm{PK}}$ in step 2.34 using K's private key. This occurs only if there was an input (v, K, i, m^{hnd}) at $\mathrm{out}_K?$. By construction of Algorithm 2, K signs a list in step 2.34 consisting of the indices of a symmetric key generated in step 2.29, a message authentication code key generated in step 2.30, and a message authentication code created in step 2.32, over m using the symmetric key generated in step 2.30; i.e., s_1 is the index for this symmetric key, s_2 is the index for the MAC key, and q_1 is the index of m. In steps 2.34 and 2.35 $\mathrm{M}_K^{\mathrm{PK}}$ then creates a list of the signed message and its certificate and encrypts this list with v's public key. By the structure of $D[j]$ and by hypothesis c), one can see that $D[x_1]$ is such a list. Since by hypothesis d) $(D[y_4].hnd_u,\ D[q_1].hnd_u,\ K) \in Nonce_u$, Correct Nonce Owner implies that this element was stored there by $\mathrm{M}_u^{\mathrm{PK}}$ while running Algorithm 1A; in particular, by construction of Algorithm 1, $\mathrm{M}_u^{\mathrm{PK}}$ must, in step 1A.9, have sent a handle to the list of index q_1 to K, which contained the nonce indexed by y_4. Since u's name is contained in that message (by Algorithm 1A step 1A.7, $q_1.arg[3] = u$), this implies that v must equal u, as checked in step 2.4, otherwise Algorithm 2 would have aborted by Convention 1. This, on the other hand, implies that $\mathrm{M}_K^{\mathrm{PK}}$ used u's public key for encryption. Hypothesis a) now gives that $D[l_1]$ was generated by $\mathrm{M}_K^{\mathrm{PK}}$ in step 2.36. Moreover, we assumed that u is honest and therefore, Key Secrecy gives that only u or K can use the key from $D[s_1]$ for encryption. Inspection of Algorithm 1 and Algorithm 2 show that u does not use this key for encryption and that K uses this key for encryption of a list containing the nonce indexed y_4 in step 2.340, i.e., $D[l_4]$ was generated by $\mathrm{M}_K^{\mathrm{PK}}$ in step 2.41. Finally, in step 2.42, $\mathrm{M}_K^{\mathrm{PK}}$ sends a handle to the list m_2 to $v = u$, where $m_2.arg[1] = D[l_1].ind$ and $m_2.arg[4] = D[l_4]$. Furthermore, Key Secrecy implies that $D[s_1].hnd_a = \downarrow$. □

If TGS T receives a TGT and an authenticator $\{u, t_u\}_{AK}$ where the key AK and the username of an honest user u are contained in the TGT, then the TGT was generated by K and the authenticator was created by u.

Lemma 5 (TGS Authentication of the TGT) *For all* $u \in \mathcal{H}$*, honest KAS* K *and TGS* T *and for all* $j \leq size$ *with* $D[j].type$ $= list$ *and* $j^{hnd} := D[j].hnd_T \neq \downarrow$:
$l_1 := D[j].arg[1]$ *with* $D[l_1].type = symenc$, $\{\approx TGT\}$
$l_2 := D[j].arg[2]$ *with* $D[l_2].type = symenc$, $\{\approx \{u, t_u\}_{AK}\}$
$x_1 := D[l_1].arg[1]$ *with* $D[x_1].type = pkse$, $\{\approx k_T\}$
$x_2 := D[l_1].arg[2]$ *with* $D[x_2].type = list$, $\{\approx AK, u, t_K\}$
$x_{2.1} := D[x_2].arg[1]$ *with* $D[x_{1.1}].type = skse$, $\{\approx AK\}$
$y_1 := D[l_2].arg[1]$ *with* $D[x_1].type = pkse$, $\{\approx AK\}$
$y_2 := D[l_2].arg[2]$ *with* $D[y_2].type = list$, $\{\approx u, t_u\}$
and if furthermore

a) $D[x_1 + 1] = skse_{KT}$
b) $D[x_{2.1} - 1] = D[y_1]$
c) $D[x_2].arg[2] = D[y_2].arg[1] = u$

then entry $D[l_1]$ *was generated by* M_K^{PK} *in step 2.38 at a time* t *and entry* $D[l_2]$ *was generated by* M_u^{PK} *in step 1B.5 at a time* $t' > t$.

Proof By hypothesis a), $D[x_2]$ is encrypted under the symmetric key $skse_{KT}$ shared between K and T. It is assumed that only $\mathrm{M}_K^{\mathrm{PK}}$ and $\mathrm{M}_T^{\mathrm{PK}}$ have handles to the key $skse_{KT}$. Since by construction of Algorithm 4 (Fig. 11), $\mathrm{M}_T^{\mathrm{PK}}$ does not use $skse_{KT}$ for encryption, $\mathrm{M}_K^{\mathrm{PK}}$ must have created $D[l_1]$ in step 2.38. This step is only executed if there was an input (v, K, i, m^{hnd}) at $\mathrm{out}_K?$. In step 2.38 $\mathrm{M}_K^{\mathrm{PK}}$ encrypts a list z_3 (in the notation of Algorithm 2) created in step 2.37 using a handle to the name of user v and a handle to a symmetric key AK that was freshly generated by $\mathrm{TH}_{\mathcal{H}}$ earlier, when receiving the command gen_sym_key from $\mathrm{M}_K^{\mathrm{PK}}$ in step 2.31 (more precisely, $z_3.arg[2] = v$, $z_3.arg[1] = AK.ind$). Hypothesis c) now implies that $u = v$. Since u is assumed to be honest, we can use Key Secrecy to infer that only u, K or T can have handles to the key AK. Hypotheses b) and c) together imply that this key was used for encryption of a list containing u's name. By construction, only $\mathrm{M}_u^{\mathrm{PK}}$ uses this key for encryption of a list containing u's name, that is to say in step 1B.5, i.e., $\mathrm{M}_u^{\mathrm{PK}}$ generated $D[l_2]$ in step 1B.5. It is obvious that this encryption happened after $\mathrm{M}_K^{\mathrm{PK}}$ created $D[l_1]$, since $\mathrm{M}_K^{\mathrm{PK}}$ generates the symmetric encryption key AK and creates $D[l_1]$ before sending out a handle to a list from which another user can obtain a handle to this key. □

If honest user u receives what appears to u to be a valid TGS_REP, then the part of that message that is verifiable by u, encrypted under the symmetric key AK, was generated by T for u and S. And an adversary cannot learn the contained session key SK.

Lemma 6 (Authentication of TGS to client and Secrecy of SK**)** *For all* $u \in \mathcal{H}$*, honest KAS* K *and TGS* T *and for all* $j \leq size$ *with* $D[j].type = symenc$ *and* $j^{hnd} := D[j].hnd_u \neq \downarrow$:
$p_1 := D[j].arg[1]$ *with* $D[p_1].type = pkse$, $\{\approx AK\}$
$p_2 := D[j].arg[2]$ *with* $D[p_2].type = list$, $\{\approx SK, n_3, t_T, S\}$
$p_{2.1} := D[p_2].arg[1]$ *with* $D[p_{2.1}].type = skse$, $\{\approx SK\}$
$p_{2.2} := D[p_2].arg[2]$ *with* $D[p_{2.2}].type = nonce$, $\{\approx n_3\}$
and if furthermore

a) $(., s_1^{hnd}, T) \in TGTicket_u$ *for* $s_1 := p_1 + 1$
b) $(p_{2.2}^{hnd}, T, D[p_2].arg[4]) \in Nonce2_u$

then $D[j]$ *was created by* M_T^{PK} *in step 4.22.*
Furthermore, $D[p_{2.1}].hnd_a = \downarrow$.

Proof Hypothesis a) guarantees that $\mathrm{M}_u^{\mathrm{PK}}$ has a handle to the symmetric key s_1 needed to decrypt the message in $D[j]$. Elements in $TGTicket_u$ are stored there by $\mathrm{M}_u^{\mathrm{PK}}$ in step 3.42, which only occurs if there was an input (v', u, i, m'^{hnd}) at $\mathrm{out}_v?$. By construction of Algorithm 3, a handle to s_1 was received by $\mathrm{M}_u^{\mathrm{PK}}$ in step 3.31, otherwise the algorithm would have aborted. Steps 3.30, 3.25, 3.20 and 3.7 show that a handle to s_1 was obtained from the list m' satisfying the hypotheses of Lemma 4 for honest user u (*e.g.*, $D[m'.arg[1]]$ has the structure of $D[l_1]$ from Lemma 4 and $D[m'.arg[4]]$ has the structure of $D[l_4]$). Therefore an adversary cannot obtain a handle to s_1. But $\mathrm{M}_u^{\mathrm{PK}}$ does not create a list of the form of $D[j]$, neither does $\mathrm{M}_K^{\mathrm{PK}}$. So $\mathrm{M}_T^{\mathrm{PK}}$ must have created $D[j]$ in step 4.22. Hypothesis b) confirms that $D[j]$ has indeed the structure of the database entry created in step 4.20 by $\mathrm{M}_T^{\mathrm{PK}}$, i.e., $p_{2.2}^{hnd}$ is a nonce generated by $\mathrm{M}_u^{\mathrm{PK}}$

and $D[p_2].arg[4] \in \{S_1, ..., S_l\}$. In order for $\mathrm{M}_T^{\mathrm{PK}}$ to run Algorithm 4, there must have been an input (v, T, i, m^{hnd}) at $\mathrm{out}_T?$. Steps 4.12–4.16 ensure that the element x_2 (in the notation of Algorithm 4) has the same structure as the element in $D[l_2]$ in Lemma 5. Since the encryption is under the key s_1, v must be honest and therefore equal to u. Key Secrecy now implies that only T and u can have handles to the key generated by $\mathrm{M}_T^{\mathrm{PK}}$ in step 4.18, i.e., $D[p_{2.1}].hnd_a = \downarrow$. □

If server S receives a ST and an authenticator $\{u, t_v\}_{SK}$ where the key SK and the name u are contained in the ST, then the ST was generated by T and the authenticator was created by u.

Lemma 7 (Server Authentication of the ST) *For all $u \in \mathcal{H}$, honest $S \in \{S_1, ..., S_l\}$, KAS K and TGS T and for all $j \leq size$ with $D[j].type =$list and $j^{hnd} := D[j].hnd_S \neq \downarrow$:*

$l_1 := D[j].arg[1]$ *with* $D[l_1].type =$*symenc,* $\quad \{\approx ST\}$
$l_2 := D[j].arg[2]$ *with* $D[l_2].type =$*symenc,* $\quad \{\approx \{u, t'_u\}_{SK}\}$
$p_1 := D[l_1].arg[1]$ *with* $D[p_1].type =$*pkse,* $\quad \{\approx k_S\}$
$p_2 := D[l_1].arg[1]$ *with* $D[p_2].type =$*list,* $\quad \{\approx SK, u, t_T\}$
$p_{2.1} := D[p_2].arg[1]$ *with* $D[p_{2.1}].type =$*skse,* $\quad \{\approx SK\}$
$r_1 := D[l_2].arg[1]$ *with* $D[r_1].type =$*pkse,* $\quad \{\approx SK\}$
$r_2 := D[l_2].arg[2]$ *with* $D[r_2].type =$*list,* $\quad \{\approx u, t'_u\}$
and if furthermore

a) $D[p_1 + 1] = skse_{TS}$
b) $D[p_{2.1}] = D[r_1 - 1]$
c) $D[p_2].arg[2] = D[r_2].arg[1] = u$

then $D[l_1]$ was created by M_T^{PK} in step 4.20 at time t and $D[l_2]$ was created by M_u^{PK} in step 3.59 at time $t' > t$.

Proof By assumption, only T and S have handles to the long-term shared key $skse_{TS}$, which was used here for encryption, as stated by hypothesis *a*). But since by construction of Algorithm 5 (Fig. 12), $\mathrm{M}_S^{\mathrm{PK}}$ does not use the key $skse_{TS}$ for encryption, $\mathrm{M}_T^{\mathrm{PK}}$ must have used it in step 4.20. This step only occurs after there was an input (v, T, i, m^{hnd}) at $\mathrm{out}_T?$. In step 4.20 $\mathrm{M}_T^{\mathrm{PK}}$ encrypts a list which includes indices of a symmetric key generated in step 4.18 and also of v's name. Using hypothesis *c*) one obtains that $v = u$ and that the generated key is $D[p_{2.1}]$. Hence, $D[l_1]$ was created by $\mathrm{M}_T^{\mathrm{PK}}$ in step 4.20. We assumed u and S to be honest, and therefore Key Secrecy implies that only v, T or S can have a handle to the symmetric key in $D[p_{2.1}]$. From hypotheses *b*) and *c*) one can infer that the symmetric key in $D[p_{2.1}]$ was used for encryption of a list containing u's name in order to create $D[l_2]$. Since neither T nor S use that symmetric key to encrypt such a list, $\mathrm{M}_u^{\mathrm{PK}}$ must have generated $D[l_2]$ in step 3.59. Obviously, this happened after $D[l_1]$ was generated by $\mathrm{M}_T^{\mathrm{PK}}$ in step 4.20 at time t, since a handle to $D[p_{2.1}]$ was encrypted in step 4.20 before $\mathrm{M}_T^{\mathrm{PK}}$ sends out any list from which a handle to $D[p_{2.1}]$ can be obtained. □

B.4 Proof of Theorem 1, Public-key Kerberos part

Now we present the proof of Theorem 1 regarding public-key Kerberos:

Proof (of Thm. 1) First we prove the Secrecy Property: Say there was an output $(\mathrm{ok}, \mathrm{PK}, S, SK^{hnd})$ at $\mathrm{KA\_out}_u!$. Examining Algorithm 3 (Fig. 9, 10) we see that the handle SK^{hnd} and the server name S form an element (S, SK^{hnd}) of the set $Session\_KeysS_u$ (see steps 3.64, 3.65). By the definition of $Session\_KeysS_u$ (see step 3.61), $\mathrm{M}_u^{\mathrm{PK}}$ obtained the handle SK^{hnd} in step 3.51 and steps 3.55 & 3.56 guarantee that SK^{hnd} is indeed a handle to symmetric keys. By Algorithm 3 (steps 3.50–3.56), SK^{hnd}, the name of server S and a handle to a nonce $x_{2.2}$ were obtained from a list l_2 (in the notation of Algorithm 3) to which $\mathrm{M}_u^{\mathrm{PK}}$ obtained a handle in step 3.50 after decrypting d_3 with a symmetric key AK; i.e., $l_2.arg[1] = SK.ind$, $l_2.arg[2] = x_{2.1}.ind$, $l_2.arg[4] = S$, $d_3.type = symenc$, $D[d_3.arg[2]].type = pkse$ and $d_3.arg[2] = AK.ind - 1$, by the definition of the command sym_decrypt. Here $l_2.hnd_u, d_3.hnd_u, AK.hnd_u \neq \downarrow$, otherwise the algorithm would abort by Convention 1; i.e., $\mathrm{M}_u^{\mathrm{PK}}$ has handles to d_3 and to the key AK. Steps 3.47 & 3.48 imply that $(., AK^{hnd}, T)$ is an element of the set $TGTicket_u$. Furthermore, $(x_{2.2}^{hnd}, T, S)$ is an element in $Nonce2_u$, otherwise there would be an abort in step 3.56. Hence, $D[d_3]$ (in the notation of Algorithm 3) satisfies the hypotheses of Lemma 6 for the element $D[j]$. In particular, this means that an adversary cannot get a handle to the key SK.

Now say there was an output $(\mathrm{ok}, \mathrm{PK}, u, SK^{hnd})$ at $\mathrm{KA\_out}_S!$. This only occurs if there was an input (u, S, i, m^{hnd}) at $\mathrm{out}_S?$ at a past time for some list m. By Algorithm 5, the handle to SK was contained in a list x (in the notation of Algorithm 5), to which $\mathrm{M}_S^{\mathrm{PK}}$ obtained a handle in step 5.2 after decryption of $m_{5.1} = m.arg[1]$ using the long-term shared key $skse_{TS}$. Here $m_{5.1}.hnd_S, x.hnd_S \neq \downarrow$ since otherwise the algorithm would abort by Convention 1. Steps 5.6 and 5.7 ensure that the index of $x_1 = SK$ really points to a symmetric key. Also, all other steps of Algorithm 5 must have been executed by $\mathrm{M}_S^{\mathrm{PK}}$ without abort before the output $(\mathrm{ok}, \mathrm{PK}, u, k^{hnd})$. Therefore we see that steps 5.2–5.7 and the definitions of the basic command sym_decrypt guarantee that $m_{5.1}$ from Algorithm 5 must have the same structure as l_1 from Lemma 7. Furthermore, steps 5.9–5.13 show that u's name was included in a list y to which S gets a handle in step 5.9 after decryption of $m_{5.2}$ using the key $x_1 = SK$. Therefore, $m_{5.2}$ from Algorithm 5 has the same structure as l_2 from Lemma 7. Since it is easy to verify that hypotheses *a*), *b*) and *c*) are also satisfied by the corresponding indices contained in $m_{5.1}$ and $m_{5.2}$, and since u is honest, we can use Lemma 7 to infer that an adversary cannot get a handle to the key SK. This proves the Secrecy Property.

Next we prove the Authentication Property: i) Say there was an output (ok, PK, v, x_1^{hnd}) at $\mathrm{KA\_out}_S!$ at a time $t_3 \in \mathbb{N}$. By construction of Algorithm 5, there must have been an input (v, S, i, m^{hnd}) at $\mathrm{out}_S?$ at a past time. In order for there not to be any abort during the execution of Algorithm 5 at some point between the input (v, S, i, m^{hnd}) at $\mathrm{out}_S?$ and the output (ok, PK, v, x_1^{hnd}) at $\mathrm{KA\_out}_S!$, we see, just as above, that m's components $m_{5.1} = m.arg[1]$ and $m_{5.2} = m.arg[2]$ must satisfy the hypotheses for Lemma 7. And since v is honest, Lemma 7 implies that $m_{5.2}$, which consists of a list that contains v's name and that is encrypted under the symmetric key SK, must have been created by $\mathrm{M}_v^{\mathrm{PK}}$ in step 3.59. By construction of Algorithm 3, there must have been an input $(T, v, i, \tilde{m}^{hnd})$ at $\mathrm{out}_v?$ at a past time and an output (ok, PK, S', SK^{hnd}) at $\mathrm{KA\_out}_v!$ at some later time. Furthermore, (S', SK) is an element of the set $Session\_KeysS_u$. By the definition of this set in step 3.61, $\mathrm{M}_v^{\mathrm{PK}}$ received a handle to SK in step 3.51 after decryption of $d_3 = D[\tilde{m}].arg[3]$ using a symmetric key AK (in the notation of Algorithm 3, i.e., $d_3.arg[2] + 1 = AK.ind$) in step 3.50. In fact, steps 3.47–3.56 and the definitions of the basic commands sym_decrypt and list_proj ensure that $D[d_3]$ has the same structure as $D[j]$ from Lemma 6 and satisfies the hypotheses of Lemma 6. Therefore, by Lemma 6, $D[d_3]$ was created by $\mathrm{M}_T^{\mathrm{PK}}$ in step 4.22, i.e., the key SK must have been generated by $\mathrm{M}_T^{\mathrm{PK}}$ step 4.18 for v and S'. Key Secrecy implies that an adversary cannot get a handle to SK. One immediately gets that $S' = S$. Algorithm 4 is only executed by $\mathrm{M}_T^{\mathrm{PK}}$ if there was an input $(v, T, i, \hat{m}^{hnd})$ at $\mathrm{out}_T?$, and handles to an in step 4.18 created element can only be obtained by other users if there was no abort during that run of Algorithm 4. This implies that $D[\hat{m}.arg[1]]$ must have the

same structure as $D[l_1]$ from Lemma 5, as ensured by steps 4.2–4.9, and $D[\hat{m}.arg[2]]$ must have the same structure as $D[l_1]$ from Lemma 5, as ensured by steps 4.11–4.16 and the definitions of the basic commands sym_decrypt and list_proj. It is obvious that all hypotheses of Lemma 5 are satisfied and so one gets, by Lemma 5, that $D[\hat{m}.arg[1]]$ was created by $\mathrm{M}_K^{\mathrm{PK}}$ in step 2.38 and $D[\hat{m}.arg[2]]$ was created by $\mathrm{M}_v^{\mathrm{PK}}$ in step 1B.5. The latter implies that there was an input (continue_prot PK, T, S, AK^{hnd}) at KA_in$_v$! at a past time $t_2 < t_3$. On the other hand, by the definition of $TGTicket_u$ in step 3.42 and by steps 1B.1 & 1B.2, there must have been an output in step 3.43 that contained a handle to the same symmetric key as in the input to Algorithm 1B, namely the key AK. Otherwise there would be an abort in step 1B.2, i.e., there was an output (ok, KAS_exchange PK, $K, T, (.,.,., AK^{hnd})$) at KA_out$_v$!. Since the execution of Algorithm 3 did not produce an error, one can use Lemma 4 to infer that $\mathrm{M}_K^{\mathrm{PK}}$ must have run Algorithm 2 and generated AK for v. Finally, by construction of Algorithm 2 and by definition of the command verify, one gets that v must have run Algorithm 1 with the input (new_prot, PK, K, T)) at KA_in$_v$! at a time $t_1 < t_2$.
ii) Now say that there was on output (ok, PK, S, SK^{hnd}) at port KA_out$_u$! at time t_2. By construction of Algorithm 3, this only happens after u received an input (S, u, i, m^{hnd}) at out$_u$? and without there being any abort during the execution of Algorithm 3 between the input and the output (steps 3.63–3.73). By steps 3.64 and 3.65, m was encrypted using a symmetric key SK (i.e., $m.type = symenc$ and $m.arg[2] = SK.ind - 1$) for which $(S, SK^{hnd}) \in Session\_KeysS_u$. Steps 3.68–3.71 ensure that u's name was not the first argument of the list l_3 (in the notation of Algorithm 3) to which $\mathrm{M}_u^{\mathrm{PK}}$ obtains a handle after decryption of m using SK (i.e., $u \neq l_3.arg[1]$). Here $l_3.hnd_u \neq \downarrow$ otherwise there would be an abort of Algorithm 3, by Convention 1. On the other hand, the element $(S, SK^{hnd}) \in Session\_KeysS_u$ was added in step 3.61 after the key SK was used for encryption in step 3.59 of a list x_5 (in the notation of Algorithm 3) which does contain u's name as its first argument (i.e., $u = x_5.arg[1]$). By construction of Algorithm 3, step 3.59 is the only time that a symmetric key SK, for which $(S, SK^{hnd}) \in Session\_KeysS_u$, is used for encryption by $\mathrm{M}_u^{\mathrm{PK}}$. Therefore, since the list l_3 in m does not contain u's name as its first argument, $\mathrm{M}_u^{\mathrm{PK}}$ did not create $D[m]$. In order for (S, SK^{hnd}) to be added to $Session\_KeysS_u$ in step 3.61, there must have been an input $(T, u, i, \tilde{m}^{hnd})$ at out$_u$? in the past and there could not have be any abort in the steps 3.44–3.61. But these steps, together with the definitions of the basic commands sym_decrypt and list_proj, guarantee that the $\tilde{m}$ component d_3 (in the notation of Algorithm 3) satisfies the hypotheses of Lemma 6 (where $SK = D[d_3.arg[1]].arg[1]$ and $S = D[d_3.arg[1]].arg[4]$). Using Lemma 6 we see that only T, u and S can have handles to SK. Hence, $\mathrm{M}_S^{\mathrm{PK}}$ must have used the handle to the key SK for encryption at a time $t_1 < t_2$. This can only happen in step 5.15 after receiving an input $(v', S, i, \tilde{m}^{hnd})$ at out$_S$?, where $v' = u$ as guaranteed by step 5.9–5.13. The encryption that $\mathrm{M}_S^{\mathrm{PK}}$ generated using the key SK must have been sent for others to obtain a handle for it, so there was no abort in step 5.16, and therefore there must have been an output (ok, PK, u, SK^{hnd}) at KA_out$_S$! at some time $t_1 < t_2$. □

B.5 Auxiliary Properties for Basic Kerberos

In the following we will consider the auxiliary properties for Basic Kerberos, i.e., in particular, 'Algorithm 1', 'Algorithm 2' and 'Algorithm 3' will here refer to the algorithm in Figures 13, 14, and 15. The remaining algorithms for the TGS T and for a server S are valid for both Public-key and Basic Kerberos (i.e., for M_T^{K5}, M_S^{K5} just like for M_T^{PK}, M_S^{PK}).

Handles contained in the sets $Nonce_u$ and $Nonce2_u$ are indeed handles of u to nonces.

Lemma 8 (Correct Nonce Owner) *For all $u \in \mathcal{H}$, and all $(x^{hnd}, \ldots) \in Nonce_u$ or $(x^{hnd}, \ldots) \in Nonce2_u$, it holds that $D[hnd_u = x^{hnd}] \neq \downarrow$ and $D[hnd_u = x^{hnd}].type = nonce$.*

Proof Let $(x^{hnd}, \ldots) \in Nonce_u$. By construction, this entry has been added to $Nonce_u$ by M_u^{K5} in step 1A.5. x^{hnd} has been generated through the input of the command gen_nonce() at some time t at port in$_u$? of $\mathrm{TH}_{\mathcal{H}}$. Convention 1 implies $x^{hnd} \neq \downarrow$, as M_u^{K5} would abort otherwise and not add the entry to $Nonce_u$. By definition of gen_nonce() and using Lemma 5.2 of [4] one gets that $D[hnd_u = x^{hnd}] \neq \downarrow$ and $D[hnd_u = x^{hnd}].type$ = nonce holds (the proof of the statement for $Nonce2_u$ is analogous). □

If K generated a symmetric key k or AK for v (i.e., on receiving a AS_REQ from v) and w has a handle to k or AK, then w must either be v or K. And if T generated a symmetric key SK for v and server S and w has a handle to SK, then w must be either v, T, or S.

Lemma 9 (Key Secrecy) *For all $u, v \in \mathcal{H}$, honest K, T, and $S \in \{S_1, \ldots, S_l\}$, and for all $j \leq size$ with $D[j].type = skse$:*

a) If $D[j]$ was created by M_K^{K5} in step 2.9 then (with the notation of Algorithm 2 (Fig. 14))

$$D[j].hnd_w \neq \downarrow \quad \textit{implies } w \in \{v,\ K,\ T\}.$$

b) If $D[j]$ was created by M_T^{K5} in step 4.18 then (with the notation of Algorithm 4 (Fig. 11))

$$D[j].hnd_w \neq \downarrow \quad \textit{implies } w \in \{v,\ T,\ S\}$$

where with the notation of Algorithm 4, $S = x_4$.

Proof a) Let $j \leq size$, $D[j].type$ = skse such that $D[j]$ was created by $\mathrm{M}_K^{\mathrm{K5}}$ in step 2.9 at time t. The message m_2 (in the notation of Algorithm 2), to which a handle is sent out in step 2.15, contains $D[j]$ encrypted under the encrypted under a symmetric key k_v shared exclusively between K and v (k_v^{hnd}, see step 2.13). More precisely, $D[m_{23}.ind = m_2.arg[3]]$ is created by applying the command sym_encrypt taking as arguments a handle to k and a handle to the list z_2 where $z_2.arg[1] = D[j].ind$. By the assumption on the long-term key k_v, only v or K can decrypt m_{23}. The message m_2 further contains $D[j]$ encrypted under a symmetric key skse_{KT} shared exclusively between K and T (more precisely, $D[TGT.ind = m_2.arg[2]]$ is created in step 2.11 by applying the command sym_encrypt taking as arguments a handle to $\mathrm{skse}_{K,T}$ and a handle to the list z_1 where $z_1.arg[1] = D[j].ind$). By construction of Algorithm 4 (Fig. 11) and since T is honest, one sees that $\mathrm{M}_T^{\mathrm{K5}}$ never sends any message from which a handle to $D[j]$ for time $t' > t$. Also, by construction of Algorithms 1 and 3 (Figs. 13 and 15)and since v and K are honest, one sees that v and K do not send out any list from which a handle to $D[j]$ can be obtained for time $t' > t$.

b) Let $j \leq size$, $D[j].type$ = skse such that $D[j]$ was created by $\mathrm{M}_T^{\mathrm{K5}}$ in step 4.18 at time t. The message m_4 (in the notation of Algorithm 4), to which a handle is sent out in step 4.24, contains $D[j]$ in ST which is a symmetric encryption under a symmetric key skse_{TS} shared exclusively between T and S (i.e., $m_4.arg[2] = ST.ind$, $ST.arg[1] = D[j].ind$) and m_4 also contains $D[j]$ in a list $m_{4.3}$ (where $m_{4.3}.ind = m_4.arg[3]$) which is a symmetric encryption under a key $y_{1.1}$. T gets a handle to the key $y_{1.1}$ in step 4.3, i.e., after decryption with the symmetric key shared exclusively between T and K (i.e., skse_{KT}^{hnd}; see step 4.1), otherwise there would be an abort, by Convention 1. Since, by construction, M_T^{K5} does not use the key $skse_{KT}$ for encryption, M_K^{K5} must have created the cyphertext containing a handle to the key $y_{1.1}$. From Algorithm 2 one can now infer that M_K^{K5} must have created the key $y_{1.1}$. Key Secrecy a) and the honesty of v, K and T imply that only v, T, K have handles to this key. T and K do not use this second key for decryption, therefore only v can get a handle to $D[j]$ through decryption with the key $y_{1.1}$. Also, only M_S^{K5} uses $skse_{TS}^{hnd}$ for decryption (in step 5.2). But, by construction, neither M_S^{K5} nor M_v^{K5} send out any message, from which a handle to $D[j]$ can be obtainedfor time $t' > t$. □

If honest user u receives what appears to u to be a valid AS_REP message then this message (disregarding the TGT) was indeed generated by K for u and an adversary cannot learn the contained symmetric keys.

Lemma 10 (Authentication of KAS to client and Secrecy of AK) *For all $u \in \mathcal{H}$, honest KAS K and TGS T, and for all $j \leq size$ with $D[j].type = list$ and $j^{hnd} := D[j].hnd_u \neq \downarrow$:*
If $l_3 := D[j].arg[3]$ with $D[l_3].type = symenc$, $\{\approx \{AK, n_1, t_K, T\}_{k_u}\}$
$x_1 := D[l_3].arg[1]$ *with* $D[x_1].type = list$, $\{\approx AK, n_1, t_K, T\}$
$y_2 := D[x_1].arg[2]$ *with* $D[y_2].type = nonce$, $\{\approx n_1\}$
and if furthermore

a) $D[D[l_3].arg[2]].ind + 1 = D[k_u].ind$, i.e., $D[l_3].arg[2]$ is the public identifier of the long-term key k_u shared between K and u

then $D[l_3]$ was created by M_K^{K5} in step 2.13 and its index is an argument in a list sent to u in step 2.15.
Furthermore, $D[x_1.arg[1]].hnd_a = \downarrow$.

Proof By hypothesis a), $D[l_3]$ is encrypted using the long-term key k_u shared between K and u. By assumption on this key and since u is honest, only M_K^{K5} and M_u^{K5} have handles to k_u. Since, by construction, M_u^{K5} does not use this key for encryption, M_K^{K5} must have used it for encryption. This occurs only in step 2.13 after there was an input (v, K, i, m^{hnd}) at $\text{out}_K?$. By Algorithm 2, one has $v = u$. Furthermore, the key contained in l_3 with index $x_1.arg[1]$ was created in step 2.9. Since u is honest, Key Secrecy implies that an adversary cannot obtain a handle to this key. For any user, including u, to be able to obtain a handle to that key, M_u^{K5} must send it first. This happens in step 2.15, where M_u^{K5} sends a list m_2 (in the notation of Algorithm 2) to $v = u$, where $m_2.arg[3] = D[l_3].ind$. Furthermore, Key Secrecy implies that $D[x_1.arg[1]].hnd_a = \downarrow$. □

If TGS T receives a TGT and an authenticator $\{u, t_u\}_{AK}$ where the key AK and the username of an honest user u are contained in the TGT, then the TGT was generated by K and the authenticator was created by u.

Lemma 11 (TGS Authentication of the TGT) *For all $u \in \mathcal{H}$, honest KAS K and TGS T and for all $j \leq size$ with $D[j].type = list$ and $j^{hnd} := D[j].hnd_T \neq \downarrow$:*
$l_1 := D[j].arg[1]$ *with* $D[l_1].type = symenc$, $\{\approx TGT\}$
$l_2 := D[j].arg[2]$ *with* $D[l_2].type = symenc$, $\{\approx \{u, t_u\}_{AK}\}$
$x_1 := D[l_1].arg[1]$ *with* $D[x_1].type = pkse$, $\{\approx k_{KT}\}$
$x_2 := D[l_1].arg[2]$ *with* $D[x_2].type = list$, $\{\approx AK, u, t_K\}$
$x_{2.1} := D[x_2].arg[1]$ *with* $D[x_{1.1}].type = skse$, $\{\approx AK\}$
$y_1 := D[l_2].arg[1]$ *with* $D[x_1].type = pkse$, $\{\approx AK\}$
$y_2 := D[l_2].arg[2]$ *with* $D[y_2].type = list$, $\{\approx u, t_u\}$
and if furthermore

a) $D[x_1 + 1] = skse_{KT}$
b) $D[x_{2.1} - 1] = D[y_1]$
c) $D[x_2].arg[2] = D[y_2].arg[1] = u$

then entry $D[l_1]$ was created by M_K^{K5} in step 2.11 at a time t and entry $D[l_2]$ was generated by M_u^{K5} in step 1B.5 at a time $t' > t$.

Proof By hypothesis a), $D[x_2]$ is encrypted under the long-term key shared between K and T. It is assumed that only M_K^{K5} and M_T^{K5} have handles to the long-term key $skse_{KT}$. Since by construction of Algorithm 4 (Fig. 11), M_T^{K5} does not use this key for encryption, M_K^{K5} must have created $D[l_1]$ in step 2.11. This step is only executed if there was an input (v, K, i, m^{hnd}) at $\text{out}_K?$. In step 2.11 M_K^{K5} encrypts a list z_1 (in the notation of Algorithm 2) created in step 2.10 using a handle to the name of user v and a handle to a symmetric key AK that was freshly generated by $\text{TH}_\mathcal{H}$ earlier, after receiving the command gen_sym_key from M_K^{K5} in step 2.9 (more precisely, $z_1.arg[2] = v$, $z_1.arg[1] = AK.ind$). Hypothesis c) now implies that $u = v$. Since u is assumed to be honest, we can use Key Secrecy to infer that only u, K or T can have handles to the key AK. Hypotheses b) and c) state that this key was used for encryption of a list containing u's name. By construction, only M_u^{K5} uses this key for encryption of a list containing u's name, that is to say in step 1B.5, i.e., M_u^{K5} generated $D[l_2]$ in step 1B.5. It is obvious that this encryption happened after M_K^{K5} created $D[l_1]$, since M_K^{K5} generates the symmetric encryption key AK and creates $D[l_1]$ before sending out a handle to a list from which another user can obtain a handle to this key. □

If honest user u receives what appears to u to be a valid TGS_REP then the for u verifiable part of that message, encrypted under the symmetric key AK, was generated by T for u and S. And an adversary cannot learn the contained session key SK.

Lemma 12 (Authentication of TGS to client and Secrecy of SK) *For all $u \in \mathcal{H}$, honest KAS K and TGS T and for all $j \leq size$ with $D[j].type = symenc$ and $j^{hnd} := D[j].hnd_u \neq \downarrow$:*
$p_1 := D[j].arg[1]$ *with* $D[p_1].type = pkse$, $\{\approx AK\}$
$p_2 := D[j].arg[2]$ *with* $D[p_2].type = list$, $\{\approx SK, n_3, t_T, S\}$
$p_{2.1} := D[p_2].arg[1]$ *with* $D[p_{2.1}].type = skse$, $\{\approx SK\}$
$p_{2.2} := D[p_2].arg[2]$ *with* $D[p_{2.2}].type = nonce$, $\{\approx n_3\}$
and if furthermore

a) $(., s_1^{hnd}, T) \in TGTicket_u$ for $s_1 := p_1 + 1$
b) $(p_{2.2}^{hnd}, T, D[p_2].arg[4]) \in Nonce2_u$

then $D[j]$ was created by M_T^{K5} in step 4.22.
Furthermore, $D[p_{2.1}].hnd_a = \downarrow$.

Proof Analogous to the proof of Lemma 6.

If server S receives a ST and an authenticator $\{u, t_v\}_{SK}$ where the key SK and the name of honest user u are contained in the ST, then the ST was generated by T and the authenticator was created by u.

Lemma 13 (Server Authentication of the ST) *For all $u \in \mathcal{H}$, honest $S \in \{S_1, ..., S_l\}$, KAS K and TGS T and for all $j \leq size$ with $D[j].type = list$ and $j^{hnd} := D[j].hnd_S \neq \downarrow$:*
$l_1 := D[j].arg[1]$ *with* $D[l_1].type = symenc$, $\{\approx ST\}$
$l_2 := D[j].arg[2]$ *with* $D[l_2].type = symenc$, $\{\approx \{u, t'_u\}_{SK}\}$
$p_1 := D[l_1].arg[1]$ *with* $D[p_1].type = pkse$, $\{\approx k_{TS}\}$
$p_2 := D[l_1].arg[1]$ *with* $D[p_2].type = list$, $\{\approx SK, u, t_T\}$
$p_{2.1} := D[p_2].arg[1]$ *with* $D[p_{2.1}].type = skse$, $\{\approx SK\}$
$r_1 := D[l_2].arg[1]$ *with* $D[r_1].type = pkse$, $\{\approx SK\}$
$r_2 := D[l_2].arg[2]$ *with* $D[r_2].type = list$, $\{\approx u, t'_u\}$
and if furthermore

a) $D[p_1 + 1] = skse_{TS}$

b) $D[p_{2.1}] = D[r_1 - 1]$
c) $D[p_2].arg[2] = D[r_2].arg[1] = u$

then $D[l_1]$ *was created by* M_T^{K5} *in step 4.20 at time* t *and* $D[l_2]$ *was created by* M_u^{K5} *in step 3.36 at time* $t' > t$.

Proof Analogous to the proof of Lemma 7.

B.6 Proof of Theorem 1, Basic Kerberos part

Now we present the proof of Thm. 1 regarding basic Kerberos:

Proof (of Thm. 1) First we prove the Secrecy Property: Say there was an output $(\mathrm{ok}, \mathrm{K5}, S, SK^{hnd})$ at $\mathrm{KA\_out}_u!$. Examining Algorithm 3 (Fig. 15) we see that the handle SK^{hnd} and the server name S form an element (S, SK^{hnd}) of the set $Session\_KeysS_u$ (see steps 3.41, 3.42). By the definition of $Session\_KeysS_u$ (see step 3.38), $\mathrm{M}_u^{\mathrm{K5}}$ obtained the handle SK^{hnd} in step 3.28, and steps 3.32 and 3.33 guarantee that SK^{hnd} is indeed a handle to symmetric keys. By Algorithm 3 (steps 3.27–3.33), SK^{hnd}, the name of server S and a handle to a nonce $x_{2.2}$ were obtained from a list l_2 (in the notation of Algorithm 3) to which $\mathrm{M}_u^{\mathrm{K5}}$ obtained a handle in step 3.27 after decrypting d_3 with a symmetric key AK; i.e., $l_2.arg[1] = SK.ind$, $l_2.arg[2] = x_{2.1}.ind$, $l_2.arg[4] = S$, $d_3.type = symenc$, $D[d_3.arg[2]].type = pkse$ and $d_3.arg[2] = AK.ind - 1$, by the definition of the command sym_decrypt. Here $l_2.hnd_u, d_3.hnd_u, AK.hnd_u \neq\downarrow$, otherwise the algorithm would abort by Convention 1; i.e., $\mathrm{M}_u^{\mathrm{K5}}$ has handles to d_3 and to the key AK. Steps 3.24 and 3.25 imply that $(., AK^{hnd}, T)$ is an element of the set $TGTicket_u$. Furthermore, $(x_{2.2}^{hnd}, T, S)$ is an element in $Nonce2_u$, otherwise there would be an abort in step 3.33. Hence, $D[d_3]$ (in the notation of Algorithm 3) satisfies the hypotheses of Lemma 12 for the element $D[j]$. In particular, this means that an adversary cannot get a handle to the key SK.
Now say there was an output $(\mathrm{ok}, \mathrm{K5}, u, SK^{hnd})$ at $\mathrm{KA\_out}_S!$. This only occurs if there was an input (u, S, i, m^{hnd}) at $\mathrm{out}_S?$ at a past time for some list m. By Algorithm 5, the handle to SK was contained in a list x (in the notation of Algorithm 5), to which $\mathrm{M}_S^{\mathrm{K5}}$ obtained a handle in step 5.2 after decryption of $m_{5.1} = m.arg[1]$ using the long-term shared key $skse_{TS}$. Here $m_{5.1}.hnd_S, x.hnd_S \neq\downarrow$ since otherwise the algorithm would abort by Convention 1. Steps 5.6 and 5.7 ensure that the index of $x_1 = SK$ really points to a symmetric key. Also, all other steps of Algorithm 5 must have been executed by $\mathrm{M}_S^{\mathrm{K5}}$ without abort before the output $(\mathrm{ok}, \mathrm{K5}, u, k^{hnd})$. Therefore we see that steps 5.2–5.7 and the definitions of the basic command sym_decrypt guarantee that $m_{5.1}$ from Algorithm 5 must have the same structure as l_1 from Lemma 13. Furthermore, steps 5.9–5.13 show that u's name was included in a list y to which S gets a handle in step 5.9 after decryption of $m_{5.2}$ using the key $x_1 = SK$. Therefore, $m_{5.2}$ from Algorithm 5 has the same structure as l_2 from Lemma 13. Since it is easy to verify that hypotheses *a*), *b*) and *c*) are also satisfied by the corresponding indices contained in $m_{5.1}$ and $m_{5.2}$, and since u is honest, we can use Lemma 13 to infer that an adversary cannot get a handle to the key SK. This proves the Secrecy Property.

Next we prove the Authentication Property: i) Say there was an output (ok, K5, v, x_1^{hnd}) at $\mathrm{KA\_out}_S!$ at a time $t_3 \in \mathbb{N}$. By construction of Algorithm 5, there must have been an input (v, S, i, m^{hnd}) at $\mathrm{out}_S?$ at a past time. In order for there not to be any abort during the execution of Algorithm 5 at some point between the input (v, S, i, m^{hnd}) at $\mathrm{out}_S?$ and the output (ok, K5, v, x_1^{hnd}) at $\mathrm{KA\_out}_S!$, we see, just as above, that m's components $m_{5.1} = m.arg[1]$ and $m_{5.2} = m.arg[2]$ must satisfy the hypotheses for Lemma 7. And since v is honest, Lemma 13 implies that $m_{5.2}$, which consists of a list that contains v's name and that is encrypted under the symmetric key SK, must have been created by $\mathrm{M}_v^{\mathrm{K5}}$ in step 3.36. By construction of Algorithm 3, there must have been an input $(T, v, i, \tilde{m}^{hnd})$ at $\mathrm{out}_v?$ at a past time and an output (ok, K5, S', SK^{hnd}) at $\mathrm{KA\_out}_v!$ at some later time. Furthermore, (S', SK) is an element of the set $Session\_KeysS_u$. By the definition of this set in step 3.38, $\mathrm{M}_v^{\mathrm{K5}}$ received a handle to SK in step 3.28 after decryption of $d_3 = D[\tilde{m}].arg[3]$ using a symmetric key AK (in the notation of Algorithm 3, i.e., $d_3.arg[2] + 1 = AK.ind$) in step 3.27. In fact, steps 3.24–3.33 and the definitions of the basic commands sym_decrypt and list_proj ensure that $D[d_3]$ has the same structure as $D[j]$ from Lemma 12 and satisfies the hypotheses of Lemma 12. Therefore, $D[d_3]$ was created by $\mathrm{M}_T^{\mathrm{K5}}$ in step 4.22, i.e., the key SK must have been generated by $\mathrm{M}_T^{\mathrm{K5}}$ step 4.18 for v and S'. Key Secrecy implies that an adversary cannot get a handle to SK. One immediately gets that $S' = S$. Algorithm 4 is only executed by $\mathrm{M}_T^{\mathrm{K5}}$ if there was an input $(v, T, i, \hat{m}^{hnd})$ at $\mathrm{out}_T?$, and handles to an in step 4.18 created element can only be obtained by other users if there was no abort during that run of Algorithm 4. This means that $D[\hat{m}.arg[1]]$ must have the same structure as $D[l_1]$ from Lemma 11, as ensured by steps 4.2–4.9, and $D[\hat{m}.arg[2]]$ must have the same structure as $D[l_1]$ from Lemma 11, as ensured by steps 4.11–4.16 and the definitions of the basic commands sym_decrypt and list_proj. It is obvious that all hypotheses of Lemma 11 are satisfied and so one gets that $D[\hat{m}.arg[1]]$ was created by $\mathrm{M}_K^{\mathrm{K5}}$ in step 2.11 and $D[\hat{m}.arg[2]]$ was created by $\mathrm{M}_v^{\mathrm{K5}}$ in step 1B.5. The latter implies that there was an input (continue_prot K5, T, S, AK^{hnd}) at $\mathrm{KA\_in}_v!$ at a past time $t_2 < t_3$. On the other hand, by the definition of $TGTicket_u$ in step 3.19 and by steps 1B.1 and 1B.2, there must have been an output in step 3.20 that contained a handle to the same symmetric key as in the input to Algorithm 1B, namely the key AK. Otherwise there would be an abort in step 1B.2, i.e., there was an output $(\mathrm{ok}, \mathrm{KAS\_exchange\ K5}, K, T, (., ., ., AK^{hnd}))$ at $\mathrm{KA\_out}_v!$. Since the execution of Algorithm 3 did not produce an error, one can use Lemma 10 to infer that $\mathrm{M}_K^{\mathrm{K5}}$ must have run Algorithm 2 and generated AK for v. Finally, by construction of Algorithm 2 and by definition of the command verify, one gets that v must have run Algorithm 1 with the input $(\mathrm{new\_prot}, \mathrm{K5}, K, T))$ at $\mathrm{KA\_in}_v!$ at a time $t_1 < t_2$.

ii) Now say that there was on output $(\mathrm{ok}, \mathrm{K5}, S, SK^{hnd})$ at port $\mathrm{KA\_out}_u!$ at time t_2. By construction of Algorithm 3, this only happens after u received an input (S, u, i, m^{hnd}) at $\mathrm{out}_u?$ and without there being any abort during the execution of Algorithm 3 between the input and the output (steps 3.40–3.50). By steps 3.41 and 3.42, m was encrypted using a symmetric key SK (i.e., $m.type = symenc$ and $m.arg[2] = SK.ind - 1$) for which $(S, SK^{hnd}) \in Session\_KeysS_u$. Steps 3.45–3.48 ensure that u's name was not the first argument of the list l_3 (in the notation of Algorithm 3) to which $\mathrm{M}_u^{\mathrm{K5}}$ obtains a handle after decryption of m using SK (i.e., $u \neq l_3.arg[1]$). Here $l_3.hnd_u \neq\downarrow$, otherwise there would be an abort of Algorithm 3 by Convention 1. On the other hand, the element $(S, SK^{hnd}) \in Session\_KeysS_u$ was added in step 3.56 after the key SK was used for encryption in step 3.36 of a list x_5 (in the notation of Algorithm 3) which does contain u's name as its first argument (i.e., $u = x_5.arg[1]$). By construction of Algorithm 3, step 3.36 is the only time that a symmetric key SK, for which $(S, SK^{hnd}) \in Session\_KeysS_u$, is used for encryption by $\mathrm{M}_u^{\mathrm{K5}}$. Therefore, since the list l_3 in m does not contain u's name as its first argument, $\mathrm{M}_u^{\mathrm{K5}}$ did not create $D[m]$. In order for (S, SK^{hnd}) to be added to $Session\_KeysS_u$ in step 3.38, there must have been an input $(T, u, i, \tilde{m}^{hnd})$ at $\mathrm{out}_u?$ in the past and there could not have be any abort in the steps 3.21–3.38. But these steps, together with the definitions of the basic commands sym_decrypt and list_proj, guarantee that the

$\tilde{m}$ components d_3 (in the notation of Algorithm 3) satisfies the hypotheses of Lemma 12 (where $SK = D[d_3.arg[1]].arg[1]$ and $S = D[d_3.arg[1]].arg[4]$). Therefore, only T, u and S can have handles to SK. Hence, $\mathrm{M}_S^{\mathrm{K5}}$ must have used the handle to the key SK for encryption at a time $t_1 < t_2$. This can only happen in step 5.15 after receiving an input $(v', S, i, \tilde{m}^{hnd})$ at $\mathrm{out}_S?$, where $v' = u$ as guaranteed by step 5.9–5.13. The encryption that $\mathrm{M}_S^{\mathrm{K5}}$ generated using the key SK must have been sent for others to obtain a handle for it, so there was no abort in step 5.16, and therefore there must have been an output $(\mathrm{ok}, \mathrm{K5}, u, SK^{hnd})$ at $\mathrm{KA\_out}_S!$ at some time $t_1 < t_2$. □