FORMAL ANALYSIS OF THE KERBEROS AUTHENTICATION PROTOCOL

Joe-Kai Tsay

A Dissertation

in

Mathematics

Presented to the Faculties of the University of Pennsylvania in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

2008

Andre Scedrov
Supervisor of Dissertation

Tony Pantev
Graduate Group Chairperson

Acknowledgments

First and foremost, I would like to thank my advisor Andre Scedrov for his tremendous support. It has been an exceptional experience for me to work as a graduate student under his supervision. I would also like to thank my other co-authors Iliano Cervesato, Aaron D. Jaggard, Bruno Blanchet, and Michael Backes for their contributions to our conference or journal papers. Furthermore, I would like to thank Paul Rowe for the fruitful discussions we had in our office.

During my work on this thesis I was partially supported by ONR Grants N00014-01-1-0795 and N00014-07-1-1039, and by NSF Grant CNS-0429689.

ABSTRACT

FORMAL ANALYSIS OF THE KERBEROS AUTHENTICATION PROTOCOL

Joe-Kai Tsay

Andre Scedrov, Advisor

The security of cryptographic protocols has traditionally been verified with respect to one of two mathematical models: One, known as the Dolev-Yao or symbolic model, abstracts cryptographic concepts into an algebra of symbolic messages. Methods based on the Dolev-Yao abstraction, which make use of simple formal languages or logics, have been successfully applied to discover structural flaws in numerous cryptographic protocols, and have also become efficient and robust enough to tackle large commercial protocols, often even automatically. The other, known as the computational or cryptographic model, retains the concrete view of messages as bitstrings and cryptographic operations as algorithmic mappings between bitstrings, while drawing security definitions from complexity theory. Proofs in the computational approach entail strong security guarantees, however, only simple cryptographic protocols, mainly of academic interest, have been verified with respect to the computational model.

This dissertation contributes to the ongoing case study of the Kerberos 5 protocol suite, a widely used authentication protocol. We report on a man-in-the-middle attack on PKINIT, the public key extension of Kerberos, which allows an

attacker to impersonate Kerberos administrative principals (KDC) and end-servers to a client and also gives the attacker the keys that the KDC would normally generate to encrypt the service requests of this client, hence defeating authentication and confidentiality guarantees of Kerberos. We have formally verified several possible fixes to PKINIT that prevent our attack using the symbolic Multiset Rewriting formalism. We also present proofs of the full Kerberos 5 suite with and without its public-key extension for the first time in the more detailed Dolev-Yao dialect of the BPW model. These proofs may be used to gain computationally sound results. Furthermore, we present a computationally sound mechanized analysis of Kerberos 5, both with and without its public-key extension PKINIT, using the prover CryptoVerif, which works directly in the computational model. We obtain proofs of authentication and key secrecy properties that are the first mechanical proofs of a full industrial protocol at the computational level. We generalize the notion of key usability, which, although weaker than the standard notion of key indistinguishability, guarantees under certain assumptions that a key can be securely used for cryptographic operations, and show that this definition is satisfied by keys in Kerberos.

Contents

Chapter 1

Introduction

Designing secure cryptographic protocols is a subtle task, which has proven to be remarkably error-prone; even if the underlying cryptographic primitives are assumed to be perfectly secure, one needs to reason about concurrent protocol sessions that run, typically, in an adversarial environment. There are many examples of protocols that were believed to be secure but were subsequently broken (*e.g.*, [63]). Security of such protocols is formally evaluated with respect to certain mathematical models - idealizations that allow a rigorous mathematical treatment. There are two main approaches to the verification of cryptographic protocols.

Computational approach

One approach, known as the *cryptographic* or *computational model* and based on the work by Goldwasser and Micali [50, 51], is based on complexity theory and retains the view of messages as bitstrings and cryptographic operations as (probabilistic)

algorithms on these bitstrings. Properties that are verified in this model hold in the presence of any probabilistic Turing machine attacking the protocol in polynomially many steps with respect to the the security parameter (which is a natural number that typically equals the key length). Results in the computational model are generally asymptotic and usually state that the advantage for an adversary to break a security property is *negligible*; *i.e.*, if η is a security parameter and $\mathbf{Adv}_{\mathcal{A},\Sigma}(\eta)$ is the probability for an adversary $\mathcal{A}$ to do something bad against a protocol Σ (*e.g.*, learning any information about a secret/key, breaking the authentication goal of a protocol, or breaking the encryption algorithm), then for any polynomial $p(x)$ there exists $N \in \mathbb{N}$ such that for all $n \geq N$ one has that $\mathbf{Adv}_{\mathcal{A},\Sigma}(n) \leq \frac{1}{p(n)}$. We note that results in the computational model do not give recommendations on how large a security parameter should to be chosen to yield some desired security. Although security properties proved in this model give strong security guarantees, only few protocols have been verified with respect to this model; most of those are rather of academic interest and not widely deployed. This is due to the fact that proofs are usually by reduction, *i.e.*, protocols are verified by showing that the existence of a feasible attack on the protocol can be reduced to a breach of one of the underlying cryptographic primitives.

Symbolic approach

Another approach, known as the *symbolic* or *Dolev-Yao model* and based on the work of Dolev and Yao [48], can be viewed as an idealization of the former approach formulated using an algebra of terms. Messages are abstracted as terms in this algebra and cryptographic operations are simply function symbols on these terms. Cryptography is assumed to be perfect, *i.e.*, one can only decrypt a ciphertext if one has the correct decryption key. The attacker can act non-deterministically but only according to a fixed set of rules (*e.g.*, the attacker can intercept/insert messages, but cannot guess nonces or keys) and there is no partial information available to the attacker (as atomic terms such as keys and nonces cannot be divided further into bits). This symbolic model has been successfully applied to uncover problems in the design of security protocols [37, 63, 64, 68]. Moreover, verification methods based on the symbolic model have become efficient and robust enough to be deployed for the analysis of even large commercial protocols [8, 30, 55, 64, 68]. Because by-hand proofs in the detailed computational model are prone to human error and are, even in the symbolic model, very time consuming for complex protocols, effort has been put into developing mechanized or fully automated provers. One area of focus has been provers that work in the symbolic model, which facilitates the use of existing theorem provers and model checkers, and some of the resulting tools have been used to analyze commercial protocols [6, 16, 23, 65]. Since the pioneering work by Abadi and Rogaway [3], attention has also been paid [13, 32, 44] to bridging the gap

between the symbolic and the computational models; in these frameworks proofs are carried out in a symbolic model, facilitating automation, and the results can be lifted to the computational model under certain conditions. However, provers based on computationally sound symbolic frameworks (*e.g.*, [41, 75]) are currently at most able to cope with academic protocols.

A third approach, the so called *information-theoretic approach*, is similar to the computational approach, where there is no time restriction on the adversary (*i.e.*, the attacker is not polynomially bounded). This approach goes back to work by Shannon [74] but is less commonly used and does not play a role in the present work. We note, however, that interest in this approach may may increase as the development of quantum cryptography progresses, as algorithms or protocols that are information theoretic secure do not depend on assumptions about computational hardness, which have not been proved.

Our Work

Here we report results on the formalization and analysis of the Kerberos 5 protocol [70], with and without its public-key extension PKINIT [57]: the discovery of a flaw in a draft version of PKINIT (which led to a Windows Security Bulletin [66]) and the symbolic proof that the fixed version is secure; by-hand proofs of the security of Kerberos using the Backes–Pfitzmann–Waidner (BPW) cryptographic library framework [8]; and computational sound mechanized proofs using

the prover CryptoVerif [22, 25, 26].

The flaw we have uncovered in PKINIT allows an attacker to impersonate the KDC, and therefore all the Kerberized services, to a user, hence defeating authentication of the server to the client. The attacker also obtains all the keys that the KDC would normally generate for the client to encrypt her service requests, hence compromising confidentiality as well. This is a protocol-level attack and was a flaw in the then-current specification, not just a particular implementation. In contrast to recently reported attacks on Kerberos 4 [79], our attack does not use an oracle, but is efficiently mounted in constant time by simply decrypting a message with one key, changing one important value, and re-encrypting it with the victim's public key. The consequences of this attack are quite serious. For example, the attacker could monitor communication between an honest client and a Kerberized network file server. This would allow the attacker to read the files that the client believes are being securely transferred to the file server.

Our attack is possible because the two messages constituting PKINIT were insufficiently bound to each other.[1] More precisely, the reply (the second message of this exchange) can easily be modified so that it appears to correspond to a request (the first message) sent by a client different from the one the reply is generated for. Assumptions required for this attack are that the attacker is a legal user, that he can intercept other clients' requests, and that PKINIT is used in "public-key encryption

[1]The possibility of an 'identity misbinding' attack was independently hypothesized by Ran Canetti, whom we consulted on some details of the specification

mode". The alternative "Diffie-Hellman (DH) mode" does not appear vulnerable to this attack; we are in the process of proving its full security.

After discovering the attack on PKINIT, we worked in close collaboration with the IETF Kerberos Working Group, in particular with the authors of the PKINIT specification documents, to correct the problem. Our contribution in this regard has been a formal analysis of a general countermeasure to this attack, as well as the particular instance proposed by the Working Group that has been adopted in the PKINIT specification [57]. Our attack led to an August 2005 Microsoft Security Bulletin and patch [66]. It was also recorded as a CERT advisory [34]. Here we extend our preliminary report [38] with additional authentication properties, full proofs of our results, and a more complete discussion of our techniques, formalization, and the protocol itself.

We also obtained security proofs of Kerberos and public-key Kerberos (*i.e.*, Kerberos with PKINIT in public-key mode) based on the Dolev-Yao style model of Backes, Pfitzmann, and Waidner [13, 14, 11] (called the *BPW model* henceforth). Kerberos is the largest and most complex protocol whose security has so far been inferred from a proof in this Dolev-Yao style approach. Earlier proofs in this approach were conducted mainly for small examples of primarily academic interest, e.g., the Needham-Schroeder-Lowe, the Otway-Rees, and the Yahalom protocols [7, 10, 12]. The BPW model provides cryptographically faithful symbolic abstractions of cryptographic primitives, i.e., the abstractions can be securely implemented using actual

cryptography. Thus, although proofs in the BPW model themselves are symbolic in nature, they refer to primitives from the BPW model and results are sound with respect to the computational level if the the actual implementations of cryptographic operations meet certain security definitions. However, the soundness of properties proved in the BPW model is out of the scope of this work. We will primarily focus on the symbolic proofs.

Furthermore, we gained computationally sound mechanized security proofs using CryptoVerif. Unlike the previously mentioned tools, CryptoVerif can verify protocols directly in the computational model. We note that CryptoVerif is different from ProVerif [23], a well-established tool which verifies protocols in the symbolic model; CryptoVerif is a next-generation prover. CryptoVerif proofs are presented as sequences of games in a probabilistic process calculus inspired by [60, 61, 62, 67]. Previously, CryptoVerif has only been used to analyze academic protocols [22, 25], so this work provides a test case for the suitability of CryptoVerif for analyzing real-world protocols. Kerberos and its public-key extension PKINIT (used in 'public-key mode' as discussed below) provide a particularly good test case because they incorporate many different design elements: symmetric and asymmetric encryption, digital signatures, and keyed hash functions. Using CryptoVerif's interactive mode, we are able to prove authentication and secrecy properties for Kerberos at the computational level. This suggests that CryptoVerif is capable of analyzing large-scale industrial protocols.

We note that although some similar work has been done on industrial protocols, e.g., [54], none of that work was for a protocol as complex as Kerberos and neither was it automated.

In proving confidentiality properties for Kerberos, we consider not only the standard notion of *key indistinguishability* but also the notion of *key usability* introduced in [44] (and which was proved by hand for Kerberos in [73]). This weaker confidentiality property ensures that a key is still 'good' for use in cryptographic operations, even though it might be distinguishable from a random bitstring. This type of property is important for protocols that, like Kerberos, perform operations with a key during a protocol run but then allow for the future use of this key; because the key has been used, it may be distinguishable from random, but that still may not help an attacker learn any information about messages that are later encrypted under that key. Here we define a notion of *strong key usability* that is less restrictive on the adversary's power than the original definition, and we use CryptoVerif to prove that certain keys in Kerberos satisfy this stronger version of key usability.

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

Related Work

Earlier work on analyzing Kerberos includes: analysis of Kerberos 4 (the previous version of Kerberos, which lacked the complexity of Kerberos 5 with PKINIT) by Bella and Paulson using Isabelle [17]; and Butler et al. gave symbolic proofs by hand of authentication and secrecy properties of basic intra-realm Kerberos 5 [28, 30] and of cross-realm authentication in basic Kerberos [30, 39]. Our work here extends these earlier analyses of Kerberos to use a mechanized tool on the full Kerberos protocol, with and without its public-key extension PKINIT; this represents the first computationally sound mechanized proof of a full industrial protocol, where we use the CryptoVerif prover [22, 25]. Although a variety of automated approaches exist [1, 5, 23, 24, 47, 78, 6, 42] and have also been applied to deployed protocols (*e.g.*, [58, 64, 68]), currently only CryptoVerif works directly in the computational model. CryptoVerif relies largely on a probabilistic polynomial-time process calculus developed by Lincoln et al. [61, 62].

Outline

Below, in Section 2.1 we recall the structure of Kerberos and give a detailed description of PKINIT. In Section 3.1 we provide an account of the attack we uncovered and outline its consequences. In Section 3.2 we discuss various approaches to prevent the attack, including the one adopted by the IETF Kerberos working group in response to our work. In Section 3.3 we review our representation language, MSR,

and use it to formalize the fixed version of PKINIT; we give some of our formal results—that the fixed version does prevent our attack and that both the broken and fixed versions have other authentication and secrecy properties—in Section 3.4.

In Section 4.1, we recall the BPW model (e.g., [9, 11, 14, 15]), and apply it to the specification of Kerberos 5 and Public-key Kerberos (i.e., Kerberos with PKINIT). Section 4.2 proves security results for these protocols.

Section 5.1 gives a brief overview of CryptoVerif. Section 5.3 briefly explains the syntax and semantics of CryptoVerif, including a sample of our formalization of Kerberos, and outlines the authentication and secrecy properties proved by CryptoVerif. Section 5.4 presents the details of these results and other aspects of our work, while Chapter 6 provides a summary and surveys areas for future work.

Chapter 2

Kerberos V5 and its Public-key Extension

2.1 Kerberos Overview

Kerberos [70] is a successful, widely deployed single sign-on protocol that is designed to authenticate clients to multiple networked services, *e.g.*, remote hosts, file servers, or print spoolers. Kerberos 5, the most recent version, is available for all major operating systems: Microsoft has included it in its Windows operating system, it is available for Linux under the name Heimdal, and commercial Unix variants as well as Apple's OS X use code from the MIT implementation of Kerberos 5. Furthermore, it is being used as a building block for higher-level protocols [77]. Introduced in the early 1990s [59], Kerberos 5 continues to evolve as new functionalities are

added to the basic protocol. One of these extensions, known as PKINIT, modifies the basic protocol to allow public-key authentication and in the process adds considerable complexity to the protocol. In chapter 3 we report a protocol-level attack on PKINIT and discuss the constructive process of fixing it. We have verified a few defenses against our attack, including one we suggested, a different one proposed in the IETF Kerberos working group (and included the final PKINIT specification), and a generalization of these two approaches.

A Kerberos session generally starts with a user logging onto a system. This triggers the creation of a client process that will transparently handle all her authentication requests. The initial authentication between the client and the Kerberos administrative principals (altogether known as the KDC, for Key Distribution Center) is traditionally based on a shared key derived from a password chosen by the user. PKINIT is intended to add flexibility, security and administrative convenience by replacing this static shared secret with two pairs of public/private keys, one assigned to the KDC and one belonging to the user. PKINIT is supported by Kerberized versions of Microsoft Windows, typically for use with smartcard authentication, including Windows 2000 Professional and Server, Windows XP, Windows Server 2003 and now Windows Vista [66]; it has also been included in Heimdal since 2002 [76]. The MIT reference implementation is being extended with PKINIT.

2.1.1 Basic Kerberos

The Kerberos protocol [70] allows a legitimate user to log on to her terminal once a day (typically) and then transparently access all the networked resources she needs in her organization for the rest of that day. Each time she wants to retrieve a file from a remote server, for example, Kerberos securely handles the required authentication behind the scene, without any user intervention.

We now review how Kerberos provides secure authentication based on a single logon. As we do this, we will be particularly interested in the initial exchange, which happens when the user first logs onto the system. Figure 2.1 gives an overview of the message flow for the entire Kerberos protocol; Figure 2.4 below refines this to show some of the message details for the basic protocol. We start this section with a detailed review of the first exchange in the protocol, both with and without PKINIT.

The client process—usually acting on behalf of a human user—interacts with three other types of principals during the three rounds of 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 first obtaining credentials, called the "ticket-granting ticket" (TGT), from a "Kerberos Authentication Server" (KAS) and then by presenting these credentials to a "Ticket-Granting Server" (TGS) in order to obtain a "service ticket" (ST), which is the credentials that the client finally presents to the application servers in order to authenticate

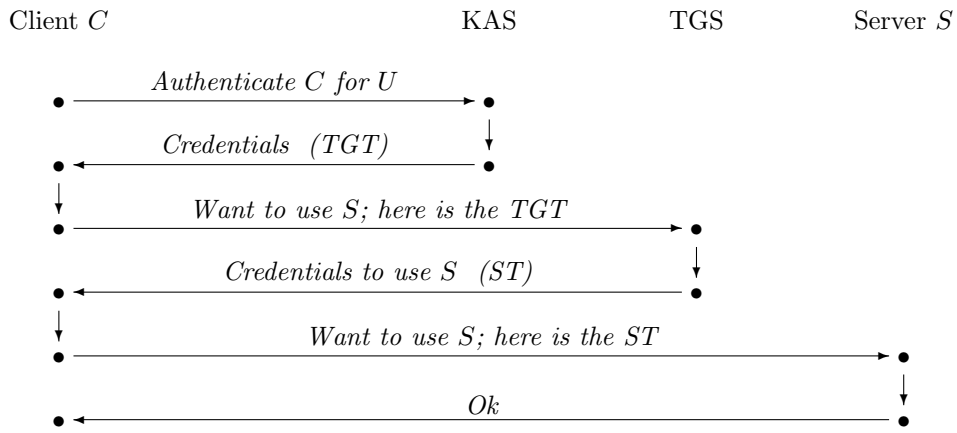

Figure 2.1: An Overview of Kerberos Authentication

herself (see Figure 2.1). 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 might be valid for a few minutes (although it, too, may be used repeatedly while it is still valid) and is used for a single application server. The KAS and the TGS are altogether known as the "Key Distribution Center" (KDC).

The client's interactions with the KAS, TGS, and application servers are called the Authentication Service (AS), Ticket-Granting (TG), and Client-Server (CS) exchanges, respectively. The focus of this work will be the AS exchange, as PKINIT does not alter the remaining parts of Kerberos.

The Traditional Authentication Service Exchange. The abstract structure of the messages in the traditional (non-PKINIT) AS exchange is given in Figure 2.2. 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

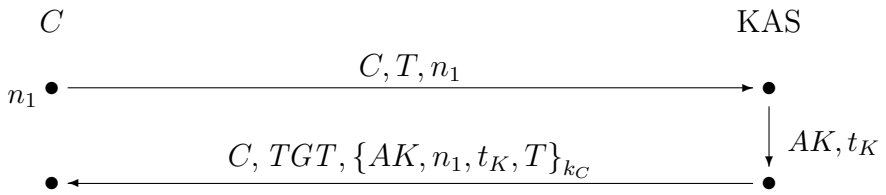

Figure 2.2: Message Flow in the Traditional AS Exchange where $TGT = \{AK, C, t_K\}_{k_T}$.

KAS. The KAS responds by generating a fresh key AK for use between the client and the TGS. This key is sent back to the client, along with the nonce (n_1) from the request, at timestamp (t_K) and other data, encrypted under a long-term key k_C shared between C and the KAS; this long-term key is usually derived from the user's password. We write $\{m\}_k$ for the encryption of m with symmetric key k. This is the only time that this long-term key is used in a standard Kerberos run because later exchanges use freshly generated keys. AK is also included in the TGT, sent alongside the message encrypted for the client. The TGT is encrypted under a long-term key k_T shared between the KAS and the TGS named in the request. 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, are more complex than the abstract view given here; the details we omit here do not affect our results and including them would only obscure their

exposition. We refer the reader to [70] for the complete specification of Kerberos 5, and to [28] for a formalization at an intermediate level of detail.

Security Considerations. One weakness of the standard Kerberos protocol is that the key k_C used to encrypt the client's credentials is derived from a password, and passwords are notoriously vulnerable to dictionary attacks [70]. Moreover, since the initial request is entirely plaintext, an active attacker can repeatedly make requests for an honest client's credentials and amass a large quantity of plaintext-ciphertext pairs, the latter component being encrypted with the client's long-term key k_C. While the attacker is unable to use these credentials to authenticate to the system, he is given considerable opportunity to perform an active dictionary attack against the key.

Kerberos can optionally use pre-authentication, a feature that is designed to prevent an attacker from actively requesting and obtaining credentials for an honest user. In brief, pre-authentication works by requiring the client to include a timestamp encrypted with her long-term key (k_C) in the initial request. The authentication server will only return credentials if the decrypted timestamp is sufficiently recent. This method successfully prevents an attacker from actively obtaining ciphertext encrypted with the long-term key; however, it does not prevent passive dictionary attacks, *i.e.*, a passive attacker could eavesdrop on network communications, record credentials as the honest client requests them, and attempt off-line dictionary decryption. Thus, pre-authentication makes it slower for an attacker to

perform cryptanalysis against the user's long-term key, but it does not fully prevent the vulnerability. One goal of PKINIT is to eliminate the possibility of this dictionary attack.

2.1.2 Public-Key Kerberos

PKINIT [57] 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. As we just saw, 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, PKINIT allows network administrators to use an existing public key infrastructure (PKI) rather than expend additional effort to manage users' long-term keys needed for 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 possess independent public/secret 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 are replaced with automatically generated asymmetric keys. 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 [45]. Other possibilities include keeping these credentials on the user's hard drive, again protected by a passphrase.

The manner in which PKINIT works depends on both the protocol version and the mode invoked. As the PKINIT extension to Kerberos has recently been published as RFC 4556 after a sequence of Internet Drafts [57], we use "PKINIT-n" to refer to the protocol as specified in the n^{th} draft revision and "PKINIT" for the protocol more generally. These various drafts and the RFC can be found at [57]. We discovered the attack described in Section 3.1 when studying PKINIT-25; our description of the vulnerable protocol is based on PKINIT-26, which does not differ from PKINIT-25 in ways that affect the attack. In response to our work described here, PKINIT-27 included a defense against our attack; we discuss this fix in Section 3.2. The version of the protocol defined in RFC 4556 does not differ from the parts of PKINIT-27 that we discuss here.

PKINIT can operate in two modes. In *public-key encryption mode*, the key pairs (pk_C, sk_C) and (pk_K, sk_K) are used for both signature and encryption. The latter is designed to (indirectly) protect the confidentiality of AK, while the former ensures its integrity. In *Diffie-Hellman (DH) mode*, the key pairs are used to provide digital signature support for an authenticated Diffie-Hellman key agreement which is used to protect the fresh key AK shared between the client and KAS. A variant of this mode allows the reuse of previously generated shared secrets. We will not discuss the DH mode in detail as our preliminary investigation did not reveal any flaw in it; we are currently working on a complete analysis of this mode. Furthermore, it appears not to have yet been included in any of the major operating systems. The only support we are aware of is within the PacketCable system [31], developed by CableLabs, a cable television research consortium.

Figure 2.3 illustrates the AS exchange in public-key encryption mode as of PKINIT-26. The differences with respect to the traditional AS exchange (see Figure 2.2) have been highlighted using boxes. In discussing this and other descriptions of the protocol, we write $[m]_{sk}$ for the digital signature of message m with secret key sk. (PKINIT realizes digital signatures by concatenating the message and a keyed hash for it, occasionally with other data in between.) In our analysis of PKINIT in Section 3.4, we make the standard assumption that digital signatures are unforgeable [52]. The encryption of m with public key pk is denoted $\{\!|m|\!\}_{pk}$. As before, we write $\{m\}_k$ for the encryption of m with symmetric key k.

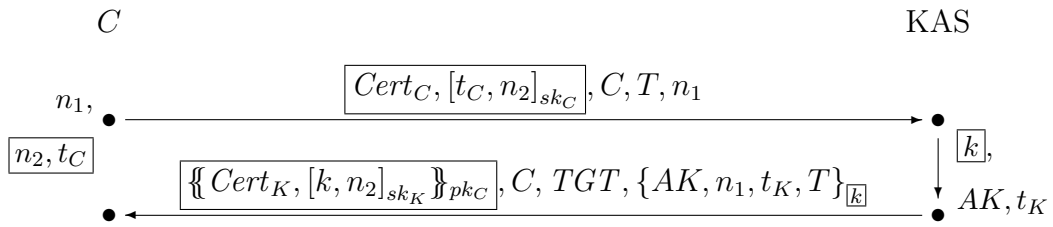

Figure 2.3: Message Flow in PKINIT-26, where $TGT = \{AK, C, t_K\}_{k_T}$.

The first line of Figure 2.3 describes the relevant parts of the request that a client C sends to a KAS K using PKINIT-26. The last part of the message—C, T, n_1—is exactly as in basic Kerberos 5, containing the client's name, the name of the TGS for which she wants a TGT, and a nonce. The boxed parts added by PKINIT include 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 to the left of this line indicate that these are generated by C specifically for this request, with the box indicating data not included in our abstract formalization of basic Kerberos 5 [28, 30].) This effectively implements a form of pre-authentication.

The second line in Figure 2.3 shows our formalization of K's response, which is more complex than in basic Kerberos. The last part of the message—C, TGT, $\{AK, n_1, t_K, T\}_{\boxed{k}}$—is very similar to K's reply in basic Kerberos; the difference (boxed) is that the symmetric key k protecting AK is now freshly generated by K and not a long-term shared key. The ticket-granting ticket TGT and the message encrypted under k are as in traditional Kerberos. 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 $\{\!\{Cert_K, [k, n_2]_{sk_K}\}\!\}_{pk_C}$. This contains K's certificates and his signature, using his secret key sk_K, over k and the nonce n_2 from C's request; all of this is encrypted under C's public key pk_C.

This abstract description leaves out a number of fields which are of no significance with respect to the reported attack or its fix. We invite the interested reader to consult the specifications [57]. Also, recall that PKINIT leaves the subsequent exchanges of Kerberos unchanged.

2.1.3 Message Flow in Later Exchanges

For the sake of completeness, we give a brief overview of the message structure in the remaining rounds of Kerberos; Section 3.1.3 discusses how the attack on PKINIT can be propagated through these later rounds. Figure 2.4 updates Figure 2.1 to show the details of the messages in basic Kerberos. PKINIT modifies the first two of these messages as illustrated in Figure 2.3.

The AS exchange (either traditional or with PKINIT) provides the client with an authentication key AK and a ticket granting ticket $TGT = \{AK, C, t_K\}_{k_T}$.

In the TG exchange, C then requests an ST from T after generating a new nonce n_3 and timestamp; her request includes the TGT (which she cannot read but simply forwards from K), an authenticator $\{C, t_C\}_{AK}$ containing her name and timestamp and encrypted with AK, the name of the server S for which she wants an ST, and the new nonce. T's response has the same structure as K's, but now with an ST in

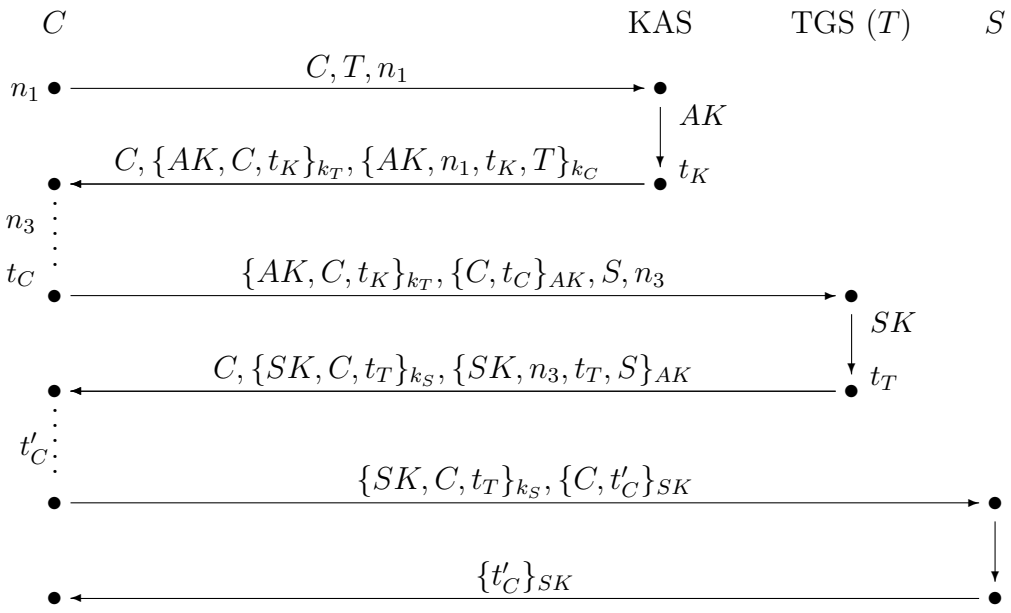

Figure 2.4: Message flow in basic Kerberos

place of the TGT and S taking the role of T in the rest of the message.

Finally, in the CS exchange, C authenticates herself to S by sending the ST and an authenticator $\{C, t'_C\}_{SK}$ containing her name and timestamp, and encrypted under the fresh key SK. S may authenticate himself back to C by encrypting C's timestamp (but not C's name, so that the result differs from the authenticator) with SK and returning this message to C.

Chapter 3

Breaking and Fixing Public-key Kerberos

3.1 The Attack

In this section, we report on a dangerous attack against PKINIT in public-key encryption mode. We discovered this attack as we were interpreting the specification documents of this protocol [57] in preparation for its formalization in MSR [39, 36, 49], the specification language for our analysis. We start with a detailed description of the attacker's actions in the AS exchange, the key to the attack. We then review the conditions required for the attack and close this section with a discussion of how the attacker may propagate the effects of her AS exchange actions throughout the rest of a protocol run.

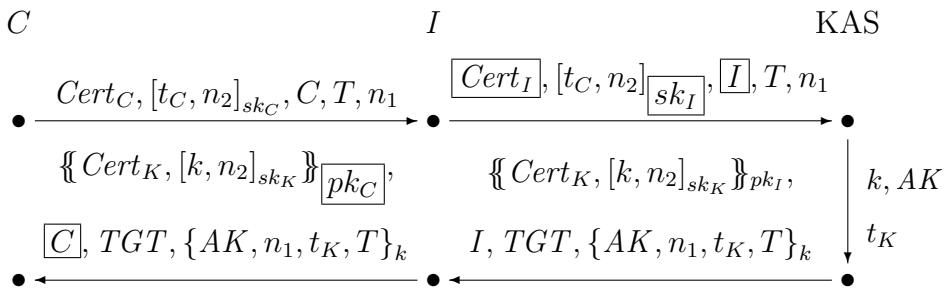

Figure 3.1: Message Flow in the Man-In-The-Middle Attack on PKINIT-26, where $TGT = \{AK, I, t_K\}_{k_T}$.

3.1.1 Message Flow

Figure 3.1 shows the AS exchange message flow in the attack. The client C sends a request to the KAS K which is intercepted by the attacker I, who constructs his own request message using the parameters from C's message. All data signed by C are sent unencrypted—indeed $[msg]_{sk}$ can be understood as an abbreviation for the plaintext msg together with a keyed hash of the message—so that I may generate his own signatures over data from C's request. The result is a well-formed request message from I, although constructed using some data originating with C. I's changes to the request message are boxed above the top-right arrow of Figure 3.1. (We have omitted an unkeyed checksum taken over unencrypted data from these messages; I can regenerate this as needed to produce a valid request.)

I forwards the fabricated request to the KAS K, who views it as a valid request for credentials if I is himself a legitimate client; there is nothing to indicate that some of the data originated with C. K responds with a reply containing credentials for I (the bottom-right arrow in Figure 3.1). The TGT has the form $\{AK, I, t_K\}_{k_T}$;

note that, because it is encrypted with the key k_T shared between K and the TGS T, it is opaque to C (and I). Another part of the reply is encrypted using the public key of the client for whom the credentials are generated, in this case I. This allows the attacker to decrypt this part of the message using his private key, learn the key k, and use this to learn the key AK. An honest client would only use this information to send a request message to the TGS T. Instead, I uses C's public key to re-encrypt the data he decrypted using his private key (having learned pk_C, if necessary, from $Cert_C$ in the original request), replaces his name with C's, and forwards the result to C. To C this message appears to be a valid reply from K generated in response to C's initial request (recall that C cannot read I's name inside the TGT).

At this point, C believes she has authenticated herself to the KAS and that the credentials she has obtained—the key AK and the accompanying TGT—were generated for her. However, the KAS has completed the PKINIT exchange with I and has generated AK and the TGT for I. The attacker knows the key AK (as well as k, which is not used other than to encrypt AK) and can therefore decrypt any message that C would protect with it.

Protocol-level attacks in the same vein of the vulnerability we uncovered have been reported in the literature for other protocols. In 1992, Diffie, van Oorschot, and Wiener noted that a signature-based variant of the Station-to-Station protocol [46] could be defeated by a man-in-the-middle (MITM) attack which bears similarities

to what we observed in the first half of our vulnerability; in 2003 Canetti and Krawczyk [33] observed that the "basic authenticated Diffie-Hellman" mode of the Internet Key Exchange protocol (IKE) [53] had this very same vulnerability. In 1996, Lowe [63] found an attack on the Needham-Schroeder public key protocol [69] that manipulates public key encryption essentially in the same way as what happens in the second half of our attack. Because it alters both signatures and asymmetric encryptions, our attack against PKINIT stems from both [63] and [46]. In 1995, Clark and Jacob [40] discovered a similar flaw on Hwang and Chen's corrected SPLICE/AS protocol [56].

3.1.2 Assumptions

In order for this attack to work, the attacker must be a legal Kerberos client so that the KAS will grant him credentials. In particular, he must possess a public/secret key pair (pk_I, sk_I) and valid certificates $Cert_I$ trusted by the KAS. The attacker must also be able to intercept messages, which is a standard assumption. Finally, PKINIT must be used in public-key encryption mode, which is commonly done as the alternative DH mode does not appear to be readily available, except for domain specific systems [45, 31].

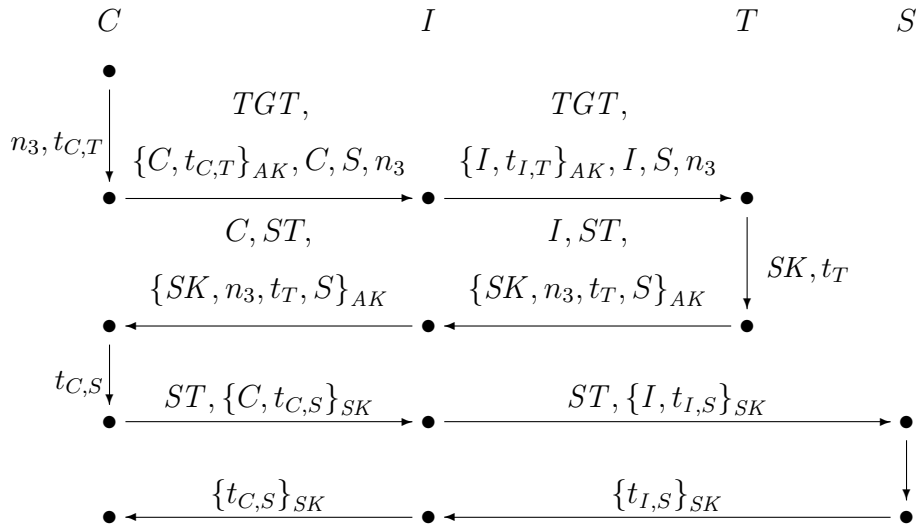

Figure 3.2: Message flow in the man-in-the-middle attack on PKINIT-26, after the messages in Figure 3.1, when the attacker forwards and observes traffic; here $TGT = \{AK, I, t_K\}_{k_T}$ and $ST = \{SK, I, t_T\}_{k_S}$.

3.1.3 Effects of the Attack

Attacker Observes Traffic

Once the attacker learns AK in the AS exchange, he may either mediate C's interactions with the various servers (essentially logging in as I while leaking data to C so she believes she has logged in) while observing this traffic or simply impersonate the servers in the later exchanges. In the first variant, which is shown in Figure 3.2, once C has AK and a TGT, she would normally contact the TGS to get an ST for some application server S. This request contains an *authenticator* of the form $\{C, t_{C,T}\}_{AK}$ (*i.e.*, C's name and a timestamp, encrypted with AK). Because I knows AK, he may intercept the request and replace the authenticator with one

that refers to himself: $\{I, t_{I,T}\}_{AK}$. The reply from the TGS contains a freshly generated key SK; this is encrypted under AK, for C to read and thus accessible to I, and also included in an ST that is opaque to all but the TGS and application server. I may intercept this message and learn SK, replace an instance of his name with C's name, and forward the result to C. As I knows SK, he can carry out a similar MITM attack on the CS exchange, replacing the authenticator $\{C, t_{C,S}\}_{SK}$ with the authenticator $\{I, t_{I,S}\}_{SK}$ and then replacing the server's reply $\{t_{I,S}\}_{SK}$ with the reply $\{t_{C,S}\}_{SK}$ that the client is expecting. This exchange ostensibly authenticates C to the application server; however, because the service ticket names I, this server would believe that he is interacting with I, not C.

Attacker Impersonates Servers

Alternatively, the attacker may intercept C's requests in the TG and CS exchanges and impersonate the involved servers rather than forwarding altered messages to them; the message flow for this version of the attack is shown in Figure 3.3. In the TG exchange, I will ignore the TGT and only decrypt the portion of the request encrypted under AK (which he learned during the initial exchange). The attacker will then generate a bogus service ticket X_{ST}, which the client expects to be opaque, and a fresh key SK encrypted (along with other data n_3, t_T, S) under AK, and send these to C in what appears to be a properly formatted reply from the TGS. In the CS exchange the attacker may again intercept the client's request;

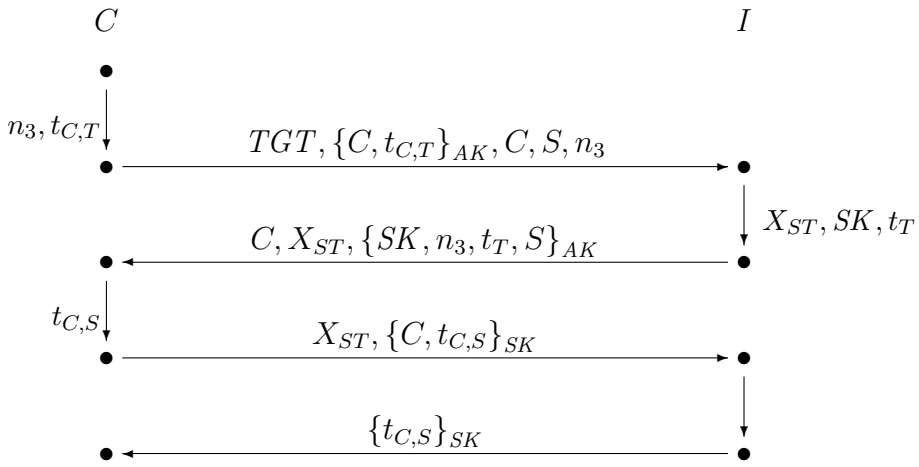

Figure 3.3: Message Flow in the Man-In-The-Middle Attack on PKINIT-26, after the messages in Figure 3.1, when the attacker impersonates the TGS and end server; here $TGT = \{AK, I, t_K\}_{k_T}$ while X_{ST} is a garbage message.

in this case, no new keys need to be generated, and the attacker only needs to return the client's timestamp encrypted under SK—which I himself generated in the previous exchange—for C to believe that she has completed this exchange with the application server S. Note that the attacker may take the first approach—mediating the exchange between C and a TGS—in the TG exchange and then the second—impersonating the application server—in the CS exchange. The reverse is not possible because I cannot forge a valid ST for S when impersonating T.

Regardless of which approach the attacker uses to propagate the attack throughout the protocol run, C finishes the CS exchange believing that she has interacted with a server S and that T has generated a fresh key SK known only to C and S. Instead, I knows SK in addition to, or instead of, S (depending on how I propagated the attack). Thus I may learn any data that C attempts to send to S;

depending on the type of server involved, such data could be quite sensitive. Note that this attack does not allow I to impersonate C to a TGS or an application server because all involved tickets name I; Section 3.4.4 discusses a related authentication property. This also means that if C is in communication with an actual server (T or S), that server will view the client as I, not C.

3.2 Preventing the Attack

The attack outlined in the previous section was possible because the two messages constituting the then-current version of PKINIT were insufficiently bound to each other. More precisely, the attack shows that, although a client can link a received response to a previous request (thanks to the nonces n_1 and n_2, and to the timestamp t_C), she cannot be sure that the KAS generated the key AK and the ticket granting ticket TGT appearing in this response *for her*. Indeed, the only evidence of the principal for whom the KAS generated these credentials appears inside the TGT, which is opaque to her. This suggests one approach to making PKINIT immune to this attack, namely to require the KAS to include the identity of this principal in a component of the response that is integrity-protected and that the client can verify. An obvious mechanism is the submessage signed by the KAS in the reply.

Following a methodology we successfully applied in previous work on Kerberos [28, 39], we have constructed a formal model of both PKINIT-26 and various

possible fixes to this protocol (including the one adopted in PKINIT-27). Details can be found in Section 3.4. Property 1 informally states that PKINIT-27 and subsequent versions satisfy a security property that we see violated in PKINIT-26, demonstrating that this fix does indeed defend against our attack.

Property 1. *In PKINIT-27 (and subsequent versions), whenever a client C processes an AS reply containing server-generated public-key credentials, the KAS previously produced such credentials for C.*

This property informally expresses the contents of Corollary 3.4.4, which we prove in Section 3.4.

As we worked on our formal analysis, we solicited feedback from the IETF Kerberos Working Group, and in particular the authors of the PKINIT specifications, about possible fixes we were considering. We also analyzed the fix, proposed by the Working Group, that was included in PKINIT-27 and subsequent revisions of this specification [57].

3.2.1 Abstract Fix

Having traced the origin of the discovered attack to the fact that the client cannot verify that the received credentials (the TGT and the key AK) were generated for her, the problem can be fixed by having the KAS include C's name in the reply in such a way that it cannot be modified *en route* and that C can check it. Following well-established recommendations [2], we initially proposed a simple and

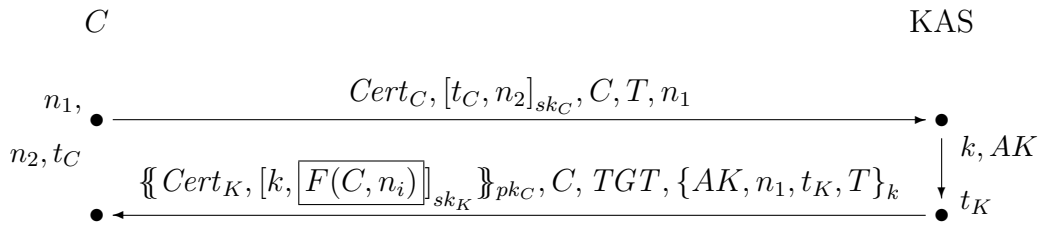

Figure 3.4: Abstract fix of PKINIT

minimally intrusive approach to doing so: including C's name in the portion of the reply signed by the KAS (in PKINIT-26, this is $[k, n_2]_{sk_K}$). We then generalized it by observing that the KAS can sign k and any message fragment $F(C, n_i)$ that is suitably built from C's name and at least one of the nonces n_1 and n_2 from C's request for credentials. With this abstract fix in place, the PKINIT exchange in public-key encryption mode is depicted in Figure 3.4, where we have used a box to highlight the modification with respect to PKINIT-26. Here F represents any construction that injectively involves C and either n_1 or n_2—*i.e.*,$F(C, n_i) = F(C', n_i')$ implies $C = C'$ and $n_i = n_i'$—and is verifiable by the client. Integrity protection is guaranteed by the fact that it appears inside a component signed by the KAS, and therefore is non-malleable by the attacker (assuming that the KAS's signature keys are secure). Intuitively, this defends against the attack since the client C can now verify that the KAS generated the received credentials for her and not for another principal (such as I in our attack). Indeed, an honest KAS will produce the signature $([k, F(C, n_i)]_{sk_K})$ only in response to a request from C. The presence of the nonces n_1 or n_2 uniquely identifies which of the (possibly several)

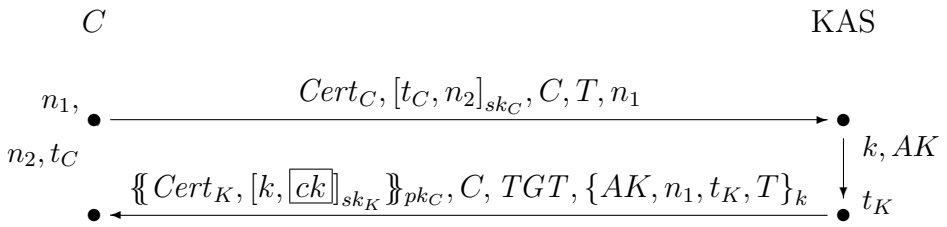

Figure 3.5: Fix of PKINIT adopted in version 27

requests from C to which this reply corresponds. Note that the fact that we do not need F to mention both n_1 and n_2 entails that the nonce n_2 is superfluous as far as authentication is concerned. We will formally prove that this variant defends against the attack in Section 3.4.

A simple instance of this general schema consists in taking $F(C, n_i)$ to be (C, n_2), yielding the signed data $[k, C, n_2]_{sk_K}$, which corresponds to simply including C's name within the signed portion of the PKINIT-26 reply. Its correctness will follow as a simple corollary of the validity of our general schema, as we will show in Section 3.4.

3.2.2 Solution Adopted in PKINIT-27

When we discussed our initial fix with the authors of the PKINIT document, we received the request to apply our methodology to verify a different solution: rather than simply including C's name in the signed portion of the reply, replace the nonce n_2 there with a keyed hash ("checksum" in Kerberos terminology) taken over the client's entire request. We did so and showed that this approach also defeats our

attack. It is on the basis of this finding that we distilled the general fix discussed above, of which both solutions are instances.

The IETF Kerberos Working Group later decided to include the checksum-based approach in PKINIT-27 and its subsequent revisions [57]. The message flow of this version of PKINIT is displayed in Figure 3.5. Here, ck is a checksum of the client's request keyed with the key k, *i.e.*, ck has the form $H_k(Cert_C, [t_C, n_2]_{sk_C}, C, T, n_1)$ where H is a preimage-resistant MAC function. This means that it is computationally infeasible for the attacker to find a message whose checksum matches that of a given message. Following the specifications in [72], which discusses cryptographic algorithms for use in the Kerberos protocol suite, current candidates for H include hmac-sha1-96-aes128. New strong keyed checksums can be used for ck as they are developed.

3.3 Multiset Rewriting for Security Protocols

We have formalized PKINIT in the language MSR [39, 36, 49] and used this specification to verify the correctness of the proposed fixes. MSR is a flexible framework for specifying complex cryptographic protocols, possibly structured as a collection of coordinated subprotocols. It uses strongly-typed multiset rewriting rules over first-order atomic formulas to express protocol actions and relies on a form of existential quantification to symbolically model the generation of fresh data (*e.g.*, nonces or short-term keys).

3.3.1 Terms and Types

MSR represents network messages and their components as first-order terms. Thus the TGT $\{AK, C, t_K\}_{k_T}$ sent from K to C is modeled as the term obtained by applying the binary encryption symbol $\{\_\}_\_$ to the constant k_T and the subterm (AK, C, t_K). This subterm is built using atomic terms and two applications of the binary concatenation symbol ("$\_,\_$"). For simplicity, we retain the semi-formal message syntax used earlier. Terms are classified by types, which describe their intended meaning and restrict the set of terms that can be legally constructed. For example, $\{\_\}_\_$ accepts a key (type key) and a message (type msg), producing a term of type msg; using a nonce as the key yields an ill-formed term. Nonces, principal names, *etc.*, often appear within messages; MSR uses the subsort relation to facilitate this. For example, defining nonce to be a subsort of msg (written nonce $<:$ msg) allows nonces to be treated as messages. Both term constructors and types are definable. This allows us to formalize the specialized principals of Kerberos 5 as subsorts of the generic principal type: we introduce types client, KAS, TGS and server, with the obvious meanings.

MSR supports more structured type definitions [36]. Dependent types allow capturing the binding between a key and the principals for whom it was created. For example, the fact that a short-term key k is intended to be shared between a particular client C and server S is expressed by declaring it to be of type shK C S. Because k is a key, shK C S is a subsort of key (for all C and S), and since k is short

term this type is also a subsort of msg as k needs to be transmitted in a message. We similarly model the long-term keys that a principal A shares with the KAS as objects of type dbK A, which is again a subsort of key but *not* of msg; these keys are not intended to be sent over the network, and this typing prohibits this. Dependent types give us elegant means to describe the public-key machinery. If (pk, sk) is the public/secret key pair of principal A, we simply declare pk of type pubK A and sk of type secK pk. Secret keys, like the long-term keys, are not intended to be sent over the messages; thus the type secK pk is a subsort of key but not of msg. Although we do not explicitly include public keys in messages here, pubK A is a subsort of msg. The constructors for encryption and digital signature are written $\{\!|m|\!\}_{pk}$ and $[m]_{sk}$, respectively, as in the text so far.

Other types used in the formalization of PKINIT include time for timestamps, CertList for lists of digital certificates, and someSecK as an auxiliary type for working with digital signatures. The use of someSecK allows us, *e.g.*, to model a signed message without declaring the public key or its owner that correspond to the signing key; in order to verify the signature we use the predicate *VerifySig* together with a list of certificates (as in Figure 3.6) instead of a specific public key. We also use the constructor $H_k(m)$ to model the checksum (keyed hash) of message m keyed with symmetric key k.

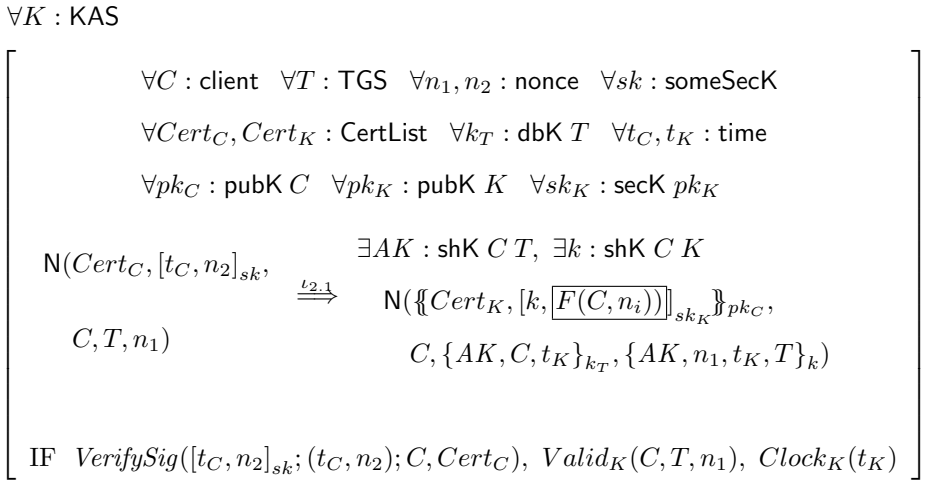

Figure 3.6: KAS's Role in the abstract fix version of the PKINIT AS Exchange

3.3.2 States, Rules, and the Formalization of PKINIT

The *state* of a protocol execution is determined by the network messages in transit, the local knowledge of each principal, and other similar data. MSR formalizes individual bits of information in a state by means of *facts* consisting of *predicate name* and one or more terms. For example, the network fact $\mathsf{N}(\{AK, C, t_K\}_{k_T})$ indicates that the term $\{AK, C, t_K\}_{k_T}$, a TGT, is present on the network, and $I(\{AK, C, t_K\}_{k_T})$ that it is known by the attacker.

A protocol consists of actions that transform the state. In MSR, this is modeled by the notion of *rule*: a description of the facts that an action removes from the current state and the facts it replaces them with to produce the next state. For example, Figure 3.6 describes the actions of the KAS in the abstract fix of PKINIT (see Section 3.2) where $F(C, n_i)$ stands for a construction that contains C and either n_1 or n_2 (or both). Ignoring for the moment the leading $\forall K : \mathsf{KAS}$ and

the outermost brackets leaves us with a single MSR rule—labeled $\iota_{2.1}$ above the arrow—that we will use to illustrate characteristics of MSR rules in general.

Rules are parametric, as evidenced by the leading string of typed universal quantifiers: actual values need to be supplied before applying the rule. The middle portion ($\cdots \Longrightarrow \cdots$) describes the transformation performed by the rule: it replaces states containing a fact of the form $\mathsf{N}(Cert_C, [t_C, n_2]_{skC}, C, T, n_1)$ with states that contain the fact on its right-hand side but which are otherwise identical. The existential marker "$\exists AK : \mathsf{shK}\, C\, T$" requires AK to be replaced with a newly generated symbol of type $\mathsf{shK}\ C\ T$, and similarly for "$\exists k : \mathsf{shK}\ C\ K$"; this is how freshness requirements are modeled in MSR. The last line, starting with the keyword IF, further constrains the applicability of the rule by requiring that certain predicates be present (differently from the left-hand side, they are not removed as a result of applying the rule). Here, we use the predicates *VerifySig* to verify that a digital signature is valid given a list of credentials ($\mathit{VerifySig}(s; m; P, Certs)$ holds if s is the signature, relative to certificates $Certs$, by principal P over the message m). Additionally, we use Valid_K to capture the local policy of K in issuing tickets, and Clock_K to model the local clock of K. While the entities following 'IF' are logically facts, in practice they are often handled procedurally, outside of MSR.

Rule $\iota_{2.1}$ completely describes the behavior of the KAS; in general, multiple rules may be needed, as when modeling the actions of the client in the AS exchange. Coordinated rules describing the behavior of a principal are collected in

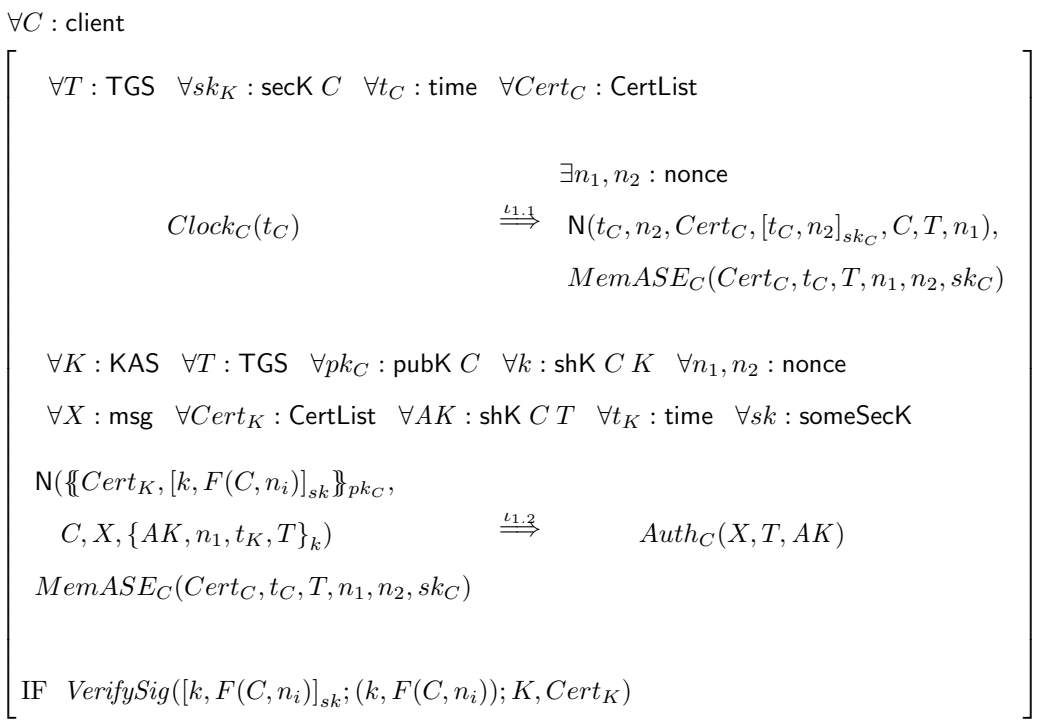

Figure 3.7: Client's Role in the abstract fix version of the PKINIT AS Exchange

a *role*. A role is just a sequence of rules, parameterized by the principal executing them (their *owner*)—the "$\forall K$: KAS" above the brackets in Figure 3.6. The two-rule role describing the client's actions of the abstract fix version of PKINIT is displayed in Figure 3.7. Formalizations of the TG and CS exchanges can be found in Appendix .1, with detailed explanations in [28, 39, 29].

3.3.3 Execution Semantics

Going into more detail, the execution semantics of MSR operates by transforming not states (as introduced in the previous section) but *configurations* of the form $\langle S \rangle^R_\Sigma$, where S is a state, the *signature* Σ contains the symbols being used (with

their type), and the *active role set* $R = (\rho_1^{a_1}, \ldots, \rho_n^{a_n})$ records the remaining actions of the currently present roles (ρ_i) together with the principals executing them (a_i).

Basic execution steps are expressed by judgments of the form $\mathcal{P} \triangleright C \longrightarrow C'$ where $\mathcal{P}$ is the protocol specification, and C and C' are consecutive configurations. These judgments are defined by the following two rules (which are somewhat simplified from [35]):

$$\frac{}{(\mathcal{P}, \rho) \triangleright \langle S \rangle_\Sigma^R \longrightarrow \langle S \rangle_\Sigma^{R,\rho^a}}\ \textbf{inst}$$

$$\frac{}{\mathcal{P} \triangleright \langle S, [\theta] lhs \rangle_\Sigma^{R,((lhs \Rightarrow \exists \vec{x}.rhs),\rho)^a} \longrightarrow \langle S, [\theta, \vec{c}/\vec{x}] rhs \rangle_{\Sigma,\vec{c}}^{R,\rho^a}}\ \textbf{rw}$$

The rule **inst** prepares a role for execution by inserting it in the active role set: it associates the role with the principal a that will be executing it. The same role can be loaded arbitrarily many times by any principal (subject to some typing limitations), which provides support for the concurrent execution of multiple sessions and therefore also for multi-session attacks. The inference **rw** describes the application of a state transforming rule $r = lhs \Rightarrow rhs$ introduced in the previous section: if an instance $[\theta]lhs$ of its left-hand side appears in the state, it is replaced by the corresponding instantiation of the right-hand side of r after instantiating the existential variables $\vec{x}$ with new constants $\vec{c}$. The rule r is then removed from the active role set of the configuration, and $\vec{c}$ is added to the signature.

In order to present rule application in a compact yet precise way, the notion of an *abstract execution step* is used which denotes a quadruple $C \xrightarrow{r,\iota} C'$. Here, C and

C' are consecutive configurations, r identifies the rule from $\mathcal{P}$ being executed, and ι stands for the overall substitution $[\theta, \vec{c}/\vec{x}]$ above. We say that r is *applicable* in C if there exist a substitution ι and a configuration C' such that $C \xrightarrow{r,\iota} C'$ is defined.

A *trace* $\mathcal{T}$ is then a sequence of the form

$$C_0 \xrightarrow{r_1,\iota_1} C_1 \xrightarrow{r_2,\iota_2} \cdots \xrightarrow{r_n,\iota_n} C_{n+1}$$

where $C_0 = \langle S_0 \rangle_{\Sigma_0}^{R_0}$ is called the *initial configuration* of $\mathcal{T}$. In the context of Kerberos, the state component S_0 of the initial configuration contains only the predicates used in constraints (*e.g.*, *VerifySig* in Figure 3.7), and the intruder's knowledge (see next section); in particular, no network message or memory predicate is contained in S_0. The initial signature Σ_0 within C_0 contains the names of all principals together with their types (client, server, KAS, *etc.*), and their keys, *etc.* The initial active role set R_0 within C_0 is empty. Note that we only need to consider finite traces, because execution (in any networked computer system) proceeds by discrete steps and started at a specific moment in time. Likewise, a generic trace will contain only a finite, although *a priori* unbounded, number of applications of inference **inst** for the same role ρ.

3.3.4 Intruder Model

The intruder model in our analysis of public-key Kerberos is a variant of the classic Dolev-Yao intruder model [69, 48]. Traditionally, the attacker in this model can non-deterministically intercept and originate network traffic, encrypt and decrypt

captured messages as long as he knows the correct key, concatenate and split messages at will, generate certain types of messages (*e.g.*, keys and nonces) but not others (*e.g.*, principals), and access public data such as principal names.

This set of intruder capabilities is given a precise specification in MSR. For this purpose, the knowledge of the intruder is modeled as a collection of facts $I(m)$ ("the intruder knows message m") distributed in the state. The intruder himself is represented as the distinguished principal I by means of the declaration $\mathsf{I} : \mathsf{principal}$. Each capability is expressed by means of a one-rule role that can be executed only by I. For example, the specifications of network message interception, message splitting and nonce generation have the following form:

$$\mathsf{I}\left[\ \forall m : \mathsf{msg}\quad \mathsf{N}(m) \overset{INT}{\Longrightarrow} I(m)\ \right] \qquad \mathsf{I}\left[\ \forall m_1, m_2 : \mathsf{msg}\quad I(m_1),\ I(m_2) \overset{CPM}{\Longrightarrow} I(m_1, m_2)\ \right] \qquad \mathsf{I}\left[\ . \overset{NG}{\Longrightarrow} \exists n : \mathsf{nonce}\quad I(n)\ \right]$$

Notice that these roles identify I as their owner (above the left brace) rather than a generic principal (introduced by a universal quantifier in the previous section).

Because public-key Kerberos relies on more than shared keys, we extend our earlier intruder formalization with the following rules for public-key encryption and decryption. These allow the intruder to learn any public key of any principal P, or to learn any of his own secret keys (but not those of other principals), and then to

encrypt and decrypt if the proper keys are known.

$$\mathsf{I}\left[\begin{array}{l}\forall P : \mathsf{principal}\ \ \forall pk : \mathsf{pubK}\ P \\ \qquad . \overset{PK'}{\Longrightarrow} I(pk)\end{array}\right] \qquad \mathsf{I}\left[\begin{array}{l}\forall pk : \mathsf{pubK}\ I\ \ \forall sk : \mathsf{secK}\ sk \\ \qquad . \overset{SK'}{\Longrightarrow} I(sk)\end{array}\right]$$

$$\mathsf{I}\left[\begin{array}{l}\forall P : \mathsf{principal}\ \ \forall pk : \mathsf{pubK}\ P \\ \forall m : \mathsf{msg} \\ I(m), I(pk) \overset{PEC'}{\Longrightarrow} I(\{\!|m|\!\}_{pk})\end{array}\right] \qquad \mathsf{I}\left[\begin{array}{l}\forall P : \mathsf{principal}\ \ \forall pk : \mathsf{pubK}\ P \\ \forall sk : \mathsf{secK}\ pk\ \ \forall m : \mathsf{msg} \\ I(\{\!|m|\!\}_{pk}), I(sk) \overset{PDEC'}{\Longrightarrow} I(m)\end{array}\right]$$

Signing is handled similarly as shown in rule *SIG* below. Rather than having the intruder verify signatures, an activity an attacker will rarely bother with (although it could easily be modeled in MSR), rule *PEEK* allows him to extract a message from a signature. It should be noted that this specification deemphasizes the PKI infrastructure on which PKINIT relies as it confines the use of certificates to the predicate *VerifySig*. A more explicit treatment is unnecessary in this case.

$$\mathsf{I}\left[\begin{array}{l}\forall P : \mathsf{principal}\ \ \forall pk : \mathsf{pubK}\ P \\ \forall sk : \mathsf{secK}\ pk\ \ \forall m : \mathsf{msg} \\ I(m),\ I(sk) \overset{SIG}{\Longrightarrow} I([m]_{sk})\end{array}\right] \qquad \mathsf{I}\left[\begin{array}{l}\forall sk : \mathsf{someSecK}\ \ \forall m : \mathsf{msg} \\ I([m]_{sk}) \overset{PEEK}{\Longrightarrow} I(m)\end{array}\right]$$

Because secret keys are typed like long-term keys, we add specific rules allowing duplication and deletion in memory of known secret keys (paralleling the rules *DPD* and *DLD* from [29]). These are straightforward, and we omit the specific rules here.

The remaining rules implementing the other traditional Dolev-Yao intruder capabilities are defined similarly. A complete list for basic Kerberos can be found in [29].

3.4 Symbolic Analysis of PKINIT

Our formal proofs rely on a double induction aimed at separating the confidentiality and authentication aspects of the analysis of Kerberos 5. Confidentiality and authentication can interact in complex ways, requiring both types of functions in a single proof. (This is not so much the case in the AS exchange, because the security of this first exchange does not rely on properties of earlier rounds, but it is seen clearly in the later rounds as illustrated in [28, 29].) Our proofs are supported by two classes of functions, rank and corank, whose definitions we recall in Section 3.4.1. We prove the correctness of the fixed protocol in Section 3.4.2, discuss other authentication properties in Sections 3.4.3 and 3.4.4, and then state and prove auxiliary lemmas in Section 3.4.5.

3.4.1 Rank and Corank

We start by reviewing the definition of rank and corank functions as originally given in [28, 30, 29] and extended to include the cryptographic primitives that PKINIT adds to basic Kerberos. We start by defining these classes of functions inductively on terms and then extend the definitions to facts.

Rank

Rank captures the amount of work done using a specific key to encrypt a specified message. For a cryptographic key k and a fixed message m_0, we define the *k-rank of a message m relative to* m_0 as shown in Figure 3.8.

If m is an atomic term, then no work has been done using k, and we set the rank to 0. If m is exactly $\{m_0\}_k$, $\{\!|m_0|\!\}_k$, or $[m_0]_k$, then we set the rank to 1. Encrypting or signing a message m of positive k-rank will increase the rank by 1 in case one uses the key k, whereas using a key $k' \neq k$ will have no effect on the k-rank of m relative to m_0. The rank of the concatenation of two messages is set to be the larger of the ranks of the constituent messages.

The extension of the rank function to facts is straightforward. For a key k and for m_0 of type msg, and terms t, t_i, and P any predicate in the protocol signature, we define the k-rank of a fact $P(t_1, ..., t_j)$ relative to m_0 by $\rho_k(P(t_1, ..., t_j); m_0) = \max_{1 \leq i \leq j} \rho_k(t_i; m_0)$. *E.g.*, for network facts the k-rank relative to m_0 is $\rho_k(N(m); m_0) = \rho_k(m; m_0)$.

Our rank functions are most closely connected to data origin authentication, although they are used in conjunction with corank functions in many proofs [30]. The relationship between rank and authentication follows Theorem 3.4.1, which was outlined in [30].

Theorem 3.4.1. *If* $\rho_k(F; m_0) = 0$ *for every fact* F *in the initial state of a generic trace* $\mathcal{T}$ *and if no intruder rule can increase the k-rank relative to* m_0 *then the*

existence of a fact F with $\rho_k(F; m_0) > 0$ in some non-initial state of $\mathcal{T}$ implies that some honest principal fired a rule which produced a fact built up from $\{m_0\}_k$.

As in [29] (although stated there without some of the primitives needed to model PKINIT), if an intruder rule increases k-rank relative to m_0, then the left-hand side of that rule contains $I(k)$. Thus, if we show that $I(k)$ never appears in a trace, we may invoke Theorem 3.4.1 to help prove authentication.

Corank

Corank captures the minimum effort needed, using keys from a specified set, to extract a (possibly) secret message from a given term. For a set E of keys, none of which is a public key, and a fixed atomic message m_0, we define the *E-corank of message m relative to m_0* as shown in Figure 3.9. If m is atomic and $m = m_0$, then no work using keys from E is needed to obtain m_0, and we set the corank to 0. If m is atomic and $m \neq m_0$, then no amount of such work can extract m_0. The number of decryptions needed to obtain m_0 from $\{m\}_k$ using keys from E is 1 more than or the same as the number of decryptions needed to obtain m_0 from m, depending on whether k (if the encryption is symmetric) or some $k' : \mathsf{secK}\, k$ (if the encryption is asymmetric) is an element of E or not. The corank of the concatenation of two messages is equal to the smaller of the coranks of the constituent messages.

The extension of the corank function to facts is slightly different from that of the rank function. The definition of corank of a fact depends on the predicate P in which

it occurs. For example, for the network predicate $\mathsf{N}()$ we define $\hat{\rho}_E(\mathsf{N}(m); m_0) = \hat{\rho}_E(m; m_0)$, while for the client's memory predicate Auth_C (used in the rules in Figures 3.7, 1, and 3) we define $\hat{\rho}_E(\mathsf{Auth}_C(t_1, t_2, t_3); m_0) = \hat{\rho}_E(t_1; m_0)$. This follows our informal rule [28, 30] that the corank of a fact depends only on the arguments of the predicate that might be put onto the network.

Our corank functions are closely connected to confidentiality, which we typically prove by invoking the following theorem from [30].

Theorem 3.4.2. *If $\hat{\rho}_E(F; m_0) > 0$ for every fact in the initial state of a generic trace $\mathcal{T}$, if no intruder can decrease the E-corank relative to m_0, and if no honest principal creates a fact F with $\hat{\rho}_E(F; m_0) = 0$, then m_0 is secret throughout $\mathcal{T}$.*

3.4.2 Correctness of the Fix

We now present the theorem that establishes the correctness of the abstract fix to PKINIT introduced in Section 3.2.1. This, in turn, implies the correctness of PKINIT-27 and subsequent versions of PKINIT (including the final specification)—*i.e.*, that Property 1 holds—because these use a special case of the abstract fix. In the following we will assume that $F(C, n_i) = F(C', n_i')$ implies $C = C'$ and $n_i = n_i'$, for any $C, C' : \mathsf{client}$ and $n_i, n_i' : \mathsf{nonce}$ (for $i = 1, 2$).

Theorem 3.4.3. ***If***

(1) the fact $\mathsf{N}(\{\!|Cert_K, [k, F(C, n_i)]_{sk_K}|\!\}_{pk_C}, C, X, \{AK, n_1, t_K, T\}_k)$ appears in a

trace, for some C : client, K : KAS, k : shK C K, sk_K : someSecK, X : msg, $Cert_K$: CertList, pk_C : pubK C, T : TGS, AK : shK C T, n_i, n_1 : nonce, *and* t_K : time*;*

(2) the fact $VerifySig([k, F(C, n_i)]_{sk_K}; (k, F(C, n_i)); K, Cert_K)$ *holds; and*

(3) for every pk_K : pubK K *and* sk : secK pk_K*, the fact* $I(sk)$ *does not appear in the trace and no fact in the initial state of the trace contained a fact of positive sk-rank relative to* $(k, F(C, n_i))$*,*

then

the KAS K *fired rule* $\iota_{2.1}$*, consuming the fact* $\mathsf{N}(Cert_C, [t_C, n_2]_{sk_C}, C, T, n_1)$ *and creating the fact* $\mathsf{N}(\{\!|Cert_K, [k, F(C, n_i)]_{sk_K}|\!\}_{pk_C}, C, \{AK, C, t_K\}_{k_T}, \{AK, n_1, t_K, T\}_k)$*, for some* $Cert_C$: CertList, sk_C : secK pk_C, n_2 : nonce, t_C, t_K : time, k_T : dbK T.

Proof. Because $VerifySig([k, F(C, n_i)]_{sk_K}; (k, F(C, n_i)); K, Cert_K)$ holds, by Lemma 3.4.7 there is some sk : secK pk_K such that $\rho_{sk}([k, F(C, n_i)]_{sk_K}; (k, F(C, n_i))) > 0$ (where pk_K : pubK K). Thus the fact $\mathsf{N}(\{\!| Cert_C, [k, F(C, n_i)]_{sk_K}|\!\}_{pk_C}, C, X, \{AK, n_1, t_K, T\}_k)$ has positive sk-rank relative to $(k, F(C, n_i))$; by hypothesis, no such fact existed in the initial state of the trace, so some rule firing during the trace must have increased this rank.

By hypothesis, $I(sk)$ does not appear in the initial state of the trace, so by Lemma 3.4.11 $I(sk)$ does not appear in any state of the trace. By Lemma 3.4.8,

this means that no rule fired by the intruder can increase sk-rank relative to $(k, F(C, n_i))$; thus, by Theorem 3.4.1, at some point in this trace an honest principal must have fired a rule that increased this rank. By inspection of the principal rules, the only one that can do this is rule $\iota_{2.1}$; in order for this rule to do so, it must be fired by the KAS K who owns sk, consume a network fact $\mathsf{N}(Cert_{C'}, [t, n_2']_{sk_{C'}}, C', T', n_1')$ for some t : time, n_2', n_1' : nonce, $Cert_{C'}$: CertList, C' : client, and T' : TGS, and produce the fact $\mathsf{N}(\{\!|Cert_K,\ [k, F(C', n_i')]_{sk}|\!\}_{pk_{C'}}$, C', $\{AK', C', t_K'\}_{k_{T'}}, \{AK', n_1', t_K', T'\}_k)$ for some $Cert_K$: CertList, sk : someSecK, $pk_{C'}$: pubK C', AK' : shK C' T', t_K' : time, $k_{T'}$: dbK T', n_1' : nonce, and $F(C', n_i') = F(C, n_i)$. By assumption, $(C', n_i') = (C, n_i)$, which implies that the request that K processed must match the request described in the hypotheses. □

The following corollary specializes the result in Theorem 3.4.3 to the particular fix used in PKINIT-27 and to the client's receipt of the network message described in the hypotheses of this theorem. It is the formal statement of Property 1, which says that if C processes a reply message (containing the signed checksum of a request that C previously sent), then some KAS K must have sent a reply message intended for C.

Corollary 3.4.4. ***If***

(1) some C : client *fires rule* $\iota_{1.2}$*, consuming the fact* $\mathsf{N}(\{\!|Cert_K, [k, ck]_{sk_K}|\!\}_{pk_C}$, $C, X, \{AK, n_1, t_K, T\}_k)$ *and producing the fact* $Auth_C(X, T, AK)$ *for some* K : KAS, k : shK C K, sk_K : someSecK, ck, X : msg, $Cert_K$: CertList,

pk_C : pubK C, T : TGS, AK : shK C T, n_1 : nonce, t_K : time, *and*

(2) $ck = H_k(Cert_C, [t_C, n_2]_{sk_C}, C, T, n_1)$, *for some* t_C : time, n_2 : nonce, $Cert_C$: CertList, sk_C : SecK pk_C, *and*

(3) for every pk_K : pubK K *and* sk : secK pk_K, *the fact* $I(sk)$ *does not appear in the trace and no fact in the initial state of the trace contained a fact of positive sk-rank relative to* (k, ck),

then

the KAS K *fired rule* $\iota_{2.1}$, *consuming the fact* $\mathsf{N}(Cert_C, [t_C, n_2]_{sk_C}, C, T, n_1)$ *and creating the fact* $\mathsf{N}(\{\!|Cert_K, [k, ck]_{sk_K}|\!\}_{pk_C}, C, \{AK, C, t_K\}_{k_T}, \{AK, n_1, t_K, T\}_k)$, *for some* k_T : dbK T, t_K : time.

Proof. This follows by letting $F(C, n_i) = ck = H_k(Cert_C, [t_C, n_2]_{sk_C}, C, T, n_1)$ in Theorem 3.4.3; this construction satisfies the assumptions on $F(C, n_i)$ for both $i = 1$ and $i = 2$. C's rule firing implies that the first two hypotheses of the theorem hold; the third hypothesis of the corollary specializes the third hypothesis of the theorem to the case of $ck = F(C, n_i)$, which implies the conclusion for some n_1', n_2' : nonce and $Cert_C'$: CertList. C's firing of rule $\iota_{1.2}$ (when specialized to the checksum) implies that the $n_1' = n_1$, $n_2' = n_2$, and $Cert_C' = Cert_C$. □

As a further remark, if in the abstract fix of PKINIT from Section 3.2.1 one chooses $F(C, n_i) = F(C, n_1)$, the proof of Theorem 3.4.3 shows that authentication

of the KAS to the client still holds. In fact, the KAS does not return any information containing n_2. This means that the following holds:

Property 2. *The signed nonce n_2 in the client's AS request is superfluous for the purpose of authentication in PKINIT.*

This property does not imply that n_2 can simply be omitted from the first message of PKINIT in general, as some signed session identifiers is necessary to correctly support authentication as in Property 3 below. Rather, it suggests that it could be simplified by replacing every occurrence of n_2 with n_1.

3.4.3 Authentication of C to K

While our primary focus here is on the authentication of K to C, because C signs a nonce in her request we may also prove authentication of C to K; note that this holds in both the broken and fixed versions of PKINIT, as the fix does not affect C's request or our reasoning about it. Informally, we state this as the following property

Property 3. *If a KAS processes a PKINIT request from a client C, then previously C sent a PKINIT request that contained the signed data in the message received and processed by the KAS.*

Formally, we state this property as the following theorem.

Theorem 3.4.5. ***If***

(1) some $K : \mathsf{KAS}$ *fires rule* $\iota_{2.1}$*, consuming the fact* $\mathsf{N}(Cert_C, [t_C, n_2]_{sk}, C, T, n_1)$ *for some* $Cert_C : \mathsf{CertList}$*,* $t_C : \mathsf{time}$*,* $n_1, n_2 : \mathsf{nonce}$*,* $sk : \mathsf{someSecK}$*,* $C : \mathsf{client}$*, and* $T : \mathsf{TGS}$*; and*

(2) for every $pk : \mathsf{pubK}\, C$ *and* $sk : \mathsf{secK}\, pk$*, the fact* $I(sk)$ *does not appear in the trace and no fact in the initial state of the trace contained a fact of positive* sk*-rank relative to* (t_C, n_2)*,*

then

at some point earlier in the trace, the client C *fired rule* $\iota_{1.1}$*, generating the fact* $\mathsf{N}(t_C, n_2, Cert'_C, [t_C, n_2]_{sk_C}, C, T', n'_1)$ *for some* $Cert'_C : \mathsf{CertList}$*,* $T' : \mathsf{TGS}$*, and* $n'_1 : \mathsf{nonce}$*.*

Proof. Having assumed that K fired rule $\iota_{2.1}$, the fact $\mathit{VerifySig}([t_C, n_2]_{sk}; (t_C, n_2); C, Cert_C)$ must hold. By Lemma 3.4.7, the fact consumed by this rule firing has positive sk-rank relative to (t_C, n_2). By hypothesis, no such fact appeared in the initial state of the trace, so some rule that was fired during the trace increased this rank. Also by hypothesis, $I(sk)$ cannot appear in the trace, so by Lemma 3.4.8, the intruder cannot increase this rank. By inspection of the principal rules, the only rule that can increase this rank is rule $\iota_{1.1}$ when fired by C to create the fact $\mathsf{N}(t_C, n_2, Cert'_C, [t_C, n_2]_{sk_C}, C, T', n'_1)$ for some $Cert'_C : \mathsf{CertList}$, $T' : \mathsf{TGS}$, and $n'_1 : \mathsf{nonce}$. □

3.4.4 Authentication in the Pre-Fix Protocol

Our attack in Section 3.1 showed that an intruder could impersonate a KAS to a client. This means that KAS-to-client authentication does not hold when PKINIT-26 is used, in the sense that a client might receive a reply containing her name that appears to come from some KAS but without any KAS having ever sent such a message. However, PKINIT-26 does provide some authentication with respect to the later exchanges as shown by Property 4 below, and this property still holds in the fixed protocol (PKINIT-27 and later versions).

Property 4. *If TGS T processes a valid request message naming a client C, and the key used to encrypt the TGT which was contained in that request message was secret when the system started, then earlier the KAS K generated a TGT naming C. Furthermore, if C's private key was initially secret (so that C is honest), then C sent a request to T after K sent a corresponding AS reply to C.*

As a consequence, we see that even if the intruder is carrying out the attack by letting C login to T and S as the intruder (so that the intruder can watch the traffic between them), some KAS must have created a TGT for the intruder, formalizing the requirement that the intruder be a legal user of the system. Furthermore, this shows that while C might successfully 'request' a service ticket as the intruder in the attack scenario, it requires the participation of the intruder; in particular, C could not obtain a TGT that names an honest client C' if the relevant keys are secret.

Theorem 3.4.6 formalizes this property; the relevant MSR roles for the TG and CS exchanges can be found in Appendix .1.

Theorem 3.4.6. ***If***

(1) T : TGS *fires rule* $\iota_{4.1}$ *consuming the fact* $\mathsf{N}(\{AK, C, t_K\}_{k_T}, \{AK, n_3, t_K, T\}_k, C, S)$ *and creating the fact* $\mathsf{N}(C, \{SK, C\}_{k_S}, \{SK, n_3, S\}_{AK})$ *for some* AK : shK $C\ T$, C : client, K : KAS, t_K : time, k_T : dbK T, n_3 : nonce, S : server, *and*

(2) the fact $I(k_T)$ *does not appear in the trace and no fact in the initial state of the trace contained a fact of positive* k_T*-rank relative to* $\{AK, C, t_K\}$, *and*

(3) the fact $I(sk_C)$ *is not inferable in the initial state of the trace for every* sk_C : SecK pk_C,

then

(i) the KAS K *earlier fired rule* $\iota_{2.1}$, *creating a fact* $\mathsf{N}(\{\!|Cert_K, [k, H_k(Cert_C, [t_C, n_2]_{skC}, C, T, n_1)]_{sk_K}|\!\}_{pk_C}, C, \{AK, C, t_K\}_{k_T}, \{AK, n_1, t_K, T\}_k)$ *for some* $Cert_K$, $Cert_C$: CertList, n_1, n_2 : nonce, *and*

(ii) C *fired rule* $\iota_{3.1}$, *creating the fact* $\mathsf{N}(\{AK, C, t_K\}_{k_T}, \{AK, n_3, t_K, T\}_k, C, S)$ *in a state later in the trace than the state at which* K *fired rule* $\iota_{2.1}$ *producing the fact described above.*

Proof. We start with consideration of k_T-rank relative to (AK, C, t_K). T's hypothesized rule firing consumed a fact for which this rank is positive, but by assumption this rank was 0 for every fact in the initial state of the trace. Thus some rule firing in the trace increased this rank.

Because $I(k_T)$ does not appear in the initial state of the trace, by Lemma 3.4.10 this fact never appears in the trace. Thus, by Lemma 3.4.8, the intruder cannot have increased k_T-rank relative to (AK, C, t_K), so some honest principal must have done this. Inspection of the principal rules shows that if k has type $\mathsf{dbK}\ T$ for some $T : \mathsf{TGS}$, then the only honest principals that increase k-rank relative to any message are those of type KAS through the firing of rule $\iota_{2.1}$. Thus some $K : \mathsf{KAS}$ fired rule $\iota_{2.1}$, which must have consumed and produced the facts claimed.

We now show that AK is secret by considering $\{k, k_T\}$-corank relative to AK. K's firing of rule $\iota_{2.1}$ freshly generates AK, so all previous states had infinite $\{k, k_T\}$-corank relative to AK, and the state that results from this rule firing has $\{k, k_T\}$-corank equal to 1 relative to AK. As noted above, $I(k_T)$ never appears in the trace; we must show that $I(k)$ does not either. K's firing of rule $\iota_{2.1}$ also freshly generated k, so all previous states had infinite E-corank relative to k for every set E; furthermore, the resulting state has $\{sk_C\}$-corank equal to 1 relative to k. Because $I(sk_C)$ was not inferable from the initial state of the trace, by Lemma 3.4.11 this fact never appears in the trace. Thus, by Lemma 3.4.9, the intruder cannot decrease $\{sk_C\}$-corank relative to k. Inspection of the principal rules show that no honest

principal will do this either; note that while C decrypts the message containing k, she does not store this key, much less in a predicate that will allow it to be put onto the network. Therefore no state in the trace has $\{sk_C\}$-corank equal to 0 relative to k, so $I(k)$ never appears in the trace.

Because neither $I(k_T)$ nor $I(k)$ ever appear in the trace, the intruder cannot decrease $\{k, k_T\}$-corank relative to AK. By inspection, we see that the only principal rule which decreases this corank is rule $\iota_{2.1}$ when it is fired to freshly generate AK. As noted above, the resulting state must have $\{k, k_T\}$-corank equal to 1 relative to AK; no principal rule will decrease this corank to 0. Thus no fact of $\{k, k_T\}$-corank equal to 0 relative to AK, and in particular the fact $I(AK)$, appears in the trace.

We may now show that C fired rule $\iota_{3.1}$ in the claimed fashion. T's hypothesized rule firing also consumed a fact of AK-rank equal to 1 relative to C, t_C. Because AK was freshly generated during the trace (as shown above) and $I(AK)$ never appears in the trace, some honest principal must have fired a rule that increased AK-rank relative to C, t_C. By inspection, we see that the only such principal rule is rule $\iota_{3.1}$, which must produce the fact $\mathsf{N}(X, \{C, t_C\}_{AK}, C, S, n_3)$ for some $X : \mathsf{msg}$, $t_C : \mathsf{time}$, $S : \mathsf{server}$, and $n_3 : \mathsf{nonce}$. Because this fact has positive AK-rank relative to C, t_C and AK was freshly generated by K, we know that C's firing of rule $\iota_{3.1}$ occurred after K's firing of rule $\iota_{2.1}$. □

3.4.5 Auxiliary Lemmas

A number of lemmas giving conditions under which various ranks and coranks can be increased and decreased by rule firings can be found in [29]. However, we need to add to them as public-key operations where not considered in that work.

Lemma 3.4.7. *If* $VerifySig(s; m; P, Certs)$ *holds, then for some* $pk : \mathsf{pubK}\ P$ *and* $sk : \mathsf{secK}\ pk$*, the sk-rank of* s *relative to* m *is positive.*

Proof. Our assumptions about $VerifySig$ imply that s is a valid signature of m under one of P's secret keys, *i.e.*, for some $pk : \mathsf{pubK}\ P$ and $sk : \mathsf{secK}\ pk$, $s = [m]_{sk}$. Thus s has positive sk-rank relative to m. □

The following lemmas were proved in [29] for the formalizations of basic Kerberos considered there. These property still holds here once we add asymmetric encryption and digital signatures.

Lemma 3.4.8. *If an intruder rule* R *can increase* k*-rank relative to a message* m_0*, then the left-hand side of* R *contains* $I(k)$*.*

Proof. By inspection of the intruder rules. □

Lemma 3.4.9. *If* m_0 *is not a principal name, time, or key of one of the types* $\mathsf{dbK}\ I$*,* $\mathsf{shK}\ I\ A$ *(for* $A : \mathsf{TGS}$ *or* $A : \mathsf{server}$*),* $\mathsf{shK}\ C\ I$ *for* $C : \mathsf{client}$*, or* $\mathsf{pubK}\ P$ *for* $P : \mathsf{principal}$*, then any intruder rule that decreases* E*-corank relative to* m_0 *either contains* $I(k)$ *in its left hand side for some* $k \in E$ *or freshly generates* m_0*.*

Proof. By inspection of the intruder rules. □

The following lemma is an analog of Lemma 6 in [29]; the additional intruder rules do not change it.

Lemma 3.4.10. *For any* P : principal, $P \neq I$, *and* k : dbK P, *if* $I(k)$ *does not appear in the initial state of the trace, then* $I(k)$ *does not appear in any state of the trace.*

Proof. By inspection of the intruder rules. Because dbK P is not a subtype of msg, if $P \neq I$ then $I(k)$ appears on the right-hand side of a rule only if it also appears on the left-hand side. □

The next lemma is analogous to the previous one, but it refers to secret keys for asymmetric encryption instead of the keys in the KAS's database.

Lemma 3.4.11. *For any* P : principal, *pubk* : pubK P, $P \neq I$, *and* k : secK *pubk, if* $I(k)$ *does not appear in the initial state of the trace, then* $I(k)$ *does not appear in any state of the trace.*

Proof. By inspection of the intruder rules. Because secK *pubk* is not a subtype of msg, for k : secK *pubk*, the only time $I(k)$ may appear on the right-hand side of an intruder rule but not on the left-hand side is in rule $_{SK'}$, in which case $P = I$. □

$$\rho_k(m;m_0) = \begin{cases} 0, & m \text{ is an atomic term} \\ 0, & m = \{m_1\}_k, \rho_k(m_1;m_0) = 0, m_1 \neq m_0 \\ 1, & m = \{m_0\}_k \\ \rho_k(m_1;m_0), & m = \{m_1\}_{k'}, k' \neq k \\ \rho_k(m_1;m_0)+1, & m = \{m_1\}_k, \rho_k(m_1;m_0) > 0 \\ 0, & m = \{\!\{m_1\}\!\}_k, \rho_k(m_1;m_0) = 0, m_1 \neq m_0 \\ 1, & m = \{\!\{m_0\}\!\}_k \\ \rho_k(m_1;m_0), & m = \{\!\{m_1\}\!\}_{k'}, k' \neq k \\ \rho_k(m_1;m_0)+1, & m = \{\!\{m_1\}\!\}_k, \rho_k(m_1;m_0) > 0 \\ 0, & m = [m_1]_k, \rho_k(m_1;m_0) = 0, m_1 \neq m_0 \\ 1, & m = [m_0]_k \\ \rho_k(m_1;m_0), & m = [m_1]_{k'}, k' \neq k \\ \rho_k(m_1;m_0)+1, & m = [m_1]_k, \rho_k(m_1;m_0) > 0 \\ \max\{\rho_k(m_1;m_0), & m = m_1, m_2 \\ \qquad \rho_k(m_2;m_0)\}, & \end{cases}$$

Figure 3.8: The definition of $\rho_k(m;m_0)$, the k-rank of m relative to m_0.

$$\hat{\rho}_E(m;m_0) = \begin{cases} \infty, & m \text{ is atomic, } m \neq m_0 \\ 0, & m \text{ is atomic, } m = m_0 \\ \hat{\rho}_E(m_1;m_0)+1, & m = \{m_1\}_k,\ k \in E \\ \hat{\rho}_E(m_1;m_0), & m = \{m_1\}_k,\ k \notin E \\ \hat{\rho}_E(m_1;m_0)+1, & m = \{\!\{m_1\}\!\}_k, \exists k' : \mathsf{secK}\ k, \quad k' \in E \\ \hat{\rho}_E(m_1;m_0), & m = \{\!\{m_1\}\!\}_k, \forall k' : \mathsf{secK}\ k, \quad k' \notin E \\ \hat{\rho}_E(m_1;m_0), & m = [m_1]_k \\ \min\{\hat{\rho}_E(m_1;m_0), & m = m_1, m_2, \quad m_1, m_2 \neq \emptyset \\ \qquad \hat{\rho}_E(m_2;m_0)\}, & \end{cases}$$

Figure 3.9: Definition of $\hat{\rho}_E(m;m_0)$, the E-corank of m relative to m_0.

Chapter 4

Formal Proofs in the ideal Backes, Pfitzmann and Waidner Model

In this chapter we will present a formalization and an analysis of basic and public-key Kerberos in the Dolev-Yao style BPW Model. We will here merely consider results in the symbolic model, and therefore leave out soundness properties of the results with respect to the computational model which one may infer from the proofs using the symbolic BPW model; these lie outside the scope of this work. In Section 4.1, we recall the BPW model (e.g., [9, 11, 14, 15]), and subsequently apply it to the specification of Kerberos 5 and Public-key Kerberos (i.e., Kerberos with PKINIT).

4.1 The symbolic BPW Model

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

The BPW model, introduced in [15], offers a deterministic Dolev-Yao style formalism of cryptographic protocols with commands for a vast range of cryptographic operations such as public-key, symmetric 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. 4.1). In this reactive scenario, semantics is based on state, i.e., of 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

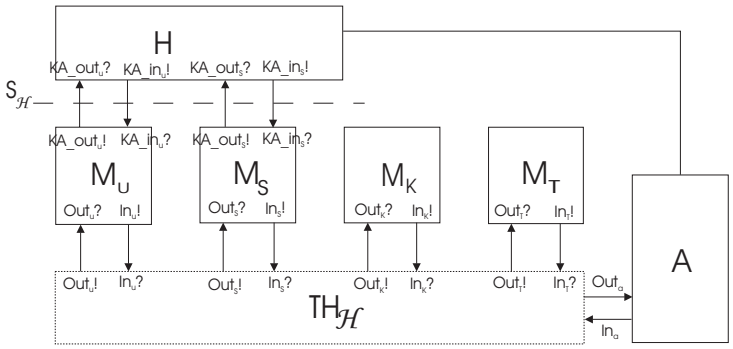

Figure 4.1: Overview of the Kerberos symbolic system

m with key k, then it makes a new abstract database entry for the ciphertext 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 ciphertext 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 ciphertext. 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 [71] 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. 4.1). 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*. To guarantee that the system is polynomially bounded in the security parameter, the BPW model maintains length functions on the entries of the abstract database. 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.

4.1.1 Kerberos in the BPW Model

We now model the Kerberos protocol in the framework of [15] using the BPW model. We write ":=" for deterministic assignment, "=" for testing for equality and "←" 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∈{PK, K5} then, as described in Section 4.1, 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[1]', 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 $\mathrm{KA\_out}_u!$ and $\mathrm{KA\_in}_u?$, respectively ("KA" for"Key sharing and Authentication"). The ports for the server machines are labeled similarly (see Fig. 4.1).

The behaviors of the protocol machines is described in Algorithm 1 to 5 (Fig. 4.2–4.7 and 5– 22). In the following, we comment on two algorithms of PKINIT (Fig. 4.2, 4.3 and Fig. 4.4–4.7) , the rest is displayed in Appendix .2. If, for instance, a protocol machine $\mathrm{M}_u^{\mathrm{PK}}$ receives a message ($\mathsf{new\_prot}$, PK, K, T) at $\mathrm{KA\_in}_u?$ then it will execute Algorithm 1A (Fig. 4.2) 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 $S \in$

[1]I.e., administrative domain; we do not consider cross-realm authentication here, although it has been analyzed symbolically in [39]

$\{S_1, \ldots, S_l\}$ 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 $S_{\mathcal{H}} := \{\mathrm{KA\_out}_u!, \mathrm{KA\_in}_u? \mid u \in \mathcal{H} \setminus \{K, T\}\}$. The symbolic system is defined as the set $Sys^{\mathrm{Kerb,\ symb}} := \{(\hat{M}_{\mathcal{H}}, S_{\mathcal{H}})\}$. Note that, since 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 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 $x.type \in typeset = \{$auth, 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 [10] 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 Section 4.1).

Initially, D is empty. $\mathrm{TH}_{\mathcal{H}}$ 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: in contrast to the the analysis 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, we are formalizing here 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. Then one should 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.

Example of Algorithms We are only going to examine PKINIT (Fig. 2.3) and explain the steps of its Algorithms 1A and 2 (Fig. 4.2 and Fig. 4.4–4.7) which are more complex than the algorithms in Kerberos 5. However, with these explanations the remaining algorithms (Appendix .2) should be easily understandable. For details on the definition of the used commands see [11, 14, 15]. For readability of the figures, we noted on the right (in curly brackets) to which terms in the more commonly used Dolev-Yao notation the terms in the algorithms correspond ($\approx$).

Protocol start of PKINIT. In order to start a new PKINIT protocol, user u inputs (new_prot, PK, K, T) at port $\mathrm{KA\_in}_u?$. Upon such an input, $\mathrm{M}_u^{\mathrm{PK}}$ runs Algorithm 1A (Fig. 4.2) 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 gen_nonce(). In step 1A.3 the command list(_, _) concatenates t_u and $n_{u,2}$ into a new list that is signed in step 1A.4 with u's private key. Since we are working in an asynchronous system, the timestamp t_u is approximated by some arbitrary message (e.g. by a nonce). The command 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 $\mathrm{out}_K?$ with $v \in \{1, .., n\}$, the machine $\mathrm{M}_K^{\mathrm{PK}}$ runs Algorithm 2 (Fig. 4.4–4.7) which first checks if the message m is a valid AS_REQ and then prepares and sends the corresponding AS_REP, *i.e.*, the second message in the AS exchange. In order to verify that the input is a possible AS_REQ, the types of the input message m's components are checked in steps 2.1–2.5. The command 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. 2.3). In the specifications of Kerberos, a key derivation function is used to create those two keys from the key k.

4.2 Formal Results

First we will summarize the security properties that we prove at the symbolic level for both basic Kerberos and Kerberos with PKINIT. Then We begin by formalizing the respective security properties and verify them properties in the BPW model in Section 4.2.1. Similar properties in symbolic terms have been proved using a formalization in MSR for basic Kerberos [28, 30] and for the AS exchange when

PKINIT is used in the section 3.4. The first property we prove here concerns the secrecy of keys, a notion that is captured formally as Definition 4.2.1 in Section 4.2.1. This property may be summarized as follows.

Property 5 (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 4.2.2 in Section 4.2. This property may be summarized as follows.

Property 6 (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 4.2.3 below shows that these properties hold for our symbolic formalizations of basic and public-key Kerberos in the BPW model.

4.2.1 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 first 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, since we do not consider the purpose of timestamps in Kerberos. Timestamps are currently not modeled in the BPW model.

Secrecy and Authentication Requirements We now define the notion of key secrecy, which was informally captured already in Property 5 of Section 4.2, as the following formal requirement in the language of the BPW model.

Definition 4.2.1 (Key secrecy requirement)**.** For Kerb $\in$\{PK, K5\} the secrecy requirement $Req^{\mathsf{Sec}}_{Kerb}$is:

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

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

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 Thm. 4.2.3 will show that the symbolic Kerberos systems specified in Section 4.1.1 satisfy this notion of secrecy, and therefore Kerberos enjoys Property 5.

Next we define the notion of authentication in Property 6 in the language of the BPW model.

Definition 4.2.2 (Authentication requirements)**.** For $Kerb \in \{PK, K5\}$:

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

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

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

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

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

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

Theorem 4.2.3 will show that this notion of authentication is satisfied by the symbolic Kerberos system. Therefore Kerberos has Property 6.

When proving that Kerberos has these properties, we will use the notion of a system *Sys perfectly fulfilling a requirement Req*, $Sys \models^{\mathrm{perf}} Req$. This means the

property *Req* holds with probability one over the probability space of runs for a fixed security parameter (as defined in Section 4.1). Later we will also need the notion of a system *Sys computationally fulfilling a requirement Req*, $Sys \models^{\text{poly}} Req$, i.e., 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 Thm. 4.2.3, we first need to prove a number of auxiliary properties (previously called *invariants* in, e.g., [7, 12]). 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 Thm. 4.2.3, which formalizes Properties 5 and 6, make authentication and confidentiality statements for the first two rounds of Kerberos. These properties are described in English below; they are formalized and proved in Appendix .3.

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 5 and 6 hold symbolically for Kerberos. We show a proof excerpt in the case of Public-key Kerberos (the outline is analogous for Kerberos 5).

Theorem 4.2.3. *(Security of the Kerberos Protocol based on the BPW Model)*

- *Let* $Sys^{\mathrm{K5,\,symb}}$ *be the symbolic Kerberos 5 system defined in Section 4.1.1, and let* Req_{K5}^{Sec} *and* Req_{K5}^{Auth} *be the secrecy and authentication requirements defined above. Then* $Sys^{\mathrm{K5,\,symb}} \models^{\mathrm{perf}} Req_{K5}^{\mathsf{Sec}} \wedge Req_{K5}^{\mathsf{Auth}}$.

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

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 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. Similar holds for the case that S successfully terminates a session run. This shows the key secrecy property Req_{PK}^{Sec}. As for the authentication property $Req_{PK}^{\mathsf{Auth1}}$, 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 $Req_{PK}^{\mathsf{Auth2}}$, 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 KA_out$_S$!. □

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, ciphertexts, 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 (corresponding to the cryptographic assumptions we make in section 5.4.1, with small additions like type tagging and additional randomization). In [11, 13, 14, 15] it was established that the cryptographic implementation of the BPW model is *at least as secure as* the BPW model, meaning that whatever an active adversary can do in the implementation can also be achieved by another adversary in the BPW model, or the underlying cryptography can be broken.

The proof of Theorem 4.2.3 shares similarities with the Dolev-Yao style proofs of analogous results attained for Kerberos 5 and PKINIT using the MSR framework ([28, 30] and section 3.4). 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). In future work, we plan to draw a formal comparison between these two Dolev-Yao encodings of a protocol, and the proof techniques they support.

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

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. $\mathsf{send\_i}(K, m_1^{hnd})$

Figure 4.2: Algorithm 1 of Public-key Kerberos, part 1: Evaluation of inputs from the user (starting the AS exchange).

B) Input:$(\mathsf{continue\_prot}, \mathrm{PK}, T, S, AK^{hnd})$ at $\mathrm{KS\_in}_u$? 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. $\mathsf{send\_i}(T, m_3^{hnd})$

Figure 4.3: Algorithm 1 of Public-key Kerberos, part 2: Evaluation of inputs from the user (starting the TG exchange).

Input:(v, K, i, m^{hnd}) at 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 $i = 1, 2, 5$ $\quad \{x_1 \approx Cert_C, x_2 \approx [t_C, n_2]_{sk_C}, x_5 \approx n_1\}$
3. $x_i \leftarrow \mathsf{retrieve}(x_i^{hnd})$ for $i = 3, 4$ $\quad \{x_3 \approx C, x_4 \approx T\}$
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. $w_1 \leftarrow \mathsf{retrieve}(w_1^{hnd})$

Figure 4.4: Algorithm 2 of Public-key Kerberos, part 1: Behavior of the KAS

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

Figure 4.5: Algorithm 2 of Public-key Kerberos, part 2: Behavior of the KAS

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. $z_4^{hnd} \leftarrow \mathsf{list}(AK^{hnd}, x_5^{hnd}, t_K^{hnd}, x_4^{hnd})$ $\{z_4 \approx AK, n_1, t_K, T\}$

Figure 4.6: Algorithm 2 of Public-key Kerberos, part 3: Behavior of the KAS

40. $m_{24} \leftarrow \mathsf{sym\_encrypt}(k_e^{hnd}, z_4^{hnd})$ $\qquad m_{24} \approx \{Ak, n_1, t_K, T\}_{k_e}\}$

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. $\mathsf{send\_i}(v, m_2^{hnd})$

Figure 4.7: Algorithm 2 of Public-key Kerberos, part 4: Behavior of the KAS

Chapter 5

Computationally Sound Mechanized Proofs with CryptoVerif

5.1 CryptoVerif Overview

In this section we give a brief overview of CryptoVerif, formalize Kerberos using it, and summarize the authentication and secrecy properties proved by CryptoVerif. The prover CryptoVerif [22, 25, 26], available at `http://www.cryptoverif.ens.fr`, can directly prove security properties of cryptographic protocols in the computational model. Protocols are formalized using a probabilistic polynomial-time process calculus which is inspired by the pi-calculus and the calculi introduced in [60]

and [67]. In this calculus, messages are bitstrings and cryptographic primitives are functions operating on bitstrings. This calculus is illustrated below on a portion of code coming from Kerberos.

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

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

5.2 Informal Security Properties

We now informally state security properties of Kerberos concerning entity authentication and key secrecy. We note again that entity authentication is a weaker form of authentication than the one that Kerberos intends to offer; we will not check when timestamps are created and therefore credentials do not carry an expiration date. This prevents us from reasoning about recency (whereas in at the end of an actual completed Kerberos run a server will be able to tell when a client sent a request to the server and reject the request if the credentials in it expired). In Section 5.4 we will formalize these security properties in the language of CryptoVerif and present the results in the computational model.

Property 7 (Authentication properties)**.**

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

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

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

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

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

We consider two different notions of key secrecy. One is the standard notion of cryptographic key secrecy, and the other is the notion of key usability [44] (see below in Section 5.4.3). We note that the latter notion had not been considered in [8].

Property 8 (Secrecy properties)**.**

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

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

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

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

5.3 The Probabilistic Polynomial-Time Process Calculus

In the following, we will recall the syntax and informal semantics of the process calculus used by CryptoVerif [22, 25]. This calculus was inspired by the pi calculus and by the calculi of [67] and of [60].

5.3.1 Syntax

The syntax of this calculus is summarized in Figure 5.1. In this calculus, we assume a countable set of channel names, denoted by c. In order to guarantee that all processes run in probabilistic polynomial time in a security parameter η, there exists a mapping maxlen_η from channels to integers, such that $\text{maxlen}_\eta(c)$ is an upper bound on the length of messages sent on channel c. For any channel c, $\text{maxlen}_\eta(c)$ is polynomial in η and longer messages are truncated.

In this calculus, there are also parameters N, which correspond to integer values that are polynomial in the security parameter. We denote the integer value interpretation of N for a given security parameter η by $I_\eta(N)$. The function $I_\eta(N)$ is

$M ::=$	term
i	replication index
$x[M_1, \ldots, M_m]$	variable access
$f(M_1, \ldots, M_m)$	function application
$Q ::=$	input process
0	nil
$Q \mid Q'$	parallel composition
$!^{i \leq N} Q$	replication N times
$\mathsf{newChannel}\ c; Q$	channel restriction
$c[M_1, \ldots, M_l](x_1[\tilde{i}] : T_1, \ldots, x_k[\tilde{i}] : T_k); P$	input
$P ::=$	output process
$\overline{c[M_1, \ldots, M_l]}\langle M'_1, \ldots, M'_k \rangle; Q$	output
$\mathsf{new}\ x[i_1, \ldots, i_m] : T; P$	random number
$\mathsf{let}\ x[i_1, \ldots, i_m] : T = M\ \mathsf{in}\ P$	assignment
$\mathsf{if}\ \mathsf{defined}(M_1, \ldots, M_l) \wedge M\ \mathsf{then}\ P\ \mathsf{else}\ P'$	conditional
$\mathsf{find}\ (\bigoplus_{j=1}^{m} u_{j1}[\tilde{i}] \leq n_{j1}, \ldots, u_{jm_j}[\tilde{i}] \leq n_{jm_j}$ $\mathsf{suchthat}\ \mathsf{defined}(M_{j1}, \ldots, M_{jl_j}) \wedge M_j\ \mathsf{then}\ P_j)$ $\mathsf{else}\ P'$	array lookup
$\mathsf{event}\ e(M_1, \ldots, M_m); P$	event

Figure 5.1: Syntax of the process calculus

assumed to be polynomially bounded and efficiently computable with respect to η.

The process calculus also uses types, denoted by T. For each value of the security parameter η, each type corresponds to a subset $I_\eta(T)$ of $\mathit{Bitstring} \cup \{\bot\}$, where *Bitstring* is the set of all bitstrings and $\bot$ is a special symbol. The set $I_\eta(T)$ is recognizable in polynomial time, *i.e.*, there exists an algorithm that decides whether $x \in I_\eta(T)$ in time polynomial in the length of x and the value of η. A type T is *fixed-length* if $I_\eta(T)$ is the set of all bitstrings of a certain length, which is (bounded by) a polynomial function in the security parameter. Particular types are predefined: *bool*, such that $I_\eta(\mathit{bool}) = \{\mathrm{true}, \mathrm{false}\}$, where false is 0 and true is 1; *bitstring*, such that $I_\eta(\mathit{bitstring}) = \mathit{Bitstring}$; $\mathit{bitstring}_\bot$ such that $I_\eta(\mathit{bitstring}_\bot) = \mathit{Bitstring} \cup \{\bot\}$; and $[1, N]$ where N is a parameter, such that $I_\eta([1, N]) = [1, I_\eta(N)]$.

The calculus represents cryptographic primitives by function symbols f. Each function symbol comes with a type declaration $f : T_1 \times \ldots \times T_m \to T$. For each value of η, each function symbol f corresponds to a function $I_\eta(f)$ from $I_\eta(T_1) \times \ldots \times I_\eta(T_m)$ to $I_\eta(T)$, such that $I_\eta(f)(x_1, \ldots, x_m)$ is computable in polynomial time in the lengths of $x_1, \ldots, x_m$ and the value of η. Particular functions are predefined, and some of them use the infix notation: $M = N$ for the equality test, $M \neq N$ for the inequality test (both taking two values of the same type T and returning a value of type *bool*), $M \vee N$ for the boolean or, $M \wedge N$ for the boolean and, $\neg M$ for the boolean negation (taking and returning values of type *bool*).

In this calculus, terms, which are, *e.g.*, constructed with the help of function

symbols, represent computations on bitstrings. We write $!^{i \leq N} Q$ when N copies of the process Q run in parallel, and we call the integer i, which helps distinguishing different copies, replication index. (Replication indices are typically used as array indices.) The variable access $x[M_1, \ldots, M_m]$ returns the content of the cell of indices $M_1, \ldots, M_m$ of the m-dimensional array variable x. We use x, y, z, u as variable names. The function application $f(M_1, \ldots, M_m)$ returns the result of applying function f to $M_1, \ldots, M_m$.

The calculus distinguishes two kinds of processes: input processes Q are ready to receive a message on a channel; output processes P output a message on a channel after executing some internal computations. The input process 0 does nothing; $Q \mid Q'$ is the parallel composition of Q and Q'; $!^{i \leq N} Q$ represents N copies of Q in parallel, each with a different index $i \in [1, N]$; $\mathsf{newChannel}\ c; Q$ creates a new private channel c and executes Q; the semantics of the input $c[M_1, \ldots, M_l](x_1[\widetilde{i}] : T_1, \ldots, x_k[\widetilde{i}] : T_k); P$ will be explained below together with the semantics of the output.

The output process $\mathsf{new}\ x[i_1, \ldots, i_m] : T; P$ picks a new random number uniformly from $I_\eta(T)$, stores it in $x[i_1, \ldots, i_m]$, and executes P. (The type T must be a fixed-length type, because probabilistic polynomial-time Turing machines can choose random numbers uniformly only in such types.) Function symbols represent deterministic functions, so all random numbers must be chosen by $\mathsf{new}\ x[i_1, \ldots, i_m] : T$. The process $\mathsf{let}\ x[i_1, \ldots, i_m] : T = M\ \mathsf{in}\ P$ stores the bitstring value of M (which must be in $I_\eta(T)$) in $x[i_1, \ldots, i_m]$ and executes P. The pro-

cess event $e(M_1, \ldots, M_m); P$ executes the event $e(M_1, \ldots, M_m)$, then executes P. Executing an event does not change the state of the system, but events are used in the specification of authentication properties.

Next, we explain the process find $(\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] \leq N_{j1}, \ldots, u_{jm_j}[\widetilde{i}] \leq N_{jm_j}$ suchthat defined$(M_{j1}, \ldots, M_{jl_j}) \wedge M_j$ then $P_j)$ else P, where $\widetilde{i}$ denotes a tuple $i_1, \ldots, i_{m'}$. The order and array indices on tuples are taken component-wise, so for instance, $u_{j1}[\widetilde{i}] \leq N_{j1}, \ldots, u_{jm_j}[\widetilde{i}] \leq N_{jm_j}$ can be further abbreviated $\widetilde{u_j}[\widetilde{i}] \leq \widetilde{N_j}$. A simple example is the following: find $u \leq N$ suchthat defined$(x[u]) \wedge x[u] = a$ then P' else P tries to find an index u such that $x[u]$ is defined and $x[u] = a$, and when such a u is found, it executes P' with that value of u; otherwise, it executes P. In other words, this find construct looks for the value a in the array x, and when a is found, it stores in u an index such that $x[u] = a$. Therefore, the find construct allows us to access arrays, which is key for our purpose. More generally, find $u_1[\widetilde{i}] \leq N_1, \ldots, u_m[\widetilde{i}] \leq N_m$ suchthat defined$(M_1, \ldots, M_l) \wedge M$ then P' else P tries to find values of $u_1, \ldots, u_m$ for which $M_1, \ldots, M_l$ are defined and M is true. In case of success, it executes P'. In case of failure, it executes P. This is further generalized to m branches: find $(\bigoplus_{j=1}^{m} u_{j1}[\widetilde{i}] \leq N_{j1}, \ldots, u_{jm_j}[\widetilde{i}] \leq N_{jm_j}$ suchthat defined$(M_{j1}, \ldots, M_{jl_j}) \wedge M_j$ then $P_j)$ else P tries to find a branch j in $[1, m]$ such that there are values of $u_{j1}, \ldots, u_{jm_j}$ for which $M_{j1}, \ldots, M_{jl_j}$ are defined and M_j is true. In case of success, it executes P_j. In case of failure for all branches, it executes P. More formally, it evaluates the conditions defined$(M_{j1}, \ldots, M_{jl_j}) \wedge M_j$ for each j and

each value of $u_{j1}[\widetilde{i}], \ldots, u_{jm_j}[\widetilde{i}]$ in $[1, N_{j1}] \times \ldots \times [1, N_{jm_j}]$. If none of these conditions is true, it executes P. Otherwise, it chooses randomly with uniform[1] probability one j and one value of $u_{j1}[\widetilde{i}], \ldots, u_{jm_j}[\widetilde{i}]$ such that the corresponding condition is true and executes P_j. The conditional $\mathsf{if\ defined}(M_1, \ldots, M_l) \wedge M\ \mathsf{then}\ P\ \mathsf{else}\ P'$ executes P if $M_1, \ldots, M_l$ are defined and M evaluates to true. Otherwise, it executes P'. This conditional is defined as syntactic sugar for $\mathsf{find\ suchthat\ defined}(M_1, \ldots, M_l) \wedge M\ \mathsf{then}\ P\ \mathsf{else}\ P'$. The conjunct $\mathsf{defined}(M_1, \ldots, M_l)$ can be omitted when $l = 0$ and M can be omitted when it is true.

Finally, let us explain the output $\overline{c[M_1, \ldots, M_l]}\langle M'_1, \ldots, M'_k\rangle; Q$. A channel $c[M_1, \ldots, M_l]$ consists of both a channel name c and a tuple of terms $M_1, \ldots, M_l$. Channel names c allow us to define private channels to which the adversary can never have access, by $\mathsf{newChannel}\ c$. (This is useful in the proofs, although all channels of protocols are often public.) Terms $M_1, \ldots, M_l$ are intuitively analogous to IP addresses and ports which are numbers that the adversary may guess. A semantic configuration always consists of a single output process (the process currently being executed) and several input processes. When the output process executes $\overline{c[M_1, \ldots, M_l]}\langle M'_1, \ldots, M'_k\rangle; Q$, one looks for an input on channel $c[M'_l \ldots, M'_l]$, where $M'_1, \ldots, M'_l$ evaluate to the same bitstrings as $M_1, \ldots, M_l$, and with the same arity k, in the available input processes. If no such input process is found, the

[1] A probabilistic polynomial-time Turing machine can choose a random number uniformly in a set of cardinal m only when m is a power of 2. When m is not a power of 2, the distribution can be made as close as we wish to the uniform distribution.

process blocks. Otherwise, one such input process $c[M'_1, \ldots, M'_l](x_1[\widetilde{i}] : T_1, \ldots, x_k[\widetilde{i}] : T_k); P$ is chosen randomly with uniform probability. The communication is then executed: for each $j \leq k$, the output message M'_j is evaluated, its result is truncated to length $\mathrm{maxlen}_\eta(c)$, the obtained bitstring is stored in $x_j[\widetilde{i}]$ if it is in $I_\eta(T_j)$ (otherwise the process blocks). Finally, the output process P that follows the input is executed. The input process Q that follows the output is stored in the available input processes for future execution. Note that the syntax requires an output to be followed by an input process, as in [60]. If one needs to output several messages consecutively, one can simply insert fictitious inputs between the outputs. The adversary can then schedule the outputs by sending messages to these inputs.

An else branch of find or if may be omitted when it is else $\overline{yield}\langle\rangle; 0$. A trailing 0 after an output may be omitted. We may abbreviate $x[i_1, \ldots, i_m]$ by x when the replications above the considered program point are $!^{i_1 \leq N_1} \ldots !^{i_m \leq N_m}$. Variables defined under a replication must be arrays; *e.g.*, $!^{i_1 \leq N_1} \ldots !^{i_m \leq N_m}\mathsf{let}\ x[i_1, \ldots, i_m] : T = M\ \mathsf{in}\ \ldots$ and we require well-formedness invariants that guarantee that each array cell is assigned at most once, that array cells can be read only after they have been defined, and that bitstrings are of their expected types.

5.3.2 Semantics

The semantics of the calculus is defined by a probabilistic reduction relation on semantic configurations $\mathbb{C}$. We denote by initConfig(Q) the initial configuration

associated to process Q. We refer the reader to [22] for additional details on this calculus and its semantics. Given a mapping ρ from variable names to bitstrings, we write $\rho, M \Downarrow a$ when the term M (built from function symbols and variables, without array accesses) evaluates to bitstring a. We denote by $\Pr[Q \rightsquigarrow_\eta \overline{c}\langle a\rangle]$ the probability that Q outputs the bitstring a on channel c after some reductions. We denote by $\mathcal{E}$ a sequence of events of the form $e(a_1, \ldots, a_m)$, where e is an event symbol and $a_1, \ldots, a_m$ are bitstrings. We denote by $\Pr[\exists(\mathbb{C}, \mathcal{E}), \text{initConfig}(Q) \xrightarrow{\mathcal{E}} \mathbb{C} \wedge \phi(\mathbb{C}, \mathcal{E})]$ the probability that there exists a sequence of events $\mathcal{E}$ and a semantic configuration $\mathbb{C}$ such that Q reduces to $\mathbb{C}$, executing events $\mathcal{E}$ on the trace, and the logical formula $\phi(\mathbb{C}, \mathcal{E})$ holds. We denote by $\Pr[Q \rightsquigarrow_\eta \mathcal{E}] := \Pr[\exists \mathbb{C}, \text{initConfig}(Q) \xrightarrow{\mathcal{E}} \mathbb{C} \wedge \mathbb{C} \text{ does not reduce}]$ the probability that the process Q executes exactly the sequence of events $\mathcal{E}$, in the order of $\mathcal{E}$. These probabilities depend on the security parameter η; we omit it to lighten notations.

We use an *evaluation context* C to represent the adversary. An evaluation context is a process with a hole, of one of the following forms: a hole $[\,]$, a process in parallel with an evaluation context $Q \mid C$, or a restriction $\mathsf{newChannel}\ c; C$, which limits the scope of the channel c to the context C. We denote by $C[Q]$ the process obtained by replacing the hole of C with Q. When V is a set of variables defined in Q, an evaluation context C is said to be *acceptable* for (Q, V) if and only if C does not contain events, the common variables of C and Q are in V, and $C[Q]$ satisfies the well-formedness invariants. The set V contains the variables the context is

allowed to access (using find).

We denote by $\mathrm{fc}(P)$ the set of free channels of P, and by $\mathrm{var}(P)$ the set of variables that occur in P. We also use the notation $\mathrm{var}(\cdot)$ for events.

5.3.3 Observational Equivalence

Definition 5.3.1 (Observational equivalence)**.** Let Q and Q' be two processes that satisfy the well-formedness invariants. Let V be a set of variables defined in Q and Q', with the same types.

We say that Q and Q' are *observationally equivalent* with public variables V, written $Q \approx^V Q'$, when for all evaluation contexts C acceptable for (Q, V) and (Q', V), for all channels c and bitstrings a, $|\Pr[C[Q] \rightsquigarrow_\eta \overline{c}\langle a\rangle] - \Pr[C[Q'] \rightsquigarrow_\eta \overline{c}\langle a\rangle]|$ is negligible and $\sum_{\mathcal{E}} |\Pr[C[Q] \rightsquigarrow_\eta \mathcal{E}] - \Pr[C[Q'] \rightsquigarrow_\eta \mathcal{E}]|$ is negligible.

This definition formalizes that the probability that an algorithm C distinguishes the games Q and Q' is negligible. The context C is allowed to access directly the variables in V (using find). When V is empty, we write $Q \approx Q'$.

This definition makes events observable, so that observationally equivalent processes execute the same events with overwhelming probability.

5.3.4 Secrecy

In this section, we recall the formal definitions of the security properties proved by CryptoVerif [22, 25]. A variable is considered secret when the adversary has

no information on it, that is, the adversary cannot distinguish it from a random number. CryptoVerif distinguishes two notions of secrecy.

Definition 5.3.2 (One-session secrecy)**.** Assume x of type T is defined in Q under replications $!^{i_1 \leq N_1} \ldots !^{i_m \leq N_m}$. Let Q' be obtained from Q by removing events. The process Q *preserves the one-session secrecy of* x when $Q' \mid Q_x \approx Q' \mid Q'_x$, where

$$\begin{aligned} Q_x = {} & c(u_1 : [1, N_1], \ldots, u_m : [1, N_m]); \\ & \mathsf{if\ defined}(x[u_1, \ldots, u_m])\ \mathsf{then}\ \overline{c}\langle x[u_1, \ldots, u_m]\rangle \\ Q'_x = {} & c(u_1 : [1, N_1], \ldots, u_m : [1, N_m]); \\ & \mathsf{if\ defined}(x[u_1, \ldots, u_m])\ \mathsf{then\ new}\ y : T; \overline{c}\langle y\rangle \end{aligned}$$

$c \notin \mathrm{fc}(Q)$, and $u_1, \ldots, u_m, y \notin \mathrm{var}(Q)$.

The adversary interacting with Q cannot distinguish any element of the array x from a uniformly distributed random number by a single test query. The test query returns either the desired element of x or a freshly generated random number, and the adversary has to distinguish between these two situations. (This notion of secrecy does not guarantee that the random numbers in x are independent.)

Definition 5.3.3 (Secrecy)**.** Assume x of type T is defined in Q under replications $!^{i_1 \leq N_1} \ldots !^{i_m \leq N_m}$. Let Q' be obtained from Q by removing events. The process Q *preserves the secrecy of* x when $Q' \mid R_x \approx Q' \mid R'_x$, where

$$\begin{aligned} R_x = {} & !^{i \leq N} c(u_1 : [1, N_1], \ldots, u_m : [1, N_m]); \\ & \mathsf{if\ defined}(x[u_1, \ldots, u_m])\ \mathsf{then}\ \overline{c}\langle x[u_1, \ldots, u_m]\rangle \end{aligned}$$

$$R'_x = !^{i \leq N} c(u_1 : [1, N_1], \ldots, u_m : [1, N_m]);$$

$$\mathsf{if\ defined}(x[u_1, \ldots, u_m])\ \mathsf{then}$$

$$\mathsf{find}\ u' \leq N\ \mathsf{suchthat\ defined}(y[u'], u_1[u'], \ldots, u_m[u'])$$

$$\wedge\, u_1[u'] = u_1 \wedge \ldots \wedge u_m[u'] = u_m$$

$$\mathsf{then}\ \overline{c}\langle y[u'] \rangle\ \mathsf{else\ new}\ y : T; \overline{c}\langle y \rangle$$

$c \notin \mathrm{fc}(Q)$, and $u_1, \ldots, u_m, u', y \notin \mathrm{var}(Q)$.

Intuitively, the adversary cannot distinguish a process that outputs the value of the secret for several indices from one that outputs independent, uniformly distributed random numbers. In this definition, the adversary can perform several test queries on the various elements of the array x, modeled by R_x and R'_x. This corresponds to the "real-or-random" definition of security [4], which is stronger notion than the standard notion of cryptographic key secrecy (as shown in [4]).

In [25] it was shown, that if two processes Q and Q' are observational equivalent with respect to public variables $V = \{x\}$, *i.e.*, $Q \approx^{\{x\}} Q'$, and if Q preserves the (one-session) secrecy of x, then Q' also preserves the (one-session) secrecy of x. This provides the soundness of proofs of secrecy properties by sequences of games in the calculus.

When the array x contains a single element (that is, x is defined under no replication), the notions of one-session secrecy and of secrecy are equivalent.

Criteria for Proving Secrecy Properties

CryptoVerif proofs secrecy properties by checking syntactical criteria which imply the corresponding property. We note here again that proofs in CryptoVerif are sound but not complete. However, it was claimed in [22] that, although the criteria may seem restrictive, that they should be sufficient for all protocols.

CryptoVerif uses the following two criteria for one-session secrecy and secrecy, which were proved in [22]:

Proposition 5.3.4. *(Blanchet) Consider a process Q such that there exists a set of Variables S such that 1) the definition of x are either restrictions* new $x[\tilde{i}] : T = z[M_1, \ldots, M_l] : T$ *where z is defined by restrictions* new $z[i'_1, \ldots, i'_l] : T$*, and $z \in S$, and 2) all access to variables $y \in S$ in Q are of the form "* let $y'[\tilde{i}] : T' = y[M_1, \ldots, M_l]$*" with $y' \in S$. Then $Q|Q_x \approx_0 Q|Q'_x$, hence Q preserves the one-session secrecy of x.*

Proposition 5.3.5. *(Blanchet) Assume that Q satisfies the hypothesis of Proposition 5.3.4. When $\mathcal{T}$ is a trace of $C[Q]$ for some evaluation context C, we define* $\mathrm{defRestr}_{\mathcal{T}}(x[\tilde{a}])$*, the defining restriction of $x[\tilde{a}]$ in trace $\mathcal{T}$, as follows: if $x[\tilde{a}]$ is defined by* new $x[\tilde{a}] : T$ *in $\mathcal{T}$,* $\mathrm{defRestr}_{\mathcal{T}}(x[\tilde{a}]) = x[\tilde{a}]$*; if $x[\tilde{a}]$ is defined by* let $x[\tilde{a}] : T = z[M_1, \ldots, M_l]$*,* $\mathrm{defRestr}_{\mathcal{T}}(x[\tilde{a}]) = z[a'_1, \ldots, a'_l]$ *where $\rho, M_k \Downarrow a'_k$ for all $k \leq l$ and ρ is the environment in $\mathcal{T}$ at the definition of $x[\tilde{a}]$. Assume that for all evaluation contexts C acceptable for $Q, 0, x$, the probability* $\Pr(\exists(\mathcal{T}, \tilde{a}, \tilde{a'}),\ C[Q]$ *reduces according to $\mathcal{T} \wedge \tilde{a} \neq \tilde{a'} \wedge \mathrm{defRestr}_{\mathcal{T}}(x[\tilde{a}]) = \mathrm{defRestr}_{\mathcal{T}}(x[\tilde{a'}])]$ is negligible.*

Then Q preserves the secrecy of x.

5.3.5 Authentication

Authentication in CryptoVerif is modeled by correspondence properties [25]. Events $e(M_1, \ldots, M_m)$ are used in order to record that a certain program point has been reached, with certain values of $M_1, \ldots, M_m$, and the correspondence properties are properties of the form "if some event has been executed, then some other events also have been executed, with overwhelming probability". More precisely, we distinguish two kinds of correspondences.

Non-injective Correspondences

A non-injective correspondence is a property of the form "if some events have been executed, then some other events have been executed at least once". Events E are formulas of the form $\mathsf{event}(e(M_1, \ldots, M_m))$. Terms $M_1, \ldots, M_m$ in events must not contain array accesses, and their variables are assumed to be distinct from variables of processes. We write $\rho, \mathcal{E} \vdash E$ when the event E is an element of the sequence of events $\mathcal{E}$, taking into account the values of variables given by the environment ρ that maps variables to bitstrings. We define formally $\rho, \mathcal{E} \vdash E$ as follows:

$\rho, \mathcal{E} \vdash \mathsf{event}(e(M_1, \ldots, M_m))$ if and only if

for all $j \leq m$, $\rho, M_j \Downarrow a_j$ and $e(a_1, \ldots, a_m) \in \mathcal{E}$

Definition 5.3.6. The sequence of events $\mathcal{E}$ satisfies $E \Rightarrow \bigwedge_{i=1}^{k} E_i$, written $\mathcal{E} \vdash E \Rightarrow \bigwedge_{i=1}^{k} E_i$, if and only if for all ρ defined on $\mathrm{var}(E)$ such that $\rho, \mathcal{E} \vdash E$, there exists an extension ρ' of ρ to $\bigcup_{i=1}^{k} \mathrm{var}(E_i)$ such that for all $i \leq k$, $\rho', \mathcal{E} \vdash E_i$.

Intuitively, a sequence of events $\mathcal{E}$ satisfies $E \Rightarrow \bigwedge_{i=1}^{k} E_i$ when, if $\mathcal{E}$ satisfies E, then $\mathcal{E}$ satisfies all E_i. The variables of E are universally quantified; those of E_i that do not occur in E are existentially quantified.

Definition 5.3.7. The process Q satisfies $E \Rightarrow \bigwedge_{i=1}^{k} E_i$ with public variables V if and only if for all evaluation contexts C acceptable for (Q, V), $\Pr[\exists(\mathbb{C}, \mathcal{E}), \mathrm{initConfig}(C[Q]) \xrightarrow{\mathcal{E}} \mathbb{C} \wedge \mathcal{E} \not\vdash E \Rightarrow \bigwedge_{i=1}^{k} E_i]$ is negligible.

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

Injective Correspondences

Injective correspondences are properties of the form "if some event has been executed n times, then some other events have been executed at least n times". In order to model them in our logical formulae, we introduce injective events $\mathsf{inj\text{-}event}(e(M_1, \ldots, M_m))$. We write $\rho, \mathcal{E} \vdash^{\tau} E$ when the injective event E has been

executed at step τ in the sequence of events $\mathcal{E}$, taking into account the values of variables given by the environment ρ. Formally,

$\rho, \mathcal{E} \vdash^{\tau} \mathsf{inj\text{-}event}(e(M_1, \ldots, M_m))$ if and only if

for all $j \leq m$, $\rho, M_j \Downarrow a_j$ and $e(a_1, \ldots, a_m) = \mathcal{E}(\tau)$

Definition 5.3.8. Let $E, E_1, \ldots, E_n$ be injective events. The sequence of events $\mathcal{E}$ satisfies $E \Rightarrow \bigwedge_{i=1}^{k} E_i$, written $\mathcal{E} \vdash E \Rightarrow \bigwedge_{i=1}^{k} E_i$, if and only if there exist injective functions $f_1, \ldots, f_n$ such that for all ρ defined on $\mathrm{var}(E)$, for all τ such that $\rho, \mathcal{E} \vdash^{\tau} E$, there exists an extension ρ' of ρ to $\bigcup_{i=1}^{k} \mathrm{var}(E_i)$ such that for all $i \leq k$, $\rho', \mathcal{E} \vdash^{f_i(\tau)} E_i$.

Intuitively, a sequence of events $\mathcal{E}$ satisfies $E \Rightarrow \bigwedge_{i=1}^{k} E_i$ when each execution of E corresponds to distinct executions of E_i for each i. (The fact that these executions are distinct is guaranteed by the injectivity of f_i.) Definition 5.3.7 is unchanged for injective correspondences. So a process Q satisfies the injective correspondence $\mathsf{inj\text{-}event}(e(M_1, \ldots, M_m)) \Rightarrow \bigwedge_{i=1}^{k} \mathsf{inj\text{-}event}(e_i(M_{i1}, \ldots, M_{im_i}))$ if and only if, with overwhelming probability, for all values of the variables in $M_1, \ldots, M_m$, for each execution of the event $e(M_1, \ldots, M_m)$, there exist distinct corresponding executions of the events $e_i(M_{i1}, \ldots, M_{im_i})$ for $i \leq k$ for some values of the variables of M_{ij} ($i \leq k$, $j \leq m_i$) not in $M_1, \ldots, M_m$.

Criteria for Proving Correspondence Properties

Similar to the way CryptoVerif proves secrecy properties, CryptoVerif proves (injective) correspondence properties by using sufficient criteria. We present these criteria, which where proved in [25], and the necessary definitions from [25] in the following.

Definition 5.3.9. (P follows F, $\mathcal{F}_{F,P}$) When $F = \text{event}(e(M_1, \ldots, M_m))$ and P is such that $\text{event}(e(M'_1, \ldots, M'_m)); P$ occurs in Q_0, we say that P *follows* F, and we define $\mathcal{F}_{F,P} := \theta'\mathcal{F}_P \cup \{\theta' M'_j = M_j | j \leq m\}$, where the substitution θ' is a renaming of the replication indices at P to distinct fresh replication indices.

If θ is a substitution equal to the identity on the variables of ψ, then it gives values to existentially quantified variables of ϕ. We write $\mathcal{F} \models\!\Rightarrow_\theta \phi$, when it can be shown that the set of facts $\mathcal{F}$ implies $\theta\phi$. Formally:

$\mathcal{F} \models\!\Rightarrow_\theta M$ if and only if $\mathcal{F} \cup \{\neg\theta M\}$ yields a contradiction,

$\mathcal{F} \models\!\Rightarrow_\theta \text{event}(e(M_{1,\ldots,M_m}))$ if and only if there exist $M'_1, \ldots, M'_m$ such that $\text{event}(e(M'_1, \ldots, M'_m)) \in \mathcal{F}$ and $\mathcal{F} \cup \{\bigvee_{j=1}^{m} \theta M_j \neq M'_j\}$ yields a contradiction,

$\mathcal{F} \models\!\Rightarrow_\theta \phi_1 \wedge \phi_2$ if and only if $\mathcal{F} \models\!\Rightarrow_\theta \phi_1$ and $\mathcal{F} \models\!\Rightarrow_\theta \phi_2$,

$\mathcal{F} \models\!\Rightarrow_\theta \phi_1 \vee \phi_2$ if and only if $\mathcal{F} \models\!\Rightarrow_\theta \phi_1$ or $\mathcal{F} \models\!\Rightarrow_\theta \phi_2$.

Terms and events are proved by the equational prover used by CryptoVerif.

Proposition 5.3.10. *(Blanchet) Let $\psi \Rightarrow \phi$ be a non-injective correspondence, with $\psi = F_1 \wedge \ldots \wedge F_m$. If for every P_1 that follows F_1, ..., for every P_m that follows*

F_m, there exists a substitution θ equal to the identity on the variables of ψ and such that $\mathcal{F}_{F_1,P_1} \cup \ldots \cup \mathcal{F}_{F_m,P_m} \models\!\!\Rightarrow_\theta \phi$, then Q_0 satisfies $\psi \Rightarrow \phi$ with any public variables V.

In order to prove injective correspondences, it is necessary to show that if the injective events of ψ have distinct execution indices then each injective event of ϕ has different replication indices. In order to prove this, CryptoVerif collects information on the replication indices of events, for each injective event of ϕ, and stores it in a set $\mathcal{S}$ with elements of the form $(\mathcal{F}, M_0, \mathcal{I}, \mathcal{V})$; where $\mathcal{F}$ is the set of acts known to hold, M_0 contains the replication indices of the considered injective event in ϕ, $\mathcal{I} = \{j \mapsto I_{P_j} | F_j \text{ is an injective event}\}$ with $\psi = F_1 \wedge \ldots \wedge F_m$ and P_j is the process executing F_j ($j \leq m$), and $\mathcal{V}$ contains the replication indices in $\mathcal{F}$ and the variables of ψ. We obtain a pseudo-fornula $\mathcal{C}$ from ϕ by replacing each injective event with the associated set in $\mathcal{S}$ and all other leaves with $\bot$. We write $\vdash C$ if for all non-bottom leaves $\mathcal{S}$ of $\mathcal{C}$, for all $(\mathcal{F}, M_0, \mathcal{I}, \mathcal{V})$, $(\mathcal{F}', M_0', \mathcal{I}', \mathcal{V}')$ in $\mathcal{S}$, $\mathcal{F} \cup \theta''\mathcal{F}' \cup \{\bigvee_{j \in Dom(\mathcal{I})} \mathcal{I}(j) \neq \theta''\mathcal{I}'(j), M_0 = \theta'' M_0'\}$ yields a contradiction where the substitution $\theta''$is a renoming of variables in $\mathcal{V}'$ to distinct fresh variables. The condition $\vdash \mathcal{C}$ guarantees injectivity.

The definition of $\mathcal{F} \models\!\!\Rightarrow_\theta \phi$ is extended to the case of injective correspondences; and we write $\mathcal{F} \models\!\!\Rightarrow_\theta^{\mathcal{I},\mathcal{V},\mathcal{C}} \phi$ if $\mathcal{F}$ implies $\theta\phi$ and $\mathcal{C}$ correctly collects the tuples $(\mathcal{F}, M_0, \mathcal{I}, \mathcal{V})$ associated to the proof. Formally:

$\mathcal{F} \models\!\!\Rightarrow_\theta^{\mathcal{I},\mathcal{V},\bot} M$ if and only if $\mathcal{F} \cup \{\neg\theta M\}$ yields a contradiction,

$\mathcal{F}|\Rightarrow_\theta^{\mathcal{I},\mathcal{V},\perp}$ event$(e(i, M_1, \ldots, M_m))$ if and only if there exist $M'_0, M'_1, \ldots, M'_m$ such that event$(e(M'_0, M'_1, \ldots, M'_m)) \in \mathcal{F}$ and $\mathcal{F} \cup \{\theta i \neq M'_0 \vee \bigvee_{j=1}^m \theta M_j \neq M'_j\}$ yields a contradiction,

$\mathcal{F}|\Rightarrow_\theta^{\mathcal{I},\mathcal{V},\mathcal{S}}$ inj $-$ event$(e(i, M_1, \ldots, M_m))$ if and only if there exist $M'_0, M'_1, \ldots, M'_m$ such that event$(e(M'_0, M'_1, \ldots, M'_m)) \in \mathcal{F}$, $\mathcal{F} \cup \{\theta i \neq M'_0 \vee \bigvee_{j=1}^m \theta M_j \neq M'_j\}$ yields a contradiction, and $(\mathcal{F}, M'_0, \mathcal{I}, \mathcal{V}) \in \mathcal{S}$,

$\mathcal{F}|\Rrightarrow_\theta^{\mathcal{I},\mathcal{V},\mathcal{C}_1 \wedge \mathcal{C}_2} \phi_1 \wedge \phi_2$ if and only if $\mathcal{F}|\Rrightarrow_\theta^{\mathcal{I},\mathcal{V},\mathcal{C}_1} \phi_1$ and $\mathcal{F}|\Rrightarrow_\theta^{\mathcal{I},\mathcal{V},\mathcal{C}_2} \phi_2$,

$\mathcal{F}|\Rrightarrow_\theta^{\mathcal{I},\mathcal{V},\mathcal{C}_1 \vee \mathcal{C}_2} \phi_1 \vee \phi_2$ if and only if $\mathcal{F}|\Rrightarrow_\theta^{\mathcal{I},\mathcal{V},\mathcal{C}_1} \phi_1$ or $\mathcal{F}|\Rrightarrow_\theta^{\mathcal{I},\mathcal{V},\mathcal{C}_2} \phi_2$.

With these definitions we are now ready to state the proposition that is used by CryptoVerif to proof injective correspondeces.

Proposition 5.3.11. *(Blanchet) Let $\psi \Rightarrow \phi$ be a correspondence with $\psi = F_1 \wedge \ldots \wedge F_m$. Assume that, for all events e used as injective events in $\psi \Rightarrow \phi$, two occurences of the event e always occur in different branches of* find *or* if *in Q_0. Assume that there exists a pseudo–formula $\mathcal{C}$ such that $\vdash \mathcal{C}$ and for every P_1 that follows F_1, ..., for every P_m that follows F_m there exists a substitution θ equal to the identity on the variables of ψ and such that $\mathcal{F}_{F_1,P_1} \cup \ldots \cup \mathcal{F}_{F_m,P_m} |\Rrightarrow_\theta^{\mathcal{I},\mathcal{V},\mathcal{C}} \phi$, where $\mathcal{I} = \{j \mapsto I_{P_j} | F_j$ is an injective event $\}$ and $\mathcal{V} = \mathrm{var}(I_{P_1}) \cup \ldots \cup \mathrm{var}(I_{P_m}) \cup \mathrm{var}(\psi)$. Then Q_0 satisfies $\psi \Rightarrow \phi$ with any public variables V.*

5.3.6 Modeling Kerberos in CryptoVerif

As an example, we present the client role of the first round of basic Kerberos (AS Exchange) from Figure 2.4 in the process calculus. More processes are detailed in Appendix .5. (The full CryptoVerif scripts are available at `http://www.cryptoverif.ens.fr/kerberos/`.)

$Q_C = !^{i_C \leq N}\, c_1[i_C](hostT : tgs);$ new $n_1 : nonce;$

$\overline{c_2[i_C]}\langle C, hostT, n_1\rangle;$

$c_3[i_C](=C, TGT : bitstring, m_2 : bitstring);$

let injbot(concat1$(AK, =n_1, t_K, =hostT)$)

$= \mathrm{dec}(m_2, k_C)$ in

event $fullCK(hostT, n_1, TGT, m_2);$

$\overline{c_{18}[i_C]}\langle acceptC1(K)\rangle;$

$(Q_{C2}[i_C]).$

Figure 5.2: CryptoVerif formalization of client's actions in AS exchange.

The replicated process $!^{i_C \leq N} P$ represents N copies of P, available simultaneously, where N is assumed to be polynomial in the security parameter η. These copies are indexed by the integer value $i_C \in [1, N]$. Each copy starts with an input $c_1[i_C](hostT : tgs)$. This input receives a message $hostT$ on channel $c_1[i_C]$. The channel is indexed by i_C, so that, by sending a message on channel $c_1[i]$ for a certain value of i, the adversary can choose which copy of the process receives the message

and is then executed. The message $hostT$ is the name of the ticket granting server (TGS) to which the client is going to send his request. The type tgs is a set of bitstrings that contains the representation of all possible names of ticket granting servers. The message is received only if it belongs to this type.

Next, the process chooses a random nonce n_1 uniformly in the type $nonce$, by the construct $\mathsf{new}\ n_1 : nonce$. (A probabilistic Turing machine can choose a number uniformly at random only from sets whose size is a power of 2; to make sure that the choice above is possible, we assume that $nonce$ consists of all bitstrings of a certain length.) The process then sends the first message of the protocol $C, hostT, n_1$ on channel $c_2[i_C]$. This message will be received by the adversary; the adversary can do whatever he wants with it, but in order to run a normal session of the protocol, he should send this message to the Kerberos authentication server (KAS).

After sending the message on channel $c_2[i_C]$, the control is returned to the adversary and the process $c_3[i_C](\ldots); \ldots$ is made available. This process waits for the second message of the protocol and will be executed when a message is sent on channel $c_3[i_C]$. The expected message is $C, TGT, m_2 = C, \{AK, t_K, C\}_{k_T}, \{AK, n_1, t_K, hostT\}_{k_C}$. The message received on channel $c_3[i_C]$ then consists of three parts: the client name C, the TGT TGT, and the message $m_2 = \{AK, n_1, hostT\}_{k_C}$. The process checks that the first component of this message is C by using the pattern $=C$; the two other parts are stored in variables.

The process Q_C cannot check the TGT, which is encrypted under a key the client

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

When a check fails, the control is returned to the adversary. When all checks succeed, Q_C executes the event $fullCK(hostT, n_1, TGT, m_2)$. Executing this event does not affect the execution of the protocol; it just records that a certain program point is reached with certain values of the variables. Events are used for specifying authentication properties, as explained in Section 5.3.5. After executing the event, Q_C announces that it has completed the AS_exchange with K by sending the message $acceptC1(K)$ on channel $c_{18}[i_C]$. Afterwards the process Q_{C2} is executed which starts the TG_exchange.

In this calculus, all variables defined under replications are implicitly arrays,

indexed by the indices of these replications. For instance, the variable $hostT$ defined under $!^{i_C \leq N}$ is in fact $hostT[i_C]$, so that each copy of the replicated process stores the value of $hostT$ in a distinct cell of the array. The arrays allow us to keep track of the whole state of the system. In the cryptographic proofs, the arrays used in the calculus of CryptoVerif replace lists often used by cryptographers. As an example of the use of lists, suppose that symmetric encryption satisfies ciphertext integrity (INT-CTXT). This assumption means that, when decryption succeeds, the considered ciphertext has been generated by calling the encryption function with the same secret key (provided the key is not leaked). Then, one usually stores the computed ciphertexts in a list, and upon decryption, one can additionally check that the ciphertext is in the list. In our calculus, the computed ciphertexts are always automatically stored in an array, instead of a list, which avoids having to add explicit list insertion instructions. The calculus provides an array lookup construct, detailed in [22].

The one-session secrecy of x is coded in CryptoVerif by the query secret1 x, while the secrecy of x is coded by secret x.

5.4 Computationally Sound Results

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

then present the authentication and secrecy results.

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

- The user needs to know the process calculus well enough to understand how exactly CryptoVerif applies the security of cryptographic primitives and to be able to read the last game of a CryptoVerif proof (which is not trivial and needs some practice). The latter is particularly important for interactive proofs.
- The user must know the underlying cryptography well enough to be able to specify the security of cryptographic primitives through indistinguishable oracles, although many primitives have already been specified in previous examples [22] and the user can copy them from there.

Furthermore, we note that Kerberos is a well-studied protocol and we found the previous work on Kerberos 5 [30, 37, 8] very valuable, as it gave us a good sense for which results we could expect to be verified by CryptoVerif in the computational model. This helped us, in cases in which we initially could not verify an expected property, to narrow down the cause—mostly issues with the underlying cryptography but, in rare cases, also issues with CryptoVerif itself (see also Section 5.4.5).

5.4.1 Cryptographic Assumptions

In our analysis, the public-key encryption scheme is assumed to be indistinguishable under adaptive chosen ciphertext attacks (IND-CCA2), and the signature scheme

is assumed to be unforgeable under chosen message attacks (UF-CMA). Symmetric encryption is assumed to be indistinguishable under chosen plaintext attacks (IND-CPA) and to satisfy ciphertext integrity (INT-CTXT). These properties guarantee indistinguishability under adaptive chosen ciphertext attacks (IND-CCA2), as shown in [19]. These assumptions are the same as in [8], and Boldyreva and Kumar showed in [27] that the encryption of the simplified profile of basic Kerberos satisfies these properties for symmetric encryption. They also showed that the general profile encryption is weak, and propose a corrected version of the general profile encryption that satisfies these properties.

The keyed hash function used to compute the checksum in PKINIT is assumed to be a message authentication code, weakly unforgeable under chosen message attacks (UF-CMA), which is in accordance with [18] and which also matches the assumptions in [8]. As it is unwise to use the same key for multiple cryptographic operations, a key derivation function is used to generate multiple keys from a base key; this key derivation function takes as input the base key and publicly known integer called *usage number* [72]. We assume that the key derivation function is a pseudo-random function and use it to derive, from a base key, a key for the message authentication code and another key for the encryption of the message component that includes the authentication key for the client (denoted k in Figure 3.5). The assumed security notions for the cryptographic primitives are presented as pairs of indistinguishable oracles in Figures 5.3, 30, 31, 30. We note that in the specifica-

tions [57, 70] a key derivation function is used not only for the key mentioned above but for all symmetric keys (even if they are not used for multiple cryptographic operations); we, however, restrict the use of a key derivation function to the key above. Implementing the use of a key derivation function for all symmetric keys will be part of future work.

For basic Kerberos, we assume that the long-term key k_C shared between the client and the KAS is generated from a random seed, although in practice this key is usually generated from a password and is vulnerable to dictionary attacks [20].

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

5.4.2 Authentication results

Here we present authentication properties directly proved in the computational model by CryptoVerif 1.06pl3 under the assumptions from Section 5.4.1.

We formalize Property 7(1) from Section 5.2 as the following theorem.

Theorem 5.4.1 (Authentication of KAS to client)**.** *In basic and public-key Kerberos, for each instance of:*

- *an honest client C completing the AS exchange with KAS K,*

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

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

- *the KAS completing the AS exchange with C,*
- *in which the KAS received the request m_{req} for a TGT for the use between C and T,*
- *sent reply m_{rep}, where all message components of m_{rep}, except the TGT, are equal to the corresponding components in m'_{rep}.*

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

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

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

Figure 5.3: Definition of INT-CTXT and IND-CPA symmetric encryption in CryptoVerif

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

- It removes assignments on k_S, that is, it replaces k_S with its value kgen(rKs): k_S is generated from a random seed rKs by the key generation algorithm kgen.
- The variable *Pkey* is assigned at two places in the game, either with the key $k_S =$ kgen(rKs), when T and S are honest, or with a key coming from the adversary. CryptoVerif renames these two assignments to *Pkey* to distinct names *Pkey_88* and *Pkey_87* respectively, which leads to distinguishing two cases, depending on whether *Pkey* is shared between honest T and S or not.
- CryptoVerif removes assignments on *Pkey_88*, that is, it replaces *Pkey_88* with its value kgen(rKs).
- CryptoVerif applies the INT-CTXT property of the symmetric encryption on the key $k_S =$ kgen(rKs). The INT-CTXT property is represented in CryptoVerif by the equivalence (INT-CTXT) of Figure 5.3. In this equivalence, the left-hand side chooses a random seed r and provides two oracles: the first one encrypts its argument x under key kgen(r) generated from r, using fresh coins r'; the second one decrypts its argument y with key kgen(r). The right-hand side provides two corresponding oracles: the first one still encrypts under kgen(r), but additionally stores the ciphertext in the variable z. This variable is implicitly an array

indexed by the number of the call to the encryption oracle. The second oracle, instead of decrypting its argument y, looks for y in the array z that contains all computed ciphertexts. When y is found in this array, that is, there exists j such that $z[j] = y$, the oracle returns the corresponding plaintext $x[j]$, injected by $\mathrm{i}_\perp$ into the set of bitstrings union the special symbol $\perp$. When no such y is found, the oracle returns $\perp$, meaning that decryption failed. Ciphertext integrity implies that the left-hand side and the right-hand side are indistinguishable for an attacker: with overwhelming probability, the attacker is unable to produce a valid ciphertext without calling the encryption oracle, so the valid ciphertexts are those stored in z and decryption succeeds if and only if the ciphertext is found in the array z.

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

As a final technical detail, the right-hand side of the equivalence uses the function symbol kgen' instead of kgen: this prevents repeated application of the game transformation since after transformation, terms of the form $\mathrm{enc}(x, \mathrm{kgen}(r), r')$ are no longer found.

- After each cryptographic transformation, the game is simplified. CryptoVerif uses essentially equational reasoning to replace terms with simpler terms and tries to determine the result of tests, thus removes branches that cannot be executed. In particular, if the initial game contained a statement of the form $\mathsf{let}\ \mathrm{injbot}(\mathrm{concat2}(SK, t_T, hostC)) = \mathrm{dec}(M, k_S)\ \mathsf{in}\ \ldots$, the decryption has been replaced by a lookup that returns plaintexts, so simplification can then select only the branche(s) of the lookup that return a value that can be equal to $\mathrm{i}_\perp(\mathrm{concat2}(SK, t_T, hostC))$.

  The simplification also removes collisions between random numbers: for instance, when a test requires that two independent random nonces are equal, this test fails with overwhelming probability.

- CryptoVerif applies the IND-CPA property of the symmetric encryption on the key $k_S = \mathrm{kgen}(rKs)$. The IND-CPA property is represented in CryptoVerif by the equivalence (IND-CPA) of Figure 5.3. This equivalence expresses that the oracle that encrypts x is indistinguishable from an oracle that encrypts $Z(x)$, where $Z(x)$ represents a bitstring of zeroes, of the same length as x. This property is implied by IND-CPA.

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

The right-hand side uses enc' instead of enc to prevent repeated application of the game transformation.

- After applying this transformation, the game is simplified. In particular, terms of the form $Z(M)$ are simplified to constants when the length of M is constant, which removes the dependency on M.

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

The probability $P(t)$ that an attacker running in time t breaks the correspondence $\mathsf{inj\text{-}event}(\mathit{fullCK}(T, n, x, y)) \Rightarrow \mathsf{inj\text{-}event}(\mathit{fullKC}(C, T, n, z, y))$ is bounded by CryptoVerif by $P(t) \leq \frac{N^2}{2\,|nonce|} + \frac{N}{|nonce|} + P_{\text{INT-CTXT}}(t + t_{C1}, N, N) + P_{\text{IND-CPA}}(t + t_{C2}, N) + P_{\text{INT-CTXT}}(t + t_{C3}, N, N) + P_{\text{IND-CPA}}(t + t_{C4}, N) + P_{\text{INT-CTXT}}(t + t_{C5}, N, N)$ where N is the maximum number of sessions of the protocol participants, $|nonce|$ is the cardinal of the set of nonces, $P_{\text{INT-CTXT}}(t, n, n')$ is the probability that an attacker running in time t breaks the INT-CTXT equivalence with at most n calls to the encryption oracle and n' calls to the decryption oracle (for one encryption key), $P_{\text{INT-CPA}}(t, n)$ is the probability that an attacker running in time t breaks the IND-CPA equivalence with at most n calls to the encryption oracle, and t_{C1}, t_{C2}, t_{C3}, t_{C4}, and t_{C5} are bounds on the running time of the part of the transformed

games not included in the INT-CTXT or IND-CPA equivalence, which are therefore considered as part of the attacker against the INT-CTXT or IND-CPA equivalence. The first two terms of $P(t)$ come from elimination of collisions between nonces, while the other terms come from cryptographic transformations using the INT-CTXT or IND-CPA properties of encryption for keys k_S, k_T, and k_C. (Only the INT-CTXT property is used for k_C.)

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

Public-key Kerberos case: when the client process completes its participation in PKINIT, it executes an event $\mathit{fullCK}(\mathit{hostZ}, \mathit{hostT}, n_1, \mathit{m21}, \mathit{TGT}, \mathit{m24})$ that contains the name *hostZ* of the KAS, the name *hostT* of the TGS and the unsigned nonce n_1 from the client's first message. Furthermore, it contains the reply from *hostZ* in *m21*, *TGT*, and *m24*, where *m21* is the part of the reply encrypted under *C*'s public key and *m24* is the reply component that contains the authentication key (AK). When the KAS process completes its participation in PKINIT, it executes

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

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

the key derivation function (*i.e.*, pseudo randomness), and the command `simplify` instructs CryptoVerif to apply the build-in simplification algorithm to the current game. The command `auto` instructs CryptoVerif to continue the proof automatically, using its built-in proof strategy. □

Theorems 5.4.2–5.4.5 below can be proved in a similar way. We detail the proof of Theorem 5.4.4 as a second example; details to other proofs are in the appendix.

We formalize Property 7(2) from Section 5.2 as the following theorem.

Theorem 5.4.2 (Authentication of request for ST)**.** *In basic and public-key Kerberos, if there is an instance of:*

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

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

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

and an instance of:

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

We formalize Property 7(3) from Section 5.2 as the following theorem.

Theorem 5.4.3 (Authentication of TGS to client)**.** *In basic and public-key Kerberos, for each instance of:*

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

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

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

We formalize Property 7(4) from Section 5.2 as the following theorem.

Theorem 5.4.4 (Authentication of request to server)**.** *In basic and public-key Kerberos, if there is an instance of:*

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

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

- *the TGS completing a TG exchange with C,*

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

and an instance of:

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

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

$\mathsf{event}(\mathit{fullTC}(C, S, n, v, v', z, w))$.

Public-key Kerberos case: As the CS exchange in public-key Kerberos does not differ from the CS exchange in basic Kerberos V5, the events $\mathit{partSC}(\mathit{hostC}, \mathit{m14}, \mathit{m15})$, $\mathit{partCS}(\mathit{hostX}, \mathit{hostY}, \mathit{ST}, \mathit{e12})$, and $\mathit{fullTC}(\mathit{hostY}, \mathit{hostW}, n', \mathit{m8}, \mathit{m9}, \mathit{ST}', \mathit{e11})$ are just as the ones described above in the basic Kerberos case. CryptoVerif can then prove the same query as above in basic Kerberos case using the same commands as in the public-key Kerberos case from Theorem 5.4.1. □

We formalize Property 7(5) from Section 5.2 as the following theorem.

Theorem 5.4.5 (Authentication of server to client)**.** *In basic and public-key Kerberos, if there is an instance of:*

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

then, with overwhelming probability, there is an instance of

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

We note that the injectivity of the correspondences in Theorems 5.4.1 and 5.4.3 stems from their challenge-response character; *i.e.*, a fresh nonce is sent and subsequently received. The correspondences in Theorems 5.4.2 and 5.4.4 are non-injective

because the 3rd and 5th messages of Kerberos, respectively, can be replayed. In practice, however, the server should use an anti-replay cache in order to prevent the replay of the 5th message [70]; we do not yet include this cache in our model, but doing so in the future may allow us to show that each instance of the server corresponds to a distinct instance of the client in Theorem 5.4.4. The reason for the non-injectivity of the correspondence in Theorem 5.4.5, however, is a little different and has to do with how we model timestamps in CryptoVerif. If the client C sends two requests to the server S with the same timestamp t'_C, then the adversary can prevent the second request from reaching S and replay S's reply to the first request as reply for the second request. In this case, two sessions of the client correspond to a single session of the server, so the correspondence is non-injective. Our model in CryptoVerif allows the timestamps of several requests to be equal with non-negligible probability, as in the above scenario. However, this is rather unlikely to happen in the real world, since the timestamps have a 1 μs resolution. If we treat timestamps as nonces, which can be equal only with negligible probability, then the correspondence of Theorem 5.4.5 can be shown to be injective (but timestamps are then considered as unguessable).

Remark 5.4.6. CryptoVerif can prove all correspondences for basic Kerberos mentioned in Theorems 5.4.1–5.4.5 simultaneously, using a single sequence of games, and likewise it can prove the correspondences for public-key Kerberos simultaneously, using the same interactive commands.

5.4.3 Key Secrecy and Strong Key Usability

In the following we present the key secrecy results we proved in the computational model using CryptoVerif 1.06pl3 under the assumptions from Section 5.4.1. First we will discuss key indistinguishability results and then we will discuss key secrecy results with respect to the notion of *key usability*, introduced in [44] and generalized here.

Key Indistinguishability

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

Proposition 5.4.7. *a) Kerberos does not offer 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.*

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. To see that Kerberos does not offer cryptographic key secrecy for SK af-

ter 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 ciphertext $\{C, t'\}_{SK}$ 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. □

For similar reasons, one also has the next proposition

Proposition 5.4.8. *Kerberos does not offer 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.*

We formalize Property 8(1) from Section 5.2 as the following theorem.

Theorem 5.4.9 (Secrecy of AK)**.** *Let $Q_{K5_{1R}}$ be the game in the process calculus formalizing solely the AS exchange of basic Kerberos and let Q_{PKINIT} be the game formalizing the public-key mode of PKINIT. Furthermore, let keyAK denote in $Q_{K5_{1R}}$ and in Q_{PKINIT}, respectively, the authentication key received by an honest client from the KAS and generated by the KAS for the use between the client and an honest TGS. Then $Q_{K5_{1R}}$ and Q_{PKINIT} preserve the secrecy of keyAK.*

Proof. For both basic Kerberos and public-key Kerberos: when the client process completes its participation in an AS exchange with an honest TGS it stores the authentication key AK in $keyAK$. CryptoVerif can then prove the query: secret $keyAK$, where in the public-key Kerberos case, the commands as in the public-key Kerberos case from Theorem 5.4.1 are used. □

Remark 5.4.10. For the flawed draft version of PKINIT, CryptoVerif was not able to produce a positive proof of either the secrecy of the key AK or the authentication of K to C. In fact, neither property holds for the flawed protocol, due to a known attack [37].

We formalize Property 8(2) from Section 5.2 as the following theorem.

Theorem 5.4.11 (Secrecy of SK)**.** *Let $Q_{K5_{2R}}$ be the game in the process calculus formalizing the AS and the TG exchange (*i.e.*, the first two rounds) of basic Kerberos and let $Q_{PK_{2R}}$ be the game formalizing the first two rounds of public-key Kerberos.*

Furthermore, let keySK denote in $Q_{K5_{2R}}$ and in $Q_{PK_{2R}}$, respectively, the service key received by an honest client from an honest TGS and generated by the TGS for the use between the client and an honest server. Then $Q_{K5_{2R}}$ and $Q_{PK_{2R}}$ preserve the secrecy of keySK.

Proof. For both basic Kerberos and public-key Kerberos: when the client process completes its participation in a TG exchange with an honest TGS it stores the session key SK in $keySK$. CryptoVerif can then prove the query: secret $keySK$, where in the public-key Kerberos case, the same commands as in the public-key Kerberos case from Theorem 5.4.1 are used. □

We note that cryptographic secrecy, *i.e.*, indistinguishability from random, which follows for the keys AK and SK from Theorems 5.4.9 and 5.4.11, respectively, does not hold any longer once AK is used in a TS request or SK is used in a CS request, as shown in [8]. This is due to the fact that AK and SK are used to encrypt the authenticators in the TG and CS exchange, respectively, which contain a partially known plaintext; namely the client name and a timestamp which was generated during a bounded time period that is typically known to the adversary. If an adversary tries to distinguish either AK or SK from random keys, he just needs to attempt to decrypt the appropriate authenticator and makes his guess dependent on whether the adversary sees the client's name and a timestamp generated in the bounded time period or not. This gives the adversary an overwhelming advantage of guessing correctly. However, Kerberos allows for the generation of an optional

sub-session key [70], which is intended for the encryption of subsequent communication (instead of the session key). This optional sub-session key may be generated by either the client or the server in the CS exchange and included in the message which the client or the server send to each other encrypted under the session key. In [8] it was noted that the optional sub-session key satisfies the notion of cryptographic key secrecy, independent of whether it is generated by the client or the server.

Theorem 5.4.12 (Secrecy of Optional Sub-Session Key)**.**
Let $Q_{K5}^{Opt,C}$ and $Q_{PK}^{Opt,C}$ be the games in the process calculus formalizing basic Kerberos and public-key Kerberos, where in both cases an optional subsession key is generated by the client. And let $Q_{K5}^{Opt,S}$ and $Q_{PK}^{Opt,S}$ be the games in the process calculus formalizing basic Kerberos and public-key Kerberos, when an optional subsession key is generated by the server. If OPkeyC and OPkeyS denote in all cases the sub-session keys an honest client and an honest server, respectively, possess after having communicated via a Kerberos session involving an honest TGS, then

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

Proof. For both basic Kerberos and public-key Kerberos: when the client process completes its participation in a CS exchange with an honest server and involving an honest TGS it stores the optional sub-session key in *OPkeyC*, and, likewise,

the server process stores the optional sub-session key in $OPkeyS$. If the optional sub-session key is generated by the server, then CryptoVerif can prove the queries: secret1 $OPkeyC$ and secret $OPkeyS$, if the commands in the public-key Kerberos case are the same as in the proof of Theorem 5.4.1. The appropriate queries are proved by CryptoVerif if the optional sub-session key is generated by the client. □

In order to understand why CryptoVerif can in some instances only prove one-session secrecy but not secrecy, we distinguish the cases in which a server receives an optional sub-session key generated by the client from the cases in which the client receives a sub-session key generated by a server. In the first case, an adversary can force the server to accept the same sub-session key in multiple sessions that use the same session key SK, by replaying the 5th message of Kerberos. This replay allows an adversary to distinguish these sub-session keys from independent random keys. However, in practice, this replay should be prevented by an anti-replay cache of the server [70], which is not included in our model. The second case stems, again, from the fact that our CryptoVerif model allows two timestamps to be equal with a non-neglible probability (see discussion at the end of Section 5.4.2). This makes it possible for an adversary to launch a similar attack as above by replaying the response from a server in multiple sessions that use the same key SK and the same timestamp t'_C. If we treat timestamps as nonces so that two timestamps can be equal only with a negligible probability, then CryptoVerif can prove secrecy of the sub-session in the second case.

Key Usability

Weaker than key indistinguishability, the notion of *key usability* [44] aims to capture whether an exchanged key, although possibly not indistinguishable from random, is still "good" to be used subsequently for certain cryptographic operations, *e.g.*, IND-CCA secure encryption. An exchanged key, which is indistinguishable from random, can be used just as a freshly generated key for any cryptographic operations. This notion, however, could sometimes be considered as a too strong since, *e.g.*, keys that are used for encryption of a partially known payload during a key exchange protocol, as is the case in Kerberos, involuntarily become distinguishable. Nonetheless, a dinstinguishable key may still be *usable* and leave an adversary with an at most negligible advantage at winning, *e.g.*, an IND-CCA attack game.

Paralleling the definition of key indistinguishability, the definition of key usability by Datta et al. [44] involves a two-phase attacker $\mathcal{A} = (\mathcal{A}_e, \mathcal{A}_c)$. Informally, given a key exchange protocol Σ and a class of applications S, in the *key exchange phase*, honest parties first run (multiple) sessions of the protocol over a network that is controlled by $\mathcal{A}_e$. Afterwards the attacker $\mathcal{A}_e$ chooses a session and hands the session id together with the information she collected over to $\mathcal{A}_c$. Now the *challenge phase* begins where $\mathcal{A}_c$ is trying to win an attack game against a scheme $\Pi \in S$ which uses keys from the session previously picked by $\mathcal{A}_e$. The syntax of the process calculus used by CryptoVerif does not allow us to formalize a sequence consisting of an exchange phase followed by an challenge phase, nor does it allow us

to directly formalize a two-phase attacker who picks a session ID and its key to play the attack game against. Therefore we use an 'auxiliary construction' to prove key usability results for Kerberos using CryptoVerif, which in fact enables us to prove a stronger version of key usability in the case of Kerberos, as we describe in the following, and which, therefore, may contribute to future discussions on the notion of key usability. Our construction involves two aspects that address the syntactical obstacles mentioned above: Firstly, the syntax of the process calculus used by CryptoVerif forces us to let the processes formalizing the exchange phase and the challenge phase run in parallel, *i.e.*, an attacker playing, *e.g.*, an IND-CCA2 game against a symmetric encryption scheme which uses the session key SK, is still able to interact with Kerberos protocol sessions and could utilize these protocol sessions in order to win the IND-CCA2 attack game. However, we make some restriction in definition 5.4.13 below which implies, for instance, that if the adversary is trying to win an IND-CCA2 attack game against the symmetric encryption scheme under the cryptographic assumptions in Section 5.4.1 using the session key SK, then we do not allow the adversary to send any output of the encryption oracle to sessions of the honest protocol principals that are carrying out the CS exchange (*i.e.*, the third round). Secondly, instead of letting the adversary choose the session ID and the key for the attack game, the key is drawn at random from the polynomially many sessions and keys.

Definition 5.4.13 (Strong Key Usability)**.** Let $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D}) \in S$ be a symmetric

encryption scheme, $b \in \{0,1\}$, Σ a key exchange protocol, and $\mathcal{A}$ an adversary. We consider the following experiment $\mathbf{Exp}^{*b}_{\mathcal{A},\Sigma,\Pi}(\eta)$:

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

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

It is clear that allowing the adversary to interact with protocol sessions during the attack game gives the adversary more power compared to a two-phase attacker as in [44]. Not letting the adversary pick the session ID (which corresponds to a replication index in CryptoVerif's process calculus), on the other hand, restricts the adversary's capabilities. However, since the number of sessions is polynomial (in the security parameter), a non-negligible advantage of winning an attack game for a two-phase adversary as in [44] implies a non-negligible advantage for the adversary we described above. More precisely, if $\mathbf{Adv}_1$ is the advantage of a two-phase attacker as in [44] who can choose the session key to attack and if $\mathbf{Adv}_2$ is the advantage of an two-phase attacker that chooses a session at random. Then one has $\mathbf{Adv}_2 \geq \frac{1}{p} * \mathbf{Adv}_1$, where p is the number of sessions. In particular, if $\mathbf{Adv}_1$ is non-negligible then $\frac{1}{p} * \mathbf{Adv}_1$ is non-negligible, and therefore $\mathbf{Adv}_2$ will also be non-negligible. And since the adversary from Definition 5.4.13 is even less restricted than the adversary with $\mathbf{Adv}_2$, since it can interact with uncompleted protocol sessions, the adversary from Definition 5.4.13 has an even greater advantage, which we will show to be negligible in the case of Kerberos.

Furthermore, an attacker in [44] may be more restricted than necessary in order to model many realistic settings. For instance, if a key k is exchanged through protocol Σ_1 to be used in an application protocol Σ_2, then usability of k with respect to the definition in [44] guarantees k to be good for, say, encryption in Σ_2 under the condition that all users on the network stop running protocol Σ_1, which

is generally not very realistic. On the other hand, if one requires that messages encrypted under k during a run of Σ_1 differ syntactically from messages encrypted in Σ_2 then messages encrypted under k in Σ_2 will be rejected from participants of Σ_1. Therefore a restriction on the adversary like the one in definition 5.4.13 could be realized and k can be securely used in Σ_2 if it satisfies strong key usability. This example suggests yet another definition of key usability; one which comes with a composition theorem for protocols Σ_1 and Σ_2. We intend to explore in subsequent work such a variant definition of key usability and how one could utilize CryptoVerif to prove that an exchanged key satisfies that notion.

We formalize Property 8(3) from Section 5.2 as the following theorem. We omit its proof and detail only the proof of the more important result on usability of SK (Theorem 5.4.16 below), which is similar.

Theorem 5.4.14 (Usability of AK)**.** *Let $Q^{AK,use}_{\Sigma,X}$ be the game in the process calculus formalizing the experiment $\mathbf{Exp}^{*b1}_{\mathcal{A},\Sigma,\Pi}(\eta)$, where Σ is basic or public-key Kerberos involving client C, TGS T, and KAS K, Π is the symmetric encryption scheme of Kerberos, and the left-right oracle uses an authentication key AK that was locally output after a completed process of $X \in \{C, T, K\}$. If C, T, and K are honest, then $Q^{AK,use}_{\Sigma,X}$ preserves the secrecy of $b1$.*

Corollary 5.4.15. *Basic and public-key Kerberos satisfy IND-CCA2 (strong) key usability for the authentication key AK, for the symmetric encryption scheme of Kerberos.*

We formalize Property 8(4) from Section 5.2 as the following theorem.

Theorem 5.4.16 (Usability of SK)**.** *Let* $Q^{SK,use}_{\Sigma,X}$ *be the game in the process calculus formalizing the experiment* $\mathbf{Exp}^{*b1}_{\mathcal{A},\Sigma,\Pi}(\eta)$*, where* Σ *is basic or public-key Kerberos involving client* C*, KAS* K*, TGS* T*, and server* S*,* Π *is the symmetric encryption scheme of Kerberos, and the left-right oracle uses an authentication key SK that was locally output after a completed process of* $X \in \{C, S, T\}$*. If* C*,* K*,* T*, and* S *are honest, then* $Q^{SK,use}_{\Sigma,X}$ *preserves the secrecy of b1.*

Corollary 5.4.17. *Basic and public-key Kerberos satisfy IND-CCA2 (strong) key usability for the service key SK, for the symmetric encryption scheme of Kerberos.*

Proof of Theorem 5.4.16. Basic Kerberos case: In the case $X = C$, the client process completes its participation in a CS exchange involving an honest TGS, it stores the session key *SK* in *keyCSK*. From these keys one is drawn at random and passed to the encryption oracle and decryption oracle. For the boolean *b1* used by the encryption oracle, we can, using CryptoVerif, prove the query: secret *b1*. This proof requires the user to inspect the last game, which CryptoVerif reaches upon the command `auto`, in order to verify that terms that are dependent on *b1* and which may help an adversary in guessing *b1* occur only in find branches that are never executed. The case $X = T$ is similar, where the session key *SK* is stored in *keyTSK* after the TGS sent the TS_reply. And an analogous result holds for $X = S$, where the proof requires the following commands before the manual inspection of the last game: `auto`, `SArename SK_33`, `simplify`, and `auto` (formatted and entered as de-

scribed above). The command `SArename SK_33` is used when the variable `SK_33` is defined several times in the game. It instructs CryptoVerif to rename each definition of this variable to a different name, which subsequently allows to distinguish cases depending on the program point at which the variable has been defined.

Public-key Kerberos case: analogously to the basic Kerberos case, the secrecy of the bit *b1* can be concluded by inspecting the last game that CryptoVerif reaches after a sequence of commands. If $X = C$ or $X = T$, the interactive commands are just the ones from the public-key Kerberos case of Theorem 5.4.1. If $X = S$, the secrecy of *b1* can be concluded after the sequence of commands: `crypto sign rkCA`, `crypto sign rkCs`, `crypto penc rkC`, `crypto sign rkKs`, `crypto keyderivation`, `simplify`, `crypto keyderivation`, `simplify`, `auto`, `SArename SK_55`, `simplify`, and `auto` (formatted and entered as described above). □

5.4.4 Varying the Strength of Cryptography

We observe that the symbolic proofs of security for Kerberos in, *e.g.*, [30] do not rely on the secrecy of the encrypted data within the authenticators ($\{C, t_C\}_{AK}$ and $\{C, t'_C\}_{SK}$) sent by the client to the TGS and end server. CryptoVerif is also able to prove security properties for Kerberos without relying on the secrecy of the authenticator data. In particular, we can modify CryptoVerif scripts so that the client sends a second, *unencrypted* copy of the authenticator contents alongside the authenticator and CryptoVerif can still prove security properties for Kerberos. For

the case that the client sends a subsession key in the CS exchange authenticator, we make this modification only in the TG exchange; if the server sends the subsession key (but not the client), then we may make this modification in both the TG and CS exchanges. Using CryptoVerif, we can then prove the following theorem about authentication and secrecy when the authenticator contents are leaked as just described.

Theorem 5.4.18. *If*

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

then Theorems 5.4.1–5.4.5 and 5.4.12 hold for both basic and public-key Kerberos.

Proof. If we modify the CryptoVerif scripts to expose the authenticator contents as described, CryptoVerif proves the queries needed for proving Theorems 5.4.1–5.4.5 and 5.4.12; in the case of public-key Kerberos, the interactive commands are the same as before. □

Similar results might be achieved by suitably relaxing the assumptions about the encryption function used for the authenticators. That, and studies of other ways in

which the cryptographic assumptions can be weakened without compromising the protocol, remains a topic of ongoing work.

5.4.5 Improvements of CryptoVerif

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

- Improvements in the proof strategy should allow us to fully automate the proof in many more cases, in particular for public-key protocols. The prover should automatically distinguish cases in which the public key belongs to a honest principal or to the adversary.
- CryptoVerif is sometimes sensitive to the ordering of instructions, although the semantics of the game does not depend on this ordering. This problem could be solved by automatically moving $\mathsf{let}\ x = \ldots$ and $\mathsf{new}\ x$ instructions under tests (duplicating them if necessary); this transformation would allow CryptoVerif to distinguish cases depending on which branch assigns x. (This transformation is currently performed only for new.)
- An additional game transformation would be helpful in order to prove some

secrecy properties, in particular for key usability: tests $\mathsf{if}\ b\ \mathsf{then}\ P\ \mathsf{else}\ P'$ should be transformed into P when P and P' make indistinguishable actions, which would allow us to prove the secrecy of b. A first step would be to perform this transformation when P and P' are equal up to renaming of variables.

Chapter 6

Conclusions and Future Work

In chapter 3, we describe our discovery of a man-in-the-middle attack against PKINIT [57], the public key extension to the popular Kerberos 5 authentication protocol [70]. The attack was found on PKINIT-25, but applies to previous versions as well. We found this attack as part of an ongoing formal analysis of Kerberos, which has previously yielded proofs of security for the core Kerberos 5 protocol [28, 30] and its use for cross-realm authentication [39]. We have used formal methods approaches to prove that, at an abstract level, several possible defenses against our attack restore security properties of Kerberos 5 that are violated in PKINIT (as shown by the attack); we also proved some security properties that do hold even for the vulnerable form of PKINIT. The fixes we analyzed include the one proposed by the IETF Kerberos Working Group, which included it in the specification of PKINIT starting with revision 27 [57]. Our attack was also addressed in a Mi-

crosoft security bulletin affecting several versions of Windows [66] and mentioned in a CERT advisory [34]. We are also in the process of extending our analysis to the Diffie-Hellman mode of PKINIT: our preliminary observations suggest that it is immune from the attack described in this paper, but we do not yet have definite results on other types of threats.

In chapter 4, we have exploited the Dolev-Yao style model of Backes, Pfitzmann, and Waidner [13, 14, 11]. This formalism is more detailed than MSR, and we obtain proofs for the Kerberos and its extension to public keys (PKINIT) (whereas in chapter 3 we only consider the public-key extension). Although our the results we present are restricted to the symbolic model, the BPW model automatically lifts the results to the computational level, assuming that all cryptography is implemented using provably secure primitives.

Concerning future work related to the BPW model, 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 in the near future, e.g., by exploiting recent ideas from [43]. Another potential improvement we plan to pursue in the near future is to augment

the BPW model with timestamps; this would in particular allow us to establish authentication properties that go beyond entity authentication [28, 30, 39, 37]. A further 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 [28, 30, 37].

In chapter 5, we have formalized and mechanically analyzed all three rounds of the Kerberos 5 protocol, both with and without its public-key extension PKINIT, using version 1.06pl3 of the CryptoVerif tool. This is the first mechanical security proof of an industrial protocol at the computational level. The success of CryptoVerif in proving security properties for Kerberos—and especially for PKINIT, the use of which makes Kerberos particularly complex—provides evidence of its utility for analyzing industrial protocols. This also extends other work on analyzing Kerberos to include mechanical analysis tools. In carrying out this work, we show that cryptographic key secrecy in the sense of indistinguishability of the exchanged key from a random key does not hold, however, we extended the idea of key usability to a new notion of strong key usability; this definition was helpful here, and we are interested in exploring its utility elsewhere.

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

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

Since the specifications of Kerberos and PKINIT [70, 57] are actually more complicated than our formalization, we would like utilize CryptoVerif on formalizations of basic and public-key Kerberos that are closer to the specifications, *e.g.*, by using a key derivation function for all symmetric keys.

Another area for future work is the mechanized analysis of PKINIT's Diffie-Hellman mode, which we did not study here. As noted in [21], the language of equivalences used by CryptoVerif will need to be extended in order to handle Diffie-Hellman key exchange, so this problem holds both theoretical and practical interest.

Bibliography

[1] Martín Abadi and Bruno Blanchet. Analyzing security protocols with secrecy types and logic programs. *Journal of the ACM*, 52(1), January 2005.

[2] Martín Abadi and Roger Needham. Prudent Engineering Practice for Cryptographic Protocols. *IEEE Trans. Software Eng.*, 22(1):6–15, January 1996.

[3] Martín Abadi and Phillip Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). In *1st IFIP International Conference on Theoretical Computer Science*, volume 1872 of *LNCS*. Springer, August 2000.

[4] Michel Abdalla, Pierre-Alain Fouque, and David Pointcheval. Password-Based Authenticated Key Exchange in the Three-Party Setting. *IEE Proc. Information Security*, 153(1), 2006.

[5] Matthias Anlauff, Dusko Pavlovic, Richard Waldinger, and Stephen Westfold. Proving authentication properties in the Protocol Derivation Assistant. In

Pierpaolo Degano, Ralph Küsters, and Luca Vigano, editors, *Proceedings of FCS-ARSPA 2006*. ACM, 2006. to appear.

[6] Alessandro Armando, David A. Basin, Yohan Boichut, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, Paul Hankes Drielsma, Pierre-Cyrille Héam, Olga Kouchnarenko, Jacopo Mantovani, Sebastian Mödersheim, David von Oheimb, Michaël Rusinowitch, Judson Santiago, Mathieu Turuani, Luca Viganò, and Laurent Vigneron. The AVISPA tool for the automated validation of internet security protocols and applications. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verfication, CAV*, volume 3576 of *LNCS*, pages 281–285. Springer, 2005.

[7] Michael Backes. A cryptographically sound Dolev-Yao style security proof of the Otway-Rees protocol. In *Proc. European Symposium on Research in Computer Security, ESORICS'04*, pages 89–108. Springer LNCS 3193, 2004.

[8] Michael Backes, Iliano Cervesato, Aaron D. Jaggard, Andre Scedrov, and Joe-Kai Tsay. Cryptographically Sound Security Proofs for Basic and Public-key Kerberos. In *Proc. ESORICS'06*, volume 4189 of *LNCS*, pages 362 – 383. Springer, September 2006.

[9] Michael Backes and Christian Jacobi. Cryptographically sound and machine-assisted verification of security protocols. In *Proc. 20th Annual Symposium*

on Theoretical Aspects of Computer Science STACS, pages 675–686. Springer LNCS 2607, 2003.

[10] Michael Backes and Birgit 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.

[11] Michael Backes and Birgit Pfitzmann. Symmetric encryption in a simulatable Dolev-Yao style cryptographic library. In *Proc. 17th IEEE Computer Security Foundations Workshop, CSFW–17*, pages 204–218, June 2004.

[12] Michael Backes and Birgit Pfitzmann. On the cryptographic key secrecy of the stregthened Yahalom protocol. In *Proceedings of 21st IFIP SEC'06*, volume 201, pages 233–245, 2006.

[13] Michael Backes, Birgit Pfitzmann, and Michael Waidner. A composable cryptographic library with nested operations (extended abstract). In *Proc. 10th ACM Conference on Computer and Communications Security, CCS'03*, pages 220–230. ACM, 2003.

[14] Michael Backes, Birgit Pfitzmann, and Michael Waidner. Symmetric authentication within a simulatable cryptographic library. In *Proc. ESORICS'03*, pages 271–290. Springer LNCS 2808, 2003.

[15] Michael Backes, Birgit Pfitzmann, and Michael Waidner. A universally composable cryptographic library. IACR Cryptology ePrint Archive, Report 2003/015, `http://eprint.iacr.org/`, January 2003.

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

[17] Giampaolo Bella and Lawrence C. Paulson. Kerberos Version IV: Inductive Analysis of the Secrecy Goals. In *Proc. ESORICS'98*, volume 1485 of *LNCS*, pages 361–375. Springer, 1998.

[18] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In *CRYPTO'96*, volume 1109 of *LNCS*. Springer, 1996.

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

[20] Steven M. Bellovin and Michael Merritt. Limitations of the Kerberos Authentication System. In *USENIX Conference Proceedings*, 1991.

[21] Bruno Blanchet. A Computationally Sound Mechanized Prover for Security Protocols. In *IEEE Symposium on Security and Privacy*, pages 140–154.

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

[23] Bruno Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *Proc. CSFW - 14*, pages 82–96. IEEE Computer Society, June 2001.

[24] Bruno Blanchet. From Secrecy to Authenticity in Security Protocols. In Manuel Hermenegildo and Germán Puebla, editors, *9th International Static Analysis Symposium, SAS'02*, volume 2477 of *LNCS*, pages 342–359. Springer Verlag, September 2002.

[25] Bruno Blanchet. Computationally sound mechanized proofs of correspondence assertions. In *20th IEEE Computer Security Foundations Symposium, CSF'07*, July 2007.

[26] Bruno Blanchet and David Pointcheval. Automated Security Proofs with Sequences of Games. In *CRYPTO 2006*, volume 4117 of *LNCS*. Springer, August 2006.

[27] Alexandra Boldyreva and Virendra Kumar. Provable-security analysis of authenticated encryption in Kerberos. In *IEEE Symposium on Security and Privacy*, 2007.

[28] Frederick Butler, Iliano Cervesato, Aaron D. Jaggard, and Andre Scedrov. An Analysis of Some Properties of Kerberos 5 Using MSR. In *CSFW'02*, 2002.

[29] Frederick Butler, Iliano Cervesato, Aaron D. Jaggard, and Andre Scedrov. Confidentiality and Authentication in Kerberos 5. Technical Report MS-CIS-04-04, UPenn, 2004.

[30] Frederick Butler, Iliano Cervesato, Aaron D. Jaggard, Andre Scedrov, and Christopher Walstad. Formal Analysis of Kerberos 5. *Theoretical Computer Science*, 367(1–2), 2006.

[31] Cable Television Laboratories, Inc. PacketCable Security Specification, 2004. Technical document PKT-SP-SEC-I11-040730.

[32] Ran Canetti and Jonathan Herzog. Universally composable symbolic analysis of mutual authentication and key exchange protocols. In *Theory of Cryptography Conference, TCC'06*, volume 3876 of *LNCS*. Springer, March 2006.

[33] Ran Canetti and Hugo Krawczyk. Security Analysis of IKE's Signature-Based Key-Exchange Protocol. In *CRYPTO'02*, volume 2442 of *LNCS*. Springer, 2002.

[34] CERT. Vulnerability Note 477341. `http://www.kb.cert.org/vuls/id/477341`, 2005.

[35] Iliano Cervesato. A Specification Language for Crypto-Protocols based on Multiset Rewriting, Dependent Types and Subsorting. In G. Delzanno, S. Etalle, and M. Gabbrielli, editors, *Workshop on Specification, Analysis and Validation for Emerging Technologies — SAVE'01*, pages 1–22, Paphos, Cyprus, 2001.

[36] Iliano Cervesato. Typed MSR: Syntax and Examples. In *MMM'01*, volume 2052 of *LNCS*. Springer, 2001.

[37] Iliano Cervesato, Aaron D. Jaggard, Andre Scedrov, Joe-Kai Tsay, and Chris Walstad. Breaking and fixing public-key Kerberos. *Information and Computation*, FCS-ARSPA'06 Special Issue:402 – 424, 2008.

[38] Iliano Cervesato, Aaron D. Jaggard, Andre Scedrov, Joe-Kai Tsay, and Christopher Walstad. Breaking and Fixing Public-key Kerberos. In *ASIAN'06*, volume 4435 of *LNCS*, pages 167 – 181. Springer, 2008.

[39] Iliano Cervesato, Aaron D. Jaggard, Andre Scedrov, and Christopher Walstad. Specifying Kerberos 5 Cross-Realm Authentication. In *Proc. Workshop on Issues in the Theory of Security, WITS'05*. ACM Digital Library, 2005.

[40] John Clark and Jeremy Jacob. On the Security of Recent Protocols. *Information Processing Letters*, 56(3):151–155, 1995.

[41] Véronique Cortier and Bogdan Warinschi. Computationally sound, automated proofs for security protocols. In *Proc. 14th European Symposium on Programming, ESOP'05*, volume 3444 of *LNCS*, April 2005.

[42] Cas J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. Ph.D. dissertation, Eindhoven University of Technology, 2006.

[43] Anupam Datta, Ante Derek, John Mitchell, and Bogdan Warinschi. Key exchange protocols: Security definition, proof method, and applications. In *CSFW-18*. IEEE Press, 2006.

[44] Anupam Datta, Ante Derek, John C. Mitchell, and Bogdan Warinschi. Computationally Sound Compositional Logic for Key Exchange Protocols. In *CSFW -18*, pages 321 – 334. IEEE Press, 2006.

[45] Jan De Clercq and Micky Balladelli. Windows 2000 authentication. `http://www.windowsitlibrary.com/Content/617/06/6.html`, 2001. Digital Press.

[46] Whitfield Diffie, Paul C. van Oorschot, and Michael J. Wiener. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography*, 2(2):107–125, 1992.

[47] Shaddin F. Doghmi, Joshua D. Guttman, and F. Javier Thayer. Searching for shapes in cryptographic protocols. In *Proc. 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, 2007.

[48] Danny Dolev and Andrew C. Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 2(29):198–208, 1983.

[49] Nancy A. Durgin, Patrick Lincoln, John C. Mitchell, and Andre Scedrov. Multiset Rewriting and the Complexity of Bounded Security Protocols. *Journal of Computer Security*, 12(2):247–311, 2004.

[50] Shafi Goldwasser and Silvio Micali. Probabilistic encryption and how to play mental poker keeping secret all partial information. In *Proc. of 14th Annual Symp. Theory of Computing*.

[51] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28:270–299, 1984.

[52] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A Digital Signature Scheme Secure Against Adaptive Chosen Message Attacks. *SIAM Journal on Computing*, 17:281–308, 1988.

[53] Dan Harkins and Dave Carrel. The Internet Key Exchange (IKE). `http://www.ietf.org/rfc/rfc2409`, November 1998.

[54] Changhua He and John C. Mitchell. Security Analysis and Improvements for IEEE 802.11i. In *Proc. Network and Distributed System Security, NDSS'05*, February 2005.

[55] Changhua He, Mukund Sundararajan, Anupam Datta, Ante Derek, and John C. Mitchell. A modular correctness proof of TLS and IEEE 802.11i. In *CCS'05*. ACM, November 2005.

[56] Tzonelih Hwang and Yung-Hsiang Chen. On the Security of SPLICE/AS — The Authentication System in WIDE Internet. *Information Processing Letters*, 53(2):91–101, 1995.

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

[58] Richard A. Kemmerer, Catherine Meadows, and Jon Millen. Three systems for cryptographic protocol analysis. *Journal of Cryptology*, 7, 1994.

[59] John Kohl and Clifford Neuman. The Kerberos Network Authentication Service (V5), September 1993. `http://www.ietf.org/rfc/rfc1510`.

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

[61] Patrick D. Lincoln, John C. Mitchell, Mark Mitchell, and Andre Scedrov. A probabilistic poly-time framework for protocol analysis. In *CCS-5*, November 1998.

[62] Patrick D. Lincoln, John C. Mitchell, Mark Mitchell, and Andre Scedrov. Probabilistic polynomial-time equivalence and security protocols. In *FM'99*, volume 1708 of *LNCS*. Springer, September 1999.

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

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

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

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

[67] John C. Mitchell, Ajith Ramanathan, Andre Scedrov, and Vanessa Teague. A Probabilistic Polynomial-Time Process Calculus for the Analysis of Cryptographic Protocols. *Theoretical Computer Science*, 353(1–3), 2006.

[68] John C. Mitchell, Vitaly Shmatikov, and Ulrich Stern. Finite-State Analysis of SSL 3.0. In 7$^{\text{th}}$ *USENIX Security Symposium*, pages 201–216, 1998.

[69] Roger Needham and Michael Schroeder. Using Encryption for Authentication in Large Networks of Computers. *Comm. ACM*, 21(12):993–999, 1978.

[70] Clifford Neuman, Tom Yu, Sam Hartman, and Kenneth Raeburn. The Kerberos Network Authentication Service (V5), July 2005. `http://www.ietf.org/rfc/rfc4120`.

[71] Birgit Pfitzmann and Michael Waidner. A model for asynchronous reactive systems and its application to secure message transmission. In *Proc. S&P*, pages 184–200, 2001.

[72] Kenneth Raeburn. Encryption and Checksum Specifications for Kerberos 5. `http://www.ietf.org/rfc/rfc3961.txt`, February 2005.

[73] Arnab Roy, Anupam Datta, Ante Derek, and John C. Mitchell. Inductive proofs of computational secrecy. In *ESORICS 2007*, volume 4734 of *LNCS*. Springer, September 2007.

[74] Claude Shannon. Communication theory of secrecy systems. *Bell System Technical Journal*, (28):656 – 715, 1949.

[75] Christoph Sprenger, Michael Backes, David Basin, Birgit Pfitzmann, and Michael Waidner. Cryptographically Sound Theorem Proving. In *Proc. CSFW-18*. IEEE Computer Society, July 2006.

[76] Mario Strasser and Andreas Steffen. Kerberos PKINIT Implementation for Unix Clients. Technical report, Zurich University of Applied Sciences Winterthur, 2002.

[77] Michael Thomas and Jan Vilhuber. Kerberized Internet Negotiation of Keys (KINK), December 2003. `http://ietfreport.isoc.org/all-ids/draft-ietf-kink-kink-06.txt`.

[78] Luca Viganò. Automated security protocol analysis with the AVISPA tool. *Electrical Notes in Theoretical Computer Science*, 155:61–86, 2006.

[79] Tom Yu, Sam Hartman, and Kenneth Raeburn. The Perils of Unauthenticated Encryption: Kerberos Version 4. In *NDSS'04*, 2004.

.1 Additional MSR Roles for Later Exchanges

Here we recall the MSR roles for the TG and CS exchanges. These are the same as for basic Kerberos because PKINIT only modifies the AS exchange. Figure 1 shows the client's role in the TG exchange. The memory predicate $\mathsf{Auth}_C(X, T, AK)$ comes from the AS exchange (formalized in Figure 3.7 above).

$$
\begin{array}{l}
\forall C : \mathsf{client} \\
\left[\begin{array}{l}
\forall X : \mathsf{msg} \quad \forall T : \mathsf{TGS} \quad \forall S : \mathsf{server} \quad \forall AK : \mathsf{shK}\ C\ T \quad \forall t_C : \mathsf{time} \quad \forall S : \mathsf{server} \\
\qquad\qquad\qquad\qquad\qquad\qquad \exists n_3 : \mathsf{nonce} \\
\qquad \mathsf{Auth}_C(X, T, AK) \quad \stackrel{\iota_{3.1}}{\Longrightarrow} \quad \mathsf{N}(X, \{C, t_C\}_{AK}, C, S, n_3), \\
\qquad\qquad\qquad\qquad\qquad\qquad MemTGE(C, T, AK, n_3), \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{Auth}_C(X, T, AK) \\
\mathrm{IF} \quad \mathsf{clock}_C(t_C) \\
\\
\forall Y : \mathsf{msg} \quad \forall S : \mathsf{server} \quad \forall SK : \mathsf{shK}\ C\ S \quad \forall n_3 : \mathsf{nonce} \quad \forall t_T : \mathsf{time} \\
\quad \mathsf{N}(C, Y, \{SK, n_3, t_T, S\}_{AK}), \quad \stackrel{\iota_{3.2}}{\Longrightarrow} \quad \mathsf{Auth}_C(Y, S, SK) \\
\quad MemTGE(C, T, AK, n_3)
\end{array}\right]
\end{array}
$$

Figure 1: The client's role in TG exchange.

Figure 2 shows the TGS's role in the TG exchange. Figures 3 and 4 show the client and server roles, respectively, for the CS exchange.

$\forall T : \mathsf{TGS}$

$$\left[\begin{array}{l} \forall C : \mathsf{client} \quad \forall S : \mathsf{server} \quad \forall AK : \mathsf{shK}\ C\ T \quad \forall k_T : \mathsf{dbK}\ T \quad \forall k_S : \mathsf{dbK}\ S \\ \forall n_3 : \mathsf{nonce} \quad \forall t_T, t_C, t : \mathsf{time}. \\ \\ \mathsf{N}(\{AK, C, t\}_{k_T}, \{C, t_C\}_{AK}, \quad \stackrel{\iota_{4.1}}{\Longrightarrow} \quad \exists SK : \mathsf{shK}\ C\ S \\ \quad C, S, n_3) \qquad\qquad\qquad\qquad\qquad\quad \mathsf{N}(C, \{SK, C, t_T\}_{k_S}, \{SK, n_3, t_T, S\}_{AK}), \\ \\ \text{IF} \quad \mathsf{Valid}_T(C, S, t_C),\ \mathsf{clock}_T(t_T) \end{array}\right]$$

Figure 2: The TGS's role in the TG exchange.

$\forall C : \mathsf{client}$

$$\left[\begin{array}{l} \forall S : \mathsf{server} \quad \forall SK : \mathsf{shK}\ C\ S \quad \forall t'_C : \mathsf{time} \quad \forall Y : \mathsf{msg}. \\ \\ \mathsf{Auth}_C(Y, S, SK) \quad \stackrel{\iota_{5.1}}{\Longrightarrow} \quad \mathsf{N}(Y, \{C, t'_C\}_{SK}), \\ \qquad\qquad\qquad\qquad\quad \mathsf{Auth}_C(Y, S, SK) \\ \\ \text{IF} \quad \mathsf{clock}_C(t'_C) \end{array}\right]$$

Figure 3: The client's role in the CS exchange.

$\forall S : \mathsf{server}$

$$\left[\begin{array}{l} \forall C : \mathsf{client} \quad \forall T : \mathsf{TGS} \quad \forall SK : \mathsf{shK}\ C\ S \quad \forall t'_C, t_T : \mathsf{time} \quad \forall k_S : \mathsf{dbK}\ S \\ \\ \mathsf{N}(\{SK, C, t_T\}_{k_S}, \{C, t'_C\}_{SK}) \quad \stackrel{\iota_{6.1}}{\Longrightarrow} \quad \mathsf{N}(\{t'_c\}_{SK}), \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad Mem_S(C, SK, t'_C) \\ \\ \text{IF} \quad \mathsf{Valid}_S(C, t'_C) \end{array}\right]$$

Figure 4: The server's role in the CS exchange.

.2 Additional Algorithms of Kerberos in BPW

This appendix collects the algorithms omitted from the main body of this paper. The algorithms for Public-key Kerberos are in Figures 4.2, 4.3, 4.4–4.7, and Figures 5–14. The algorithms for Kerberos 5 are in Figures 11–20. Note that the algorithms for the TGS T and a server S (i.e., Algorithms 4 and 5 in Figures 11–14) are identical for Public-key Kerberos and Kerberos 5.

.3 Additional Proofs for Kerberos in BPW

.3.1 Conventions

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

Convention 1. Let $Kerb \in \{PK, K5\}$. 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 $in_w!$ and receives $\downarrow$ at port $out_w?$ as the immediate answer from $\mathrm{TH}_{\mathcal{H}}$, then M_w^{Kerb} aborts the execution of the current algorithm, except if the command was of the form list_proj or send_i.

.3.2 Auxiliary Properties for Public-key Kerberos

Next we will consider the auxiliary properties from Section 4.2.1 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:

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

Figure 5: Algorithm 3 of Public-key Kerberos, part 1: Behavior of user in after initialization

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 $i = 1, 2, 3$ $\{x_1 \approx k_e, x_2 \approx k_a, x_3 \approx ck\}$

Figure 6: Algorithm 3 of Public-key Kerberos, part 2: Behavior of user in after initialization

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. $b \leftarrow \mathsf{auth\_test}(x_3^{hnd}, x_2^{hnd}, \tilde{m}^{hnd})$ $\{x_3 \approx ck = H_k(\tilde{m})\}$

Figure 7: Algorithm 3 of Public-key Kerberos, part 3: Behavior of user in after initialization

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 $\mathrm{KA\_out}_u$!

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. $l_2^{hnd} \leftarrow \mathsf{sym\_decrypt}(AK^{hnd}, d_3^{hnd})$ $\{l_2 \approx SK, n_3, t_T, S\}$

Figure 8: Algorithm 3 of Public-key Kerberos, part 4: Behavior of user in after initialization

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}

Figure 9: Algorithm 3 of Public-key Kerberos, part 5: Behavior of user in after initialization

64. **if** $(\nexists!(S, SK^{hnd}) \in Session\_KeysS_u$: 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 KA_out$_u$!

Figure 10: Algorithm 3 of Public-key Kerberos, part 6: Behavior of user after initialization

Input: (v, T, i, m^{hnd}) at 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. $Nonce4_T := Nonce4_T \cup \{(x_5^{hnd}, v)\}$

Figure 11: Algorithm 4, part 1: Behavior of TGS

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. $\mathsf{send\_i}(v, m_4^{hnd})$

Figure 12: Algorithm 4, part 2: Behavior of TGS

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

1. $m_{5.i}^{hnd} \leftarrow \mathsf{list\_proj}(m^{hnd}, i)$ for $i = 1, 2$ $\{m_{5.1} \approx ST,\ m_{5.2} \approx \{C, t'_C\}_{SK}\}$
2. $x^{hnd} \leftarrow \mathsf{sym\_decrypt}(skse_{TS}^{hnd}, m_{5.1}^{hnd})$
3. $x_i^{hnd} \leftarrow \mathsf{list\_proj}(x^{hnd}, i)$ for $i = 1, 2$ $\{x_1 \approx SK,\ x_2 \approx C\}$
4. $x_2 \leftarrow \mathsf{retrieve}(x_2^{hnd})$
5. $type_1 \leftarrow \mathsf{get\_type}(x_1^{hnd})$
6. **if** $(type_1 \neq \text{skse}) \vee (x_2 \neq v)$ **then**
7. Abort
8. **end if**
9. $y^{hnd} \leftarrow \mathsf{sym\_decrypt}(x_1^{hnd}, m_{5.2}^{hnd})$ $\{y \approx C, t'_C\}$
10. $y_i^{hnd} \leftarrow \mathsf{list\_proj}(y^{hnd}, i)$ for $i = 1, 2$ $\{y_1 \approx C,\ y_2 \approx t'_C\}$
11. $y_1 \leftarrow \mathsf{retrieve}(y_1^{hnd})$
12. **if** $(y_1 \neq v)$ **then**

Figure 13: Algorithm 5, part 1: Behavior of server

13. Abort

14. **end if**

15. $m_6^{hnd} \leftarrow \mathsf{sym\_encrypt}(x_1^{hnd}, y_2^{hnd})$ $\{m_6 \approx \{t'_C\}_{SK}\}$

16. $\mathsf{send\_i}(S, m_6^{hnd})$

17. output $(\mathsf{ok}, \mathsf{PK}, v, x_1^{hnd})$ at $\mathrm{KA\_out}_S$!

Figure 14: Algorithm 5, part 2: Behavior of server

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

Lemma .3.1 (Correct Nonce Owner)**.** *For all $u \in \mathcal{H}$, and $(x^{hnd}, \ldots) \in Nonce_u$ or $(x^{hnd}, \ldots) \in Nonce2_u$, it holds $D[hnd_u = x^{hnd}] \neq\downarrow$ 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 $\mathrm{in}_u?$ of $\mathrm{TH}_{\mathcal{H}}$. 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 [7] 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

A) Input:$(\mathsf{new\_prot}, \mathsf{K5}, K, T)$ at KA_in_u? .

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

Figure 15: Algorithm 1 of Kerberos 5, part 1: Evaluation of inputs from the user (starting the AS exchange).

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 .3.2 (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. 4.4–4.7))

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

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

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

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

Figure 16: Algorithm 1 of Kerberos 5: Evaluation of inputs from the user (starting the TG exchange).

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

1. $x_i^{hnd} \leftarrow \mathsf{list\_proj}(m^{hnd}, i)$ for $i = 1, 2, 3$ $\{x_1 \approx C, x_2 \approx T, x_3 \approx n_1\}$
2. $type_1 \leftarrow \mathsf{get\_type}(x_3^{hnd})$
3. $x_i \leftarrow \mathsf{retrieve}(x_i^{hnd})$ for $i = 1, 2$
4. **if** $(type_1 \neq \text{nonce}) \vee ((x_3, .) \in Nonce3_K) \vee (x_1 \neq v) \vee (x_2 \neq T)$ **then**
5. Abort
6. **end if**
7. $v^{hnd} \leftarrow \mathsf{store}(v)$
8. $Nonce3_K := Nonce3_K \cup \{(x_3^{hnd}, v)\}$
9. $AK^{hnd} \leftarrow \mathsf{gen\_symenc\_key}()$

Figure 17: Algorithm 2 of Kerberos 5, part 1: Behavior of the KAS

10. $z_1^{hnd} \leftarrow \mathsf{list}(AK^{hnd}, v^{hnd}, t_K^{hnd})$ $\{z_1 \approx AK, C, t_K\}$

11. $TGT^{hnd} \leftarrow \mathsf{sym\_encrypt}(skse_{K,x_2}^{hnd}, z_1^{hnd})$ $\{TGT \approx \{AK, C, t_K\}_{k_T}\}$

12. $z_2^{hnd} \leftarrow \mathsf{list}(AK^{hnd}, x_3^{hnd}, t_K^{hnd}, x_2^{hnd})$ $\{z_2 \approx AK, n_1, t_K, T\}$

13. $m_{23} \leftarrow \mathsf{sym\_encrypt}(k_v^{hnd}, z_2^{hnd})$ $\{m_{23} \approx \{AK, n_1, t_K, T\}_{k_C}\}$

14. $m_2^{hnd} \leftarrow \mathsf{list}(v^{hnd}, TGT^{hnd}, m_{23}^{hnd})$ $\{m_2 \approx C, TGT, \{AK, n_1, t_K, T\}_{k_C}\}$

15. $\mathsf{send\_i}(v, m_2^{hnd})$

Figure 18: Algorithm 2 of Kerberos 5, part 2: Behavior of the KAS

(Fig. 4.4–4.7))

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

c) If $D[j]$ was created by M_T^{PK} in step 4.18 then (with the notation of Algorithm 4 in Fig. 11,12)

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

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

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

1. **if** $v = K$ **then** {AS_REP is input}
2. $c_i^{hnd} \leftarrow \mathsf{list\_proj}(m^{hnd}, i)$ for $i = 1, 2, 3$ $\{c_1 \approx C, c_2 \approx TGT, c_3 \approx \{AK, n_1, t_K, T\}_{k_C}\ \}$
3. $c_1 \leftarrow \mathsf{retrieve}(c_1^{hnd})$
4. **if** $(c_1 \neq u)$ **then**
5. Abort
6. **end if**
7. $type_i \leftarrow \mathsf{get\_type}(c_i^{hnd})$ for $i = 2, 3$
8. **if** $(type_2 \neq \text{skse}) \vee (type_3 \neq \text{auth})$ **then**
9. Abort
10. **end if**
11. $l_3^{hnd} \leftarrow \mathsf{sym\_decrypt}(skse_{uK}^{hnd}, c_3^{hnd})$ $\{l_3 \approx AK, n_1, t_K, T\}$
12. $y_i^{hnd} \leftarrow \mathsf{list\_proj}(l_3^{hnd}), i$ for $i = 1, 2, 4$ $\{y_1 \approx AK, y_2 \approx n_1, y_4 \approx T\}$

Figure 19: Algorithm 3 of Kerberos 5, part 1: Behavior of user after initialization

13. $type_4 \leftarrow \mathsf{get\_type}(y_1^{hnd})$

14. $type_5 \leftarrow \mathsf{get\_type}(y_2^{hnd})$

15. $y_4 \leftarrow \mathsf{retrieve}(y_4^{hnd})$

16. **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**

17. Abort

18. **end if**

19. $TGTicket_u := TGTicket_u \cup \{(c_2^{hnd}, y_1^{hnd}, T)\}$

20. output $(\mathsf{ok}, \mathsf{KAS\_exchange\ K5}, K, T, y_1^{hnd}, c_2^{hnd})$ at KA_out_u!

21. **else if** $v = T$ **then** {TGS_REP is input}

22. $d_i^{hnd} \leftarrow \mathsf{list\_proj}(m^{hnd}, i)$ for $i = 1, 2, 3$ $\{ \, d_1 \approx C, \, d_2 \approx ST, \, d_3 \approx \{SK, n_3, t_T, S\}_{AK} \, \}$

23. $d_1 \leftarrow \mathsf{retrieve}(d_1^{hnd})$

24. **if** $(d_1 \neq u)$
$\vee \, (\nexists! \, (., AK^{hnd}, T) \in TGTicket_u) : \mathsf{sym\_decrypt}(AK^{hnd}, d_3^{hnd}) \neq \downarrow)$ **then**

Figure 20: Algorithm 3 of Kerberos 5, part 2: Behavior of user after initialization

25. Abort

26. **end if**

27. $l_2^{hnd} \leftarrow \mathsf{sym\_decrypt}(AK^{hnd}, d_3^{hnd})$ $\{l_2 \approx SK, n_3, t_T, S\}$

28. $x_{2.i}^{hnd} \leftarrow \mathsf{list\_proj}(l_2^{hnd}, i)$ for $i = 1, 2, 4$ $\{x_{2.1} \approx SKey,\ x_{2.2} \approx n_3,\ x_{2.4} \approx S\}$

29. $type_5 \leftarrow \mathsf{get\_type}(x_{2.1}^{hnd})$

30. $type_6 \leftarrow \mathsf{get\_type}(x_{2.2}^{hnd})$

31. $S \leftarrow retrieve(x_{2.4}^{hnd})$

32. **if** $(type_5 \neq \text{skse}) \vee (type_6 \neq \text{nonce}) \vee ((x_{2.2}^{hnd}, T, S) \notin Nonce2_u)$ **then**

33. Abort

34. **end if**

35. $x_5^{hnd} \leftarrow \mathsf{list}(u^{hnd}, t_u'^{hnd})$ $\{x_5 \approx C, t'_C\}$

36. $m_{5.2}^{hnd} \leftarrow \mathsf{sym\_encrypt}(x_{2.1}^{hnd}, x_5^{hnd})$ $\{m_{5.2} \approx \{C, t'_u\}_{SK}\}$

Figure 21: Algorithm 3 of Kerberos 5, part 3: Behavior of user after initialization

37. $m_5^{hnd} \leftarrow \mathsf{list}(d_2^{hnd}, m_{5.2}^{hnd})$ $\{m_5 \approx ST, \{C, t'_u\}_{SK}\}$

38. $Session\_KeysS_u := Session\_KeysS_u \cup \{(S, x_{2.1}^{hnd})\}$

39. $\mathsf{send\_i}(S, m_5^{hnd})$

40. **else if** $v = S \in \{S_1, ..., S_l\}$ **then** {AP_REP is input}

41. **if** $(\nexists!(S, SK^{hnd}) \in Session\_KeysS_u$: sym_decrypt$(SK^{hnd}, m^{hnd}) \neq\downarrow)$ **then**

42. Abort

43. **end if**

44. $l_3^{hnd} \leftarrow \mathsf{sym\_decrypt}(SK^{hnd}, m^{hnd})$ $\{m \approx \{t'_C\}_{SK}\}$

45. $x_{3.1}^{hnd} \leftarrow \mathsf{list\_proj}(l_3^{hnd}, 1)$ $\{x_{3.1} \approx t'_C\}$

46. $x_{3.1} \leftarrow retrieve(x_{3.1}^{hnd})$

47. **if** $x_{3.1} = u$ **then**

48. Abort

49. **end if**

50. output $(\mathsf{ok}, \mathsf{K5}, S, SK^{hnd})$ at KA_out$_u$!

Figure 22: Algorithm 3 of Kerberos 5, part 4: Behavior of user after initialization

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 decrypt, list_proj, 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 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 $j \leq size$, $D[j].type =$ 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 sym_encrypt taking as arguments a handle to k_e and a handle to the 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 $\mathrm{skse}_{K,T}$ 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 sym_encrypt taking as arguments a handle to $\mathrm{skse}_{K,T}$ and a handle to the list z_3,

where $z_3.arg[1] = D[j].ind$). By construction of Algorithm 4 (Fig. 11,12) 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 (Fig. 4.2, 4.3 and 5–10) 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 $j \leq size$, $D[j].type =$ 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 ciphertext 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 $t' > t$. □

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 .3.3 (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 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 & 2.35 does $\mathrm{M}_K^{\mathrm{PK}}$ then create 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 .3.4 (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. 11,12), $\mathrm{M}_T^{\mathrm{PK}}$ 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 out_K?. In step 2.38 M_K^{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 $\text{TH}_{\mathcal{H}}$ earlier, when receiving the command gen_sym_key from M_K^{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 M_u^{PK} uses this key for encryption of a list containing u's name, that is to say in step 1B.5, *i.e.*, M_u^{PK} generated $D[l_2]$ in step 1B.5. It is obvious that this encryption happened after M_K^{PK} created $D[l_1]$, since M_K^{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 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 .3.5 (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 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 .3.3 for honest user u (*e.g.*, $D[m'.arg[1]]$ has the structure of $D[l_1]$ from Lemma .3.3 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

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 .3.4. 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 M_T^{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 .3.6 (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^{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. 13,14), $\mathrm{M}_S^{\mathrm{PK}}$ 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 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. □

.3.3 Proof of Theorem 4.2.3, Public-key Kerberos part

Now we present the proof of Thm. 4.2.3 regarding public-key Kerberos:

of Thm. 4.2.3. First we prove the Secrecy Property: Say there was an output $(\text{ok}, \text{PK}, S, SK^{hnd})$ at KA_out_u!. Examining Algorithm 3 (Fig. 5–10) 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 $Session\_KeysS_u$ (step 3.61), M_u^{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 M_u^{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.*, M_u^{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 .3.5 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 $(\text{ok}, \text{PK}, u, SK^{hnd})$ at KA_out_S!. This only occurs if there was an input (u, S, i, m^{hnd}) at 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 M_S^{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, 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 M_S^{PK} without abort before the output (ok, 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 .3.6. 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 .3.6. 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 .3.6 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 $KA\_out_S!$ 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 $out_S?$ 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 $out_S?$ and the output (ok, PK, v, x_1^{hnd}) at $KA\_out_S!$, 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 .3.6. And since v is honest, Lemma .3.6 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 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 .3.5 and satisfies the hypotheses of Lemma .3.5. Therefore, by Lemma .3.5, $D[d_3]$ 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 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 .3.4, 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 .3.4, 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 .3.4 are satisfied and so one gets, by Lemma .3.4, 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 $\mathrm{PK}, T, S, AK^{hnd}$) 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 3.42 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 $(\mathrm{ok}, \mathrm{KAS\_exchange\ PK}, K, T, (.,.,., AK^{hnd}))$ at $\mathrm{KA\_out}_v$!. Since the execution of Algorithm 3 did not produce an error, one can use Lemma .3.3 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 $(\mathrm{new\_prot}, \mathrm{PK}, K, T))$ at $\mathrm{KA\_in}_v$! at a time $t_1 < t_2$. ii) Now say that there was on output $(\mathrm{ok}, \mathrm{PK}, S, SK^{hnd})$ at $\mathrm{KA\_out}_u$! at time t_2. By construction of Algorithm 3, this only happens after u received an input (S, u, i, m^{hnd}) at 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 & 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 $\mathrm{out}_u?$ 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 .3.5 (where $SK = D[d_3.arg[1]].arg[1]$ and $S = D[d_3.arg[1]].arg[4]$). Using Lemma .3.5 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 $\mathrm{out}_S?$, 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 $(\mathrm{ok}, \mathrm{PK}, u, SK^{hnd})$ at $\mathrm{KA\_out}_S!$ at some time $t_1 < t_2$. □

.3.4 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 15,16, 17, 18, and 19 –22. 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 .3.7 (Correct Nonce Owner)**.** *For all $u \in \mathcal{H}$, and $(x^{hnd}, \ldots) \in Nonce_u$ or $(x^{hnd}, \ldots) \in Nonce2_u$, it holds $D[hnd_u = x^{hnd}] \neq\downarrow$ 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 $\mathrm{in}_u?$ 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 [7] 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 .3.8 (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. 17,18))

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

b) If $D[j]$ was created by M_T^{K5} in step 4.18 then (with the notation of Algorithm 4 (Fig. 11,12))

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

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 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 $\mathrm{skse}_{K,T}$ 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 $\text{skse}_{K,T}$ and a handle to the list z_1 where $z_1.arg[1] = D[j].ind$). By construction of Algorithm 4 (Fig. 11,12) and since T is honest, one sees that M_T^{K5} never sends any message from which a handle to $D[j]$ for time $t' > t$. Also, by construction of Algorithms 1 and 3 (Fig. 15,16, 19– 22) 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 $j \leq size$, $D[j].type = \text{skse}$ such that $D[j]$ was created by M_T^{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 ciphertext 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

$\mathrm{M}_S^{\mathrm{K5}}$ uses skse_{TS}^{hnd} for decryption (in step 5.2). But, by construction, neither $\mathrm{M}_S^{\mathrm{K5}}$ nor $\mathrm{M}_v^{\mathrm{K5}}$ 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 .3.9 (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 $\mathrm{M}_K^{\mathrm{K5}}$ and

$\mathrm{M}_u^{\mathrm{K5}}$ have handles to k_u. Since, by construction, $\mathrm{M}_u^{\mathrm{K5}}$ does not use this key for encryption, $\mathrm{M}_K^{\mathrm{K5}}$ must have used it for encryption. This occurs only in step 2.13 after there was an input (v, K, i, m^{hnd}) at 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, $\mathrm{M}_u^{\mathrm{K5}}$ must send it first. This happens in step 2.15, where $\mathrm{M}_u^{\mathrm{K5}}$ 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 .3.10 (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 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 $\mathrm{M}_K^{\mathrm{K5}}$ and $\mathrm{M}_T^{\mathrm{K5}}$ have handles to the long-term key $skse_{KT}$. Since by construction of Algorithm 4 (Fig. 11,12), $\mathrm{M}_T^{\mathrm{K5}}$ does not use this key for encryption, $\mathrm{M}_K^{\mathrm{K5}}$ 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 out_K?. In step 2.11 $\mathrm{M}_K^{\mathrm{K5}}$ 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 $\mathrm{TH}_{\mathcal{H}}$ earlier, after receiving the command gen_sym_key from $\mathrm{M}_K^{\mathrm{K5}}$ 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 $\mathrm{M}_u^{\mathrm{K5}}$ uses this key for encryption of a list containing u's name, that is to say in step 1B.5, *i.e.*, $\mathrm{M}_u^{\mathrm{K5}}$ generated $D[l_2]$ in step 1B.5. It is obvious that this encryption

happened after $\mathrm{M}_K^{\mathrm{K5}}$ created $D[l_1]$, since $\mathrm{M}_K^{\mathrm{K5}}$ 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 .3.11 (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 .3.5. □

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 .3.12 (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, $\{\approx k_S\}$

$p_2 := D[l_1].arg[1]$ *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 .3.6. □

.3.5 Proof of Theorem 4.2.3, Basic Kerberos part

Now we present the proof of Thm. 4.2.3 regarding basic Kerberos:

of Thm. 4.2.3. First we prove the Secrecy Property: Say there was an output $(\text{ok}, \text{K5}, S, SK^{hnd})$ at KA_out_u!. Examining Algorithm 3 (Fig. 19–22) 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 $Session\_KeysS_u$ (step 3.38), M_u^{K5} obtained the handle SK^{hnd} in step 3.28 and steps 3.32 & 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 M_u^{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.*, M_u^{K5} has handles to d_3 and to the key AK. Steps 3.24 & 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 .3.11 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 $(\text{ok}, \text{K5}, u, SK^{hnd})$ at KA_out_S!. This only occurs if

there was an input (u, S, i, m^{hnd}) at $\text{out}_S?$ 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 M_S^{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, 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 M_S^{K5} without abort before the output $(\text{ok}, \text{K5}, u, k^{hnd})$. 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 .3.12. 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 .3.12. 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 .3.12 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 $\text{KA\_out}_S!$ 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 $\text{out}_S?$ 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 $\text{out}_S?$ and the output (ok, K5, v, x_1^{hnd}) at $\text{KA\_out}_S!$,

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 .3.6. And since v is honest, Lemma .3.12 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 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 .3.11 and satisfies the hypotheses of Lemma .3.11. 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 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 .3.10, 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 .3.10, 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 .3.10 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 $\mathrm{K5}, T, S, AK^{hnd}$) 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 & 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 .3.9 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 $\mathrm{KA\_out}_u$! at time t_2. By construction of Algorithm 3, this only happens after u received an input (S, u, i, m^{hnd}) at 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 & 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 .3.11 (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 $t_1 < t_2$. □

.4 Additional Cryptoverif Proof Details

Proof of Theorem 5.4.2. Basic Kerberos case: when the TGS process validates a received request in a TG exchange, it executes an event $partTC(hostC, m8, m9)$ that contains the name $hostC$ of the client contained in the request's TGT, the the ciphertext part of the TGT in $m8$, and the ciphertext part of the matching authenticator in $m9$. When the KAS process completes its participation in an AS exchange, it executes an event $fullKC(hostY, hostW, n', TGT', e4)$ as described in the proof of Theorem 5.4.1 in the basic Kerberos case. When the client process requests a service ticket from a TGS, it executes an event $partCT(hostX, TGT, e5)$ that contains the name $hostX$ of the TGS the client requests a service ticket from, the ciphertext part of the alleged TGT in TGT and the ciphertext part of the authenticator of C in $e5$ containing C's name and a timestamp encrypted under the authentication key (AK) which C received in the same AS reply as TGT. CryptoVerif 1.06pl3 can then automatically prove the query:

$\mathsf{event}(partTC(C, x, y)) \Rightarrow \mathsf{event}(partCT(T, x', y) \wedge \mathsf{event}(fullKC(C, T, n, x, y'))$.

Public-key Kerberos case: As the TG exchange in public-key Kerberos does not differ from the TG exchange in basic Kerberos V5, the events $partTC(hostC, m8, m9)$ and $partCT(hostX, TGT, e5)$ are just as the ones described above in the basic Kerberos case. Furthermore, the event $fullKC(hostY, hostW, n', e21, TGT', e24)$ is as described in the proof of Theorem 5.4.1 in the public-key Kerberos case. CryptoVerif 1.06pl3 can then prove the query:

$\mathsf{event}(partTC(C, x, y)) \Rightarrow \mathsf{event}(partCT(T, x', y)) \wedge \mathsf{event}(fullKC(C, T, n, w, x, y'))$.

We note that the proof for the public-key Kerberos case is again an interactive proof which requires the same commands as the corresponding case from Theorem 5.4.1. □

Proof of Theorem 5.4.3. Basic Kerberos case: when the client process completes its participation in an TG exchange, it executes an event $fullCT(hostT, hostS, n, ST, e5, m7)$ that contains the name $hostT$ of the TGS, the name $hostS$ of the server, the nonce n and the authenticator $e5$ from the client's request m_{req}. Furthermore, the event contains the reply from K in ST and $m7$, where $m7$ is the message component of the reply encrypted under the authentication key, and ST is assumed to be a service ticket. All ST, $e5$ and $m7$ do not contain the associated MACs. When the TGS process completes its participation in an TG exchange, it executes an event $fullTC(hostY, hostW, n', m8, m9, ST', e11)$ that contains the name $hostY$ of the client, the names $hostW$, n', $m8$ and $m9$ of the server, the nonce, the TGT, and the authenticator, respectively, which were all listed in the request m'_{req}. Furthermore, the event contains the service ticket ST' generated by the TGS containing the service key SK', and the message component $e11$ of the reply that is encrypted under the authentication key. Again, $m8$, $m9$, ST', and $e11$ only contain the ciphertext parts of the encrypt-then-MAC ciphertext pairs without the associated MACs. CryptoVerif 1.06pl3 can then automatically prove the query:

$\mathsf{inj\text{-}event}(fullCT(T, S, n, x, z, y)) \Rightarrow \mathsf{inj\text{-}event}(fullTC(C, S, n, x', z', v, y))$.

Public-key Kerberos case: As the TG exchange in public-key Kerberos does not differ from the TG exchange in basic Kerberos V5, the events $\mathit{fullCT}(\mathit{hostT}, \mathit{hostS}, n, ST, \mathit{e5}, \mathit{m7})$ and $\mathit{fullTC}(\mathit{hostY}, \mathit{hostW}, n', \mathit{m8}, \mathit{m9}, ST', \mathit{e11})$ are just as the ones described above in the basic Kerberos case. CryptoVerif 1.06pl2 can then prove the query:

$\mathsf{inj\text{-}event}(\mathit{fullCT}(T, S, n, x, z, y)) \Rightarrow \mathsf{inj\text{-}event}(\mathit{fullTC}(C, S, n, x', z', v, y))$.

We note that the proof for the public-key Kerberos case is again an interactive proof which requires the same commands as the corresponding case from Theorem 5.4.1. □

Proof of Theorem 5.4.5. Basic Kerberos case: when the client process completes its participation in a CS exchange, it executes an event $\mathit{fullCS}(\mathit{hostS}, \mathit{hostT}, \mathit{m7}, \mathit{e12}, \mathit{e13})$ that contains, in addition to the names that are also included in *partCS* (in the proof of Theorem 5.4.4), the name *e13* of the cipertext part of the response from *hostS*. When the server process completes its participation in an TG exchange, it executes an event $\mathit{fullSC}(\mathit{hostC}, \mathit{m14}, \mathit{m15}, \mathit{m16})$ that contains, in addition to the names that are also included in *partSC* (in the proof of Theorem 5.4.4), the ciphertext part *m16* of S's reply to *hostC*. CryptoVerif 1.06pl3 can then automatically prove the query: $\mathsf{event}(\mathit{fullCS}(S, T, z, y, w)) \Rightarrow \mathsf{event}(\mathit{fullSC}(C, x, y', w))$.

Public-key Kerberos case: As the CS exchange in public-key Kerberos does not differ from the CS exchange in basic Kerberos V5, the events $\mathit{fullCS}(\mathit{hostS}, \mathit{hostT}, \mathit{m7}, \mathit{e12}, \mathit{e13})$ and $\mathit{fullSC}(\mathit{hostC}, \mathit{m14}, \mathit{m15}, \mathit{m16})$ are just as the ones described

above in the basic Kerberos case. CryptoVerif 1.06pl2 can then prove the same query as above in basic Kerberos case using the same commands as in the public-key Kerberos case from Theorem 5.4.1. □

Proof of Theorem 5.4.14. Basic Kerberos case: when, *e.g.*. for $X = C$, the client process completes its participation in a CS exchange involving an honest TGS, it stores the authentication key pair (AK, mAK) in $keyCAK$ and $mkeyCAK$. From these key pairs one is drawn at random and passed to the encryption oracle and decryption oracle. For the boolean *b1* used by the left-right encryption oracle, we can, using CryptoVerif, prove the query: secret *b1*. This proof requires the user to inspect the last game, which CryptoVerif reaches upon the command `auto`, in order to verify that terms that are dependent on *b1* and which may help an adversary in guessing *b1* occur only in find branches that are never executed. A similar result holds for $X = K$ and $X = T$, where in the latter case the following commands are required before the manual inspection of the last game: `auto`, `SArename AK_37`, `simplify`, `auto`. The command `SArename AK_37` is used when the variable `AK_37` is defined several times in the game. It instructs CryptoVerif to rename each definition of this variable to a different name, which subsequently allows to distinguish cases depending on the program point at which the variable has been defined. The command `simplify` instructs CryptoVerif to simplify the game as much as possible.

Public-key Kerberos case: analogous to the basis Kerberos case, the secrecy of the bit $b1$ can be concluded by inspecting the last game that CryptoVerif reaches

after a sequence of commands. When $X = K$ or $X = T$, the required commands are just the ones from the public-key Kerberos case of Theorem 5.4.1. If $X = C$, the secrecy of $b1$ can be concluded after the following sequence of commands: `crypto sign rkCA`, `crypto sign rkCs`, `crypto penc rkC`, `crypto sign rkKs`, `auto`, `remove_assign binder AK_70`, `SArename AK_62`, `simplify`, `auto`. The command `remove_assign binder AK_70` instructs CryptoVerif to remove assignments on the variable `AK_70`, that is, each occurrence of this variable is replaced with its value. □

.5 Additional Roles in Process Calculus

Together with Figure 5.2 from Section 5.3.6, the Figures 23,24,25,26,27,29 in this appendix describe in more details from the formalization of Kerberos in the process calculus, as used to prove authentication properties for basic Kerberos. As noted before, the full CryptoVerif scripts are available at `http://www.cryptoverif.ens.fr/kerberos/`. The main process is detailed in Figure 23.

$Q_0 = start();$ new $rKc : keyseed;$ let $k_C = \text{kgen}(rKc)$ in

new $rKt : keyseed;$ let $k_T = \text{kgen}(rKt)$ in

new $rKs : keyseed;$ let $k_S = \text{kgen}(rkS)$ in

$\overline{c_{25}}\langle\rangle; (Q_C|Q_K|Q_T|Q_S|Q_{C_{key}}|Q_{T_{key}}|Q_{S_{key}})$

Figure 23: Main basic Kerberos process

$Q_{C2}[i_C] = !^{i_{C2} \leq N_2}\, c_{17}[i_C, i_{C2}](hostT' : tgs, hostS : server);$

if $hostT' = hostT$ then

new $n_3 : nonce$; new $t_C : timest$; new $r_1 : seed$;

let $e5 = \mathrm{enc}(\mathrm{pad}(C, t_C), AK, r_1)$ in event $partCT(hostT, m, e5)$;

$\overline{c_4[i_C, i_{C2}]}\langle m, e5, hostS, n3\rangle$;

$c5[i_C, i_{C2}](= C, m_6 : bitstring, m_7 : bitstring))$;

let $\mathrm{injbot}(\mathrm{concat3}(SK, = n_3, t_T, = hostS)) = \mathrm{dec}(m_7, AK)$ in

event $fullCT(hostT, hostS, n_3, m, e5, m_7)$;

$\overline{c_{19}[i_C, i_{C2}]}\langle acceptC2(hostT, hostS)\rangle; (Q_{C3}[i_C, i_{C2}])$.

Figure 24: CryptoVerif formalization of client's actions in TG exchange.

$Q_{C2}[i_C, i_{C2}] = !^{i_{C3} \leq N_4}\, c_{20}[i_C, i_{C2}, i_{C3}](hostS' : server);$

if $hostS' = hostS$ then

new $t'_C : timest$; new $r_2 : seed$;

let $e12 = \mathrm{enc}(\mathrm{pad}(C, t'_C), SK, r_2)$ in event $partCS(hostS, hostT, m_7, e12)$;

$\overline{c_6[i_C, i_{C2}, i_{C3}]}\langle m_6, e12\rangle$; $c_9[i_C, i_{C2}, i_{C3}](m_{13} : bitstring)$;

if $\mathrm{injbot}(\mathrm{padts}(t'_C)) = \mathrm{dec}(m_{13}, SK)$ then

event $fullCS(hostS, hostT, m_7, e12, m_{13})$;

$\overline{c_{10}[[i_C, i_{C2}, i_{C3}]}\langle acceptC3(hostS)\rangle$.

Figure 25: CryptoVerif formalization of client's actions in CS exchange.

$Q_K = !^{i_K \leq N}\, c_{14}[i_K](hostY : client, hostW : tgs, n : nonce);$

find $j_1 \leq N_2$ suchthat

defined$(Khost[j_1], Rkey[j_1]) \wedge (Khost[j_1] = hostY)$ then

find $j_2 \leq N_2$ suchthat

defined$(Lhost[j_2], Qkey[j_2]) \wedge (Lhost[j_2] = hostW)$ then

new $r_{AK} : keyseed$; let $AK = \mathrm{kgen}(r_{AK})$ in

new $t_K : timest$; new $r_3 : seed$;

let $e3 = \mathrm{enc}(\mathrm{concat2}(AK, t_K, hostY), Qkey[j_2], r_3)$ in

new $r_4 : seed$; let $e4 = \mathrm{enc}(\mathrm{concat1}(AK, n, t_K, hostW), Rkey[j_1], r_4)$ in

event $fullKC(hostY, hostW, n, e3, e4)$; $\overline{c_{15}[i_K]}\langle hostY, e3, e4\rangle$.

Figure 26: CryptoVerif formalization of KAS's actions.

.6 Additional Security Definitions for CryptoVerif

Together with Figure 5.3 in Section 5.4.2, the Figures 30, 31, 32 in this appendix describe additional cryptographic assumptions, that we made in Section 5.4.1, as pairs of indistinguishable random oracles in the process calculus, as required by CryptoVerif in order to apply these assumptions during game transformations.

$Q_T = !^{i_T \leq N}\, c_7[i_T](m_8 : bitstring, m_9 : bitstring, hostW : server, n' : nonce);$

let $\mathrm{injbot}(\mathrm{concat2}(AK, t_K, hostY)) = \mathrm{dec}(m_8, k_T)$ in

let $\mathrm{injbot}(\mathrm{pad}(=hostY, t_s)) = \mathrm{dec}(m_9, AK$ in

event $partTC(hostY, m_8, m_9)$;

find $j_3 \leq N_2$ suchthat

defined$(Mhost1[j_3], Mhost2, Pkey[j_3])$

$\wedge (Mhost1[j_3] = T \wedge Mhost2[j_3] = hostW)$ then

new $r_{SK} : keyseed$; let $SK = \mathrm{kgen}(r_{SK})$ innew $t_T : timest$;

new $r_{10} : seed$ let $e10 = \mathrm{enc}(\mathrm{concat2}(SK, t_T, hostY), Pkey[j_3], r_{10})$ in

new $r_{11} : seed$; let $e11 = \mathrm{enc}(\mathrm{concat3}(SK, n', t_T, hostW), AK, r_{11})$ in

event $fullTC(hostY, hostW, n', m_8, m_9, e10, e11)$;

$\overline{c_8[i_T]}\langle hostY, e10, e11, acceptT(hostY, hostW)\rangle$.

Figure 27: CryptoVerif formalization of TGS's actions.

$Q_S = !^{i_S \leq N}\, c_{11}[i_S](m_{14} : bitstring, m_{15} : bitstring);$

$\mathsf{let}\ \mathrm{injbot}(\mathrm{concat2}(SK, t_T, hostC)) = \mathrm{dec}(m_{14}, k_S)\ \mathsf{in}$

$\mathsf{let}\ \mathrm{injbot}(\mathrm{pad}(=hostC, t'_C)) = \mathrm{dec}(m_{15}, SK)\ \mathsf{in}$

$\mathsf{new}\ r_{16} : seed;\ \mathsf{let}\ e16 = \mathrm{enc}(\mathrm{padts}(t'_C), SK, r_{16})\ \mathsf{in}$

$\mathsf{event}\ partSC(hostC, m_{14}, m_{15});$

$\mathsf{event}\ fullSC(hostC, m_{14}, m_{15}, e16);$

$\overline{c_{12}[i_S]}\langle e16, acceptS(hostC)\rangle.$

Figure 28: CryptoVerif formalization of server's actions.

$Q_{Ckey} = !^{i_u \leq N_2}\, c_{13}[i_u](Khost : client, Kkey : key);$

let $Rkey : key =$

if $Khost = C$ then k_C else

$Kkey.$

$Q_{Tkey} = !^{i_v \leq N_2}\, c_{21}[i_v](Lhost : tgs, Lkey : key);$

let $Qkey : key =$

if $L_{host} = T$ then k_T else

$Lkey.$

$Q_{Skey} = !^{i_w \leq N_2}\, c_{16}[i_w](Mhost1 : tgs, Mhost2 : server, Mkey : key);$

let $Pkey : key =$

if $Mhost1 = T \wedge Mhost2 = S$ then k_S else

$Mkey.$

Figure 29: Processes storing keys (for honest and dishonest clients and server)

$$
\begin{aligned}
&!^{i' \leq n''} \mathsf{new}\ r : keyseed; (() \rightarrow \mathrm{pkgen}(r),\\
&(!^{i \leq n}(m : bitstring) \rightarrow \mathrm{pdec}(m, \mathrm{skgen}(r))),\\
&!^{i \leq n'}(x : blocksize, y : pkey) \rightarrow \mathsf{new}\ r' : seed; \mathrm{penc}(x, y, r')\ [all]\\
\approx\ &!^{i' \leq n''} \mathsf{new}\ r : keyseed; (() \rightarrow \mathrm{pkgen}'(r),\\
&(!^{i \leq n}(m : bitstring) \rightarrow\\
&\quad \mathsf{find}\ j \leq n'\ \mathsf{suchthat}\ \mathsf{defined}(m'[j], y[j], x[j])\\
&\qquad \wedge\, y[j] = \mathrm{pkgen}'(r) \wedge m = m'[j]\\
&\quad \mathsf{then}\ \mathrm{injbot}(x[j])\ \mathsf{else}\ \mathrm{pdec}'(m, \mathrm{skgen}'(r))),\\
&!^{i \leq n'}(x : blocksize, y : pkey) \rightarrow\\
&\quad \mathsf{find}\ k \leq n''\ \mathsf{suchthat}\ \mathsf{defined}(r[k]) \wedge y = \mathrm{pkgen}'(r[k])\\
&\quad \mathsf{then}\ (\mathsf{new}\ r' : seed; \mathsf{let}\ m' : bitstring = \mathrm{penc}'(Z, y, r')\ \mathsf{in}\ m'\\
&\quad \mathsf{else}\ \mathsf{new}\ r'' : seed; \mathrm{penc}(x, y, r''')
\end{aligned}
\qquad \text{(IND-CCA2)}
$$

Figure 30: Definition of IND-CCA2 public-key encryption in CryptoVerif

$$
\begin{array}{l}
!^{i' \leq n''}\mathsf{new}\ r : skeyseed; (() \rightarrow \mathrm{spkgen}(r), \\
\quad (!^{i \leq n}(x : sblocksize) \rightarrow \mathsf{new}\ r' : sseed;\ \mathrm{sign}(x, \mathrm{sskgen}(r), r'), \\
\quad !^{i \leq n'}(m : sblocksize,\ y : spkey,\ si : sinature) \rightarrow check(m, y, si)\ [all] \\
\approx !^{i' \leq n''}\mathsf{new}\ r : skeyseed; (() \rightarrow \mathrm{spkgen}'(r), \\
\quad (!^{i \leq n}(m : sblocksize) \rightarrow \mathsf{new}\ r' : sseed; \mathrm{sign}'(x, \mathrm{sskgen}'(r), r')) \qquad ((\text{UF-CMA}) \\
\quad !^{i \leq n'}(m : sblocksize,\ y : spkey,\ si : sinature) \rightarrow \\
\qquad \mathsf{find}\ j \leq n', k \leq n''\ \ \mathsf{suchthat}\ \mathsf{defined}(x[j,k], r[k]) \\
\qquad\quad \wedge y = \mathrm{spkgen}'(r[k]) \wedge m = x[j,k] \wedge \mathrm{check}'(m, y, si)\ \mathsf{then}\ \mathrm{true}\ \mathsf{else} \\
\qquad \mathsf{find}\ k \leq n''\ \mathsf{suchthat}\ \mathsf{defined}(r[k]) \wedge y = \mathrm{spkgen}'(r[k]) \\
\qquad\quad \mathsf{then}\ \mathrm{false}\ \mathsf{else}\ \mathrm{check}(m, y, si) \\
\\
!^{i' \leq n'}\mathsf{new}\ r : symkeyseed;\ (!^{i \leq n}(x : maxmac) \rightarrow \mathrm{mac}(x, \mathrm{kgen}(r)), \\
\quad (!^{i \leq n''}(m : maxmac, ma : macs) \rightarrow \mathrm{checkmac}(m, \mathrm{kgen}(r), ma)) \\
\approx !^{i' \leq n'}\mathsf{new}\ r : symkeyseed;\ (!^{i \leq n}(x : maxmac) \rightarrow \mathrm{mac}'(x, \mathrm{kgen}'(r)) \\
\quad (!^{i \leq n''}(m : maxmac, ma : macs) \rightarrow \qquad ((\text{W})\text{UF-CMA}) \\
\qquad \mathsf{find}\ j \leq n\ \ \mathsf{suchthat}\ \mathsf{defined}(x[j]) \\
\qquad\quad \wedge (m = x[j]) \wedge \mathrm{checkmac}'(x[j], \mathrm{kgen}'()r, ma) \\
\qquad \mathsf{then}\ \mathrm{true}\ \mathsf{else}\ \mathrm{false})
\end{array}
$$

Figure 31: Definition of UF-CMA signature und (W)UF-CMA message authentication code in CryptoVerif

$!^{i' \leq n''}$ new $r : protkey$;

$!^{i' \leq n}((x : usenum) \rightarrow \text{keyderivation}(z, x)$

$\approx !^{i' \leq n''}$

$!^{i' \leq n}((x : usenum) \rightarrow$ (Key Deriv)

find $u \leq n$ suchthat defined$(x[u], r[u]) \wedge x = x[u]$

then $r[u]$

else new $s : symkeyseed$; let$r : key = \text{kgen}(s)$ in r

Figure 32: Definition of pseudo-random key-derivation function in CryptoVerif